My First Attempt at Building a Data Warehouse

Contents:

1. INTRO

This project represents my journey in learning SQL and building a data warehouse in SQL Server through a guided project delivered by Baraa Khatib Salkini. The project was completed as part of Baraa's course 'The Complete SQL Bootcamp'.

I strongly recommend the course for anyone seeking to learn SQL and the basics of data warehouse design.

Project Overview

I built a three-layer Medallion Architecture in SQL Server, ingesting and integrating data from fictitious CRM and ERP source systems and applying extensive data-quality checks, cleansing rules, and key-mapping logic.

In the Silver and Gold layers, I standardized and joined cross-system customer, product, and sales data, and designed a Star Schema with dimension and fact views to support analytics.

Throughout the project, I used SQL for modeling and transformation, Git for version control, and Draw.io for architectural and lineage documentation.

The final warehouse delivers a fully validated, analytics-ready dataset and demonstrates practical competency in SQL, data modeling, and data product development.

2. Project Resources & Plan Development

Project Resources:

Data

- Cust_info.csv
- Prd_info.csv
- sales_details.csv
- CUST_AZ12.csv
- LOC_A101csv
- PX_CAT_G1V2.csv

Data / Code Repository: GitHub
Project Management Tool: Notion
Diagram Development: draw.io

What I did to get started:

1. Obtained the project data from the instructor and saved locally on my machine
2. Created a Github account.
3. Downloaded and installed draw.io for desktop. Draw.io was used for the architecture diagram for this project.
4. Created a Notion account and set up high-level project plan with tasks for the project. Used the collection of Epics and tasks provided by the instructor, but made some tweaks as the project progressed..

Project Plan As Built in Notion

Name	Progress	DWH Tasks
Requirements Analysis	100.00%	Analyze & Understand the Requirements
Design Data Architecture	100.00%	Choose Data Management Approach Brainstorm & Design the Layers Draw the Data Architecture (Draw.io)
Project Initialization	100.00%	Prepare the Git Repo & Prepare the Structure Create DB & Schemas, Create Detailed Project Tasks (Notion) Define Project Naming Conventions
Build Bronze Layer	100.00%	Analyzing: Source Systems, Coding: Data Ingestion Validating: Data Completeness & Schema Checks Document: Draw Data Flow (Draw.io), Commit Code in Git Repo
Build Silver Layer	100.00%	Analyzing: Explore & Understand Data, Coding: Data Cleansing Validating: Data Correctness Checks Documenting & Versioning in GIT, Commit Code in GIT Repo Document: Draw Data Integration (Draw.io)
Build Gold Layer	100.00%	Analyzing: Explore Business Objects, Coding: Data Integration Validating: Data Integration Checks Document: Draw Data Model of Star Schema (Draw.io) Document: Create Data Catalog
Exploratory SQL Data Analysis	0.00%	Change Over Time Analysis, Dimensions vs Measures Database Exploration, Dimensions Exploration Date Exploration, Measure Exploration, Magnitude Analysis Ranking Analysis
SQL Advanced Analytics	0.00%	Change Over Time Analysis, Cumulative Analysis Performance Analysis, Part to Whole Analysis Data Segmentation, Build Customers Report Build Products Report, Document in GIT

3. Project Requirements

Objective

Develop a modern data warehouse using SQL Server to consolidate sales data, enabling analytical reporting and informed decision-making.

Specifications

- **Data Sources**: Import data from two source systems (ERP and CRM) provided as CSV files.
- **Data Quality**: Cleanse and resolve data quality issues prior to analysis.
- **Integration**: Combine both sources into a single, user-friendly data model designed for analytical queries.
- **Scope**: Focus on the latest dataset only; historization of data is not required.
- **Documentation**: Provide clear documentation of the data model to support both business stakeholders and analytics teams.

BI: Analytics & Reporting (Data Analysis)

Objective

Develop SQL-based analytics to deliver detailed insights into:

- Customer Behavior
- Product Performance
- Sales Trends

These insights empower stakeholders with key business metrics, enabling strategic decision-making.

4. Designing the Data Architecture

a. Choosing the Data Management Approach

In this task, I evaluated the requirements of the project and selected Data Warehouse as the appropriate approach from the following data architectures:

- Data Warehouse
- Data Lake
- Data Lakehouse
- Data Mesh

Next I learned about the types of data warehouse approaches, evaluated each against the requirements and selected the Medallion Architecture approach as it is most appropriate for this effort

- **Inmon**: Start with staging, next layer data is modeled in an EDW in 3NF, then broken into a third layer of data marts, then data BI tools are connected to the data marts
- **Kimbal**: Similar to Inmon, however, there is no EDW and data is moved from staging directing to the data marts as the EDW is time intensive. Kimbal is a faster approach than Inmon, but it can lead to chaos in the data mart with duplicate efforts
- **Data Vault**: Similar to Inmon however, the EDW layer is split into a Raw Vault (original data) and Business Vault (includes the business rules and transformations which prepare data for data marts)
- **Medallion Architecture**: Comprised of three layers (bronze, silver, and gold). Bronze Layer: Original data as-is. Enables traceability and issue identification; Silver: Transformed and cleansed data. Gold Layer: Similar to Data Mart in which business-ready objects are built and ready to share as products (marchine learning, AI, reports) with business rules / logic applied.

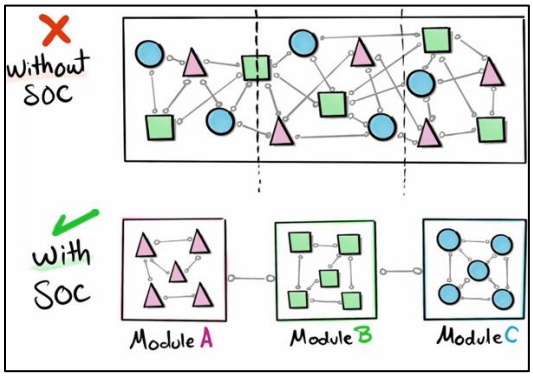
b. Define Architecture Layers

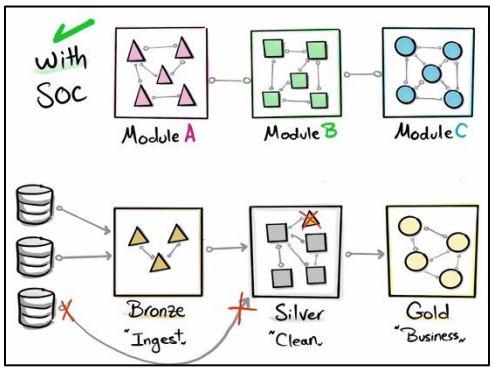
Summary:

Here I documented the definition and objective for each layer (Bronze, Silver, and Gold)

	Bronze	Silver	Gold Layer
Definition	Raw, unprocessed data as-is from sources	Cleansed and standardized data	Business-Ready Data
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 (not required as there are no tables
Data Transformation	None (as-is)	• Data Cleansing • Data Standardization • Data Normalization • Derived Columns • Data Enrichment	• Data Integration • Data Aggregation • Business Logic & Rules
Data Modeling	None (as-is)	None (as-is)	• Start Schema • Aggregated Objects • Flat Tables
Target Audience	Data Engineers	Data Analysts Data Engineers	Data Analysts Business Users

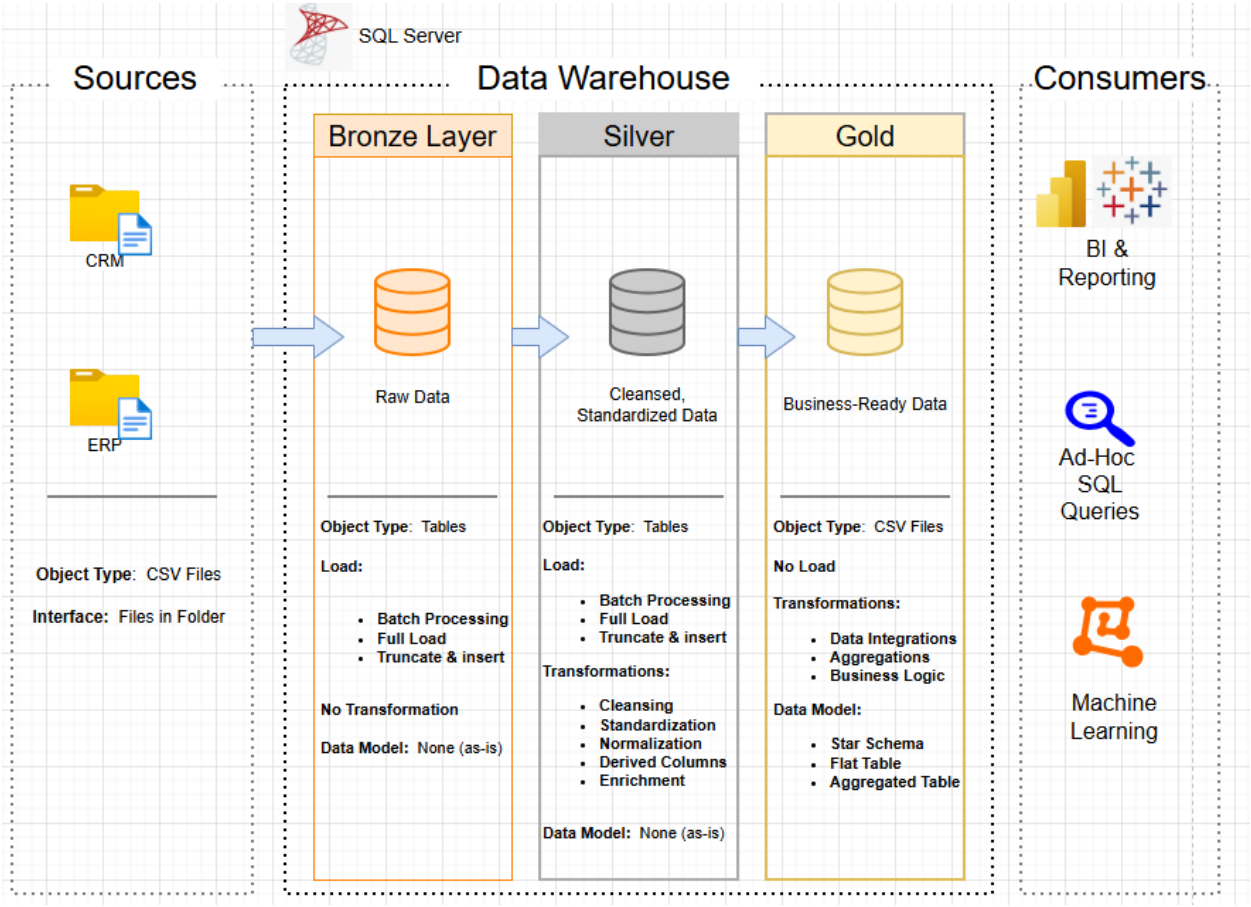
DOCUMENTING DETAIL IN THE TABLE ABOVE ENABLED SEPARATION OF CONCERNS (SOCs): Breaking the system into separate components with specific purposes with no overlap





c. Build High-Level Architecture Diagram

The objective here was to draw the data architecture using Draw.io according to the requirements, medallion architecture, and defined design layers within the architecture



5. Project Initialization

a. Define Naming Conventions

Here I defined the set of rules for naming everything in the project to ensure standardization so all members of the project are using the same naming conventions. This is done for:

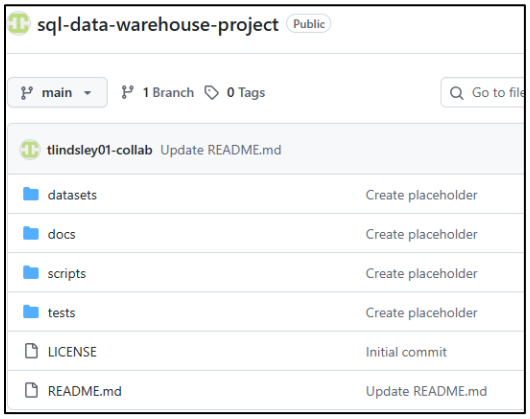
- The Database
- Schema
- Tables
- Stored Procedures, etc

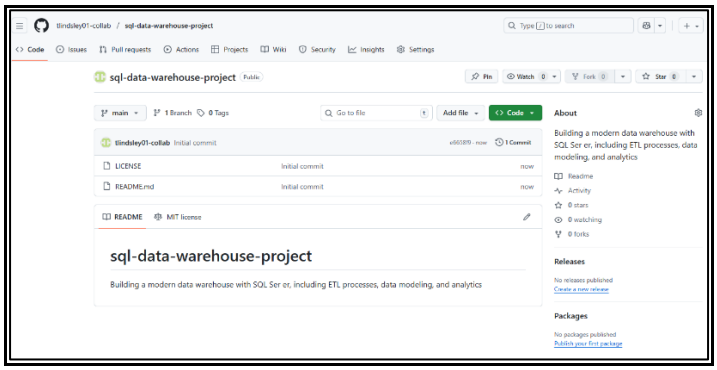
b. Create Git Repo

For this task, I created a Github account, and learned how to create a repo structure for a project. A Git repo (repository) is a safe location for storing and tracking code used for a project and collaborating with the team. Additionally you can share your repository as part of your portfolio to show that you know how to build and maintain a well-documented git repository for your project.

I created my first repository in Git hub through this effort and gave it the name 'sql-data-warehouse-project'. I then began creating folders and placeholders required for the project.

The project content structure in Git





c. Create Schemas

Below are the steps I took to create the database:

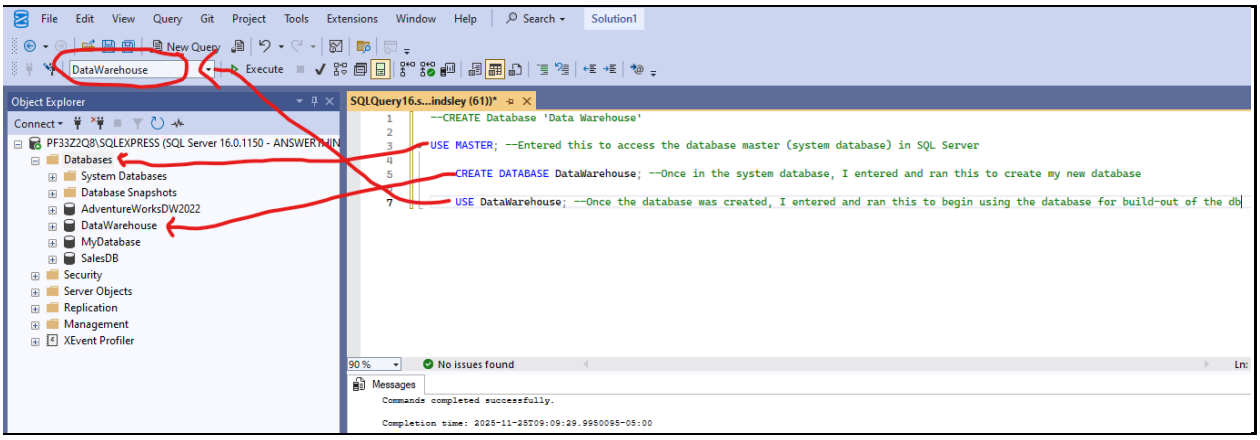
1. Open SQL Server Management Studio

2. Access the database master (system database) by creating a new query and entering: "USE MASTER"

3. Created the database with the following script in the same query window: "CREATE DATABASE DataWarehouse";

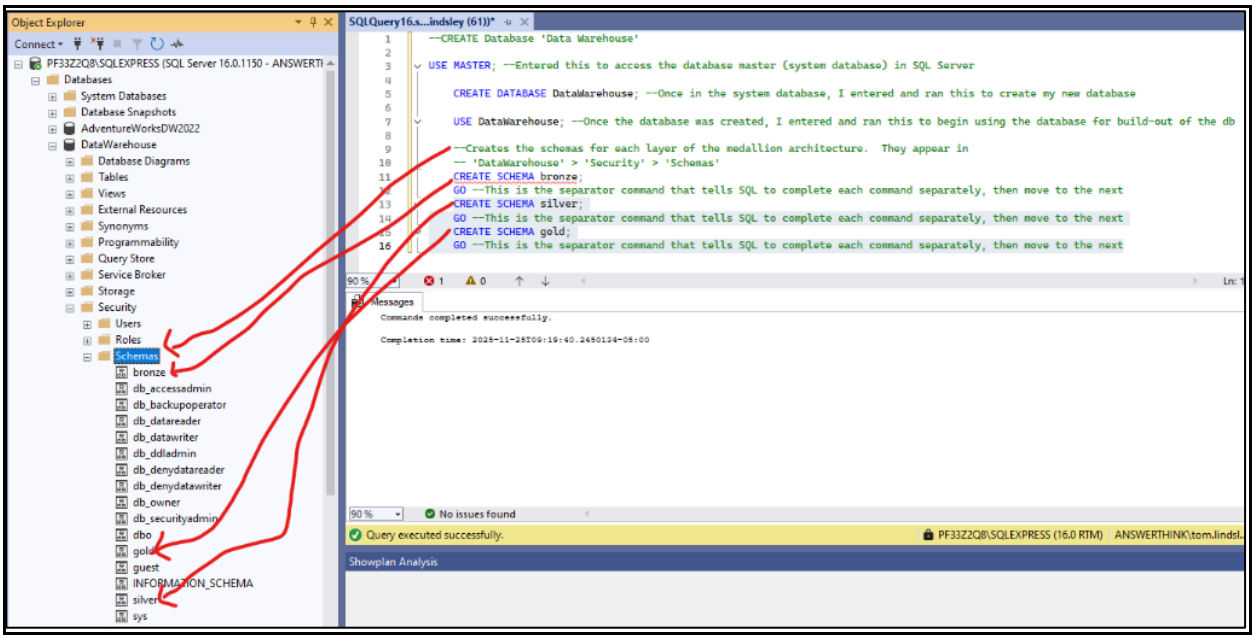
4. I then with to the object explorer on the left hand side of SQL Server Management Studio, right clicked on 'Databases' and selected 'Refresh'. The new database 'DataWarehouse' now appears in the collection of databases.

5. I now need to begin building the database. To do so, I entered ‘USE DataWarehouse’ in the same query window. The screenshot below shows the queries and outputs of steps 1-5



6. Next I executed the following query to create the schema for each layer of the medallion architecture (Bronze, Silver, and Gold)

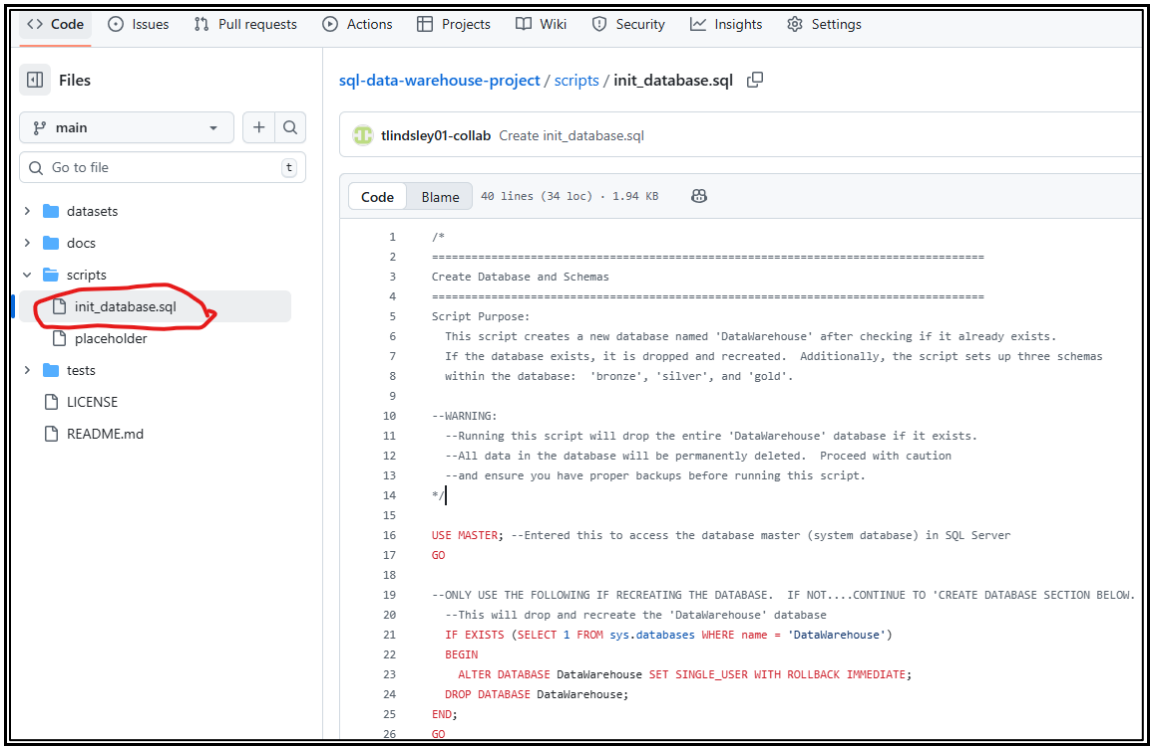
Schema Creation SQL



7. Once Schemas were built for each layer, I then returned to Github to commit the code to my repo using the steps listed below. Completion of these steps also resulted in the completion of all tasks within the 'Project Initialization' EPIC in my project plan:

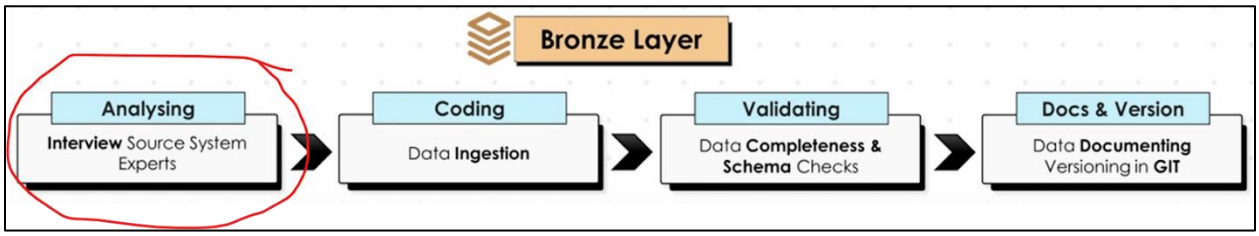
a. Opened the 'scripts' folder in my sql-datawarehouse-project
b. Selected 'Add file'
c. Selected 'Create New File'
d. Assigned file name 'init_database.sql'
e. Pasted the code from SQL Server to document the steps taken to execute steps 1-6
f. Added cautionary and informative commentary to the file to provide users of the script with and understanding of what the code will do and considerations before executing.
g. Finally, I committed the code by selecting 'Commit Changes'

Code Commit in Git



6. Bronze Layer Build

a. Analyzing Source Systems



The first step in building the layer is to understand the source system. To do so in a normal project, I would conduct interviews with the source system owners / SMEs to understand the nature of the system to be connected to the warehouse. As this is a fictional project with fictional systems….no interviews were conducted

Conducting the interview allows you to understand:

- Business Context & Ownership
- Who owns the data
- What business processes it supports
- System and data documentation
- Data model and data catalog for the source system

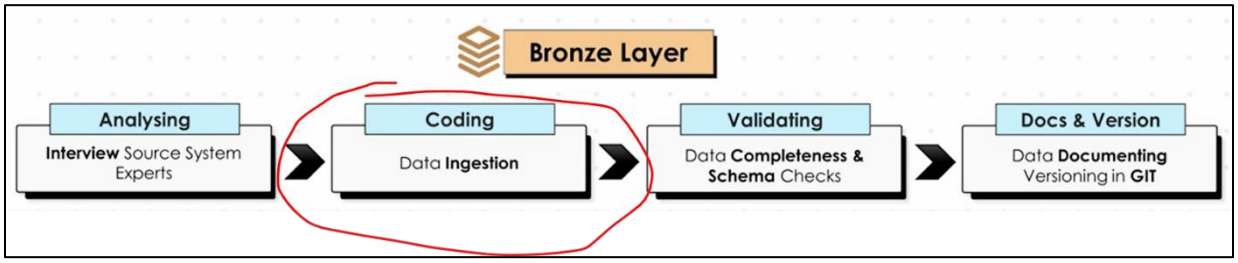
Architecture & Technology Stack

- How is the data stored? (On prem in SQL Server or Oracle. In the cloud on Azure or AWS)
- What are the integration capabilities of the source system? Is it through APIs, Kafka, File Extract, Direct DB connection?)

Extract & Load

- Can we do incremental loads vs full loads?
- Data scope & historical needs?
- Expected extract size (gigabytes / terrabytes). Important for understanding whether we have the right tools and platforms to connect and extract source system data.
- Are there data volume limitations of the source system
- How do we avoid impacting the source system's performance in the connection and extraction
- What are the authentication and authorization requirements of the source system that need to be adhered to / made available to access and extract the data (tokens, SSH keys, VPN, IP whitelisting….)

b. Coding: Data Ingestion



The first step in this task is to understand the meta data, the structure, the schema of the incoming data before defining the structure of the tables for the Bronze layer in DDL (data definition language). This can be accomplished through the interviews with the source system SMEs / owners ***and*** through data

profiling / exploring the data on your own to identify column names and data types. If exploring on your own first, ensure that your assumptions are validated by the system owners / SMEs before building the bronze layer.

The following steps were used in this project to gain this understanding of the data sets provided for this project.

1. Open the 'cust_info.csv' data file for CRM and identify the column names, review the data and determine the data types

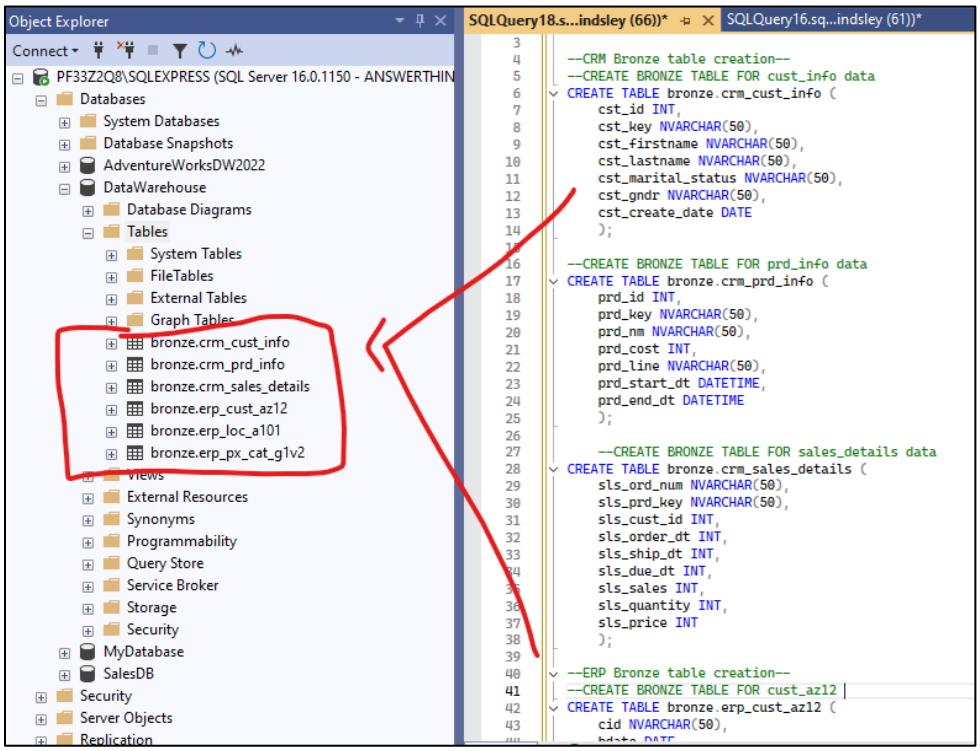
2. Open a new query in 'DataWarehouse' database and enter the following command to create the table in the bronze layer of the database (adhering to naming conventions defined previously in this project):
CREATE TABLE bronze.crm_cust_info

3. In the same query, create the following columns and assign their data types based on what is understood from the provided data set. The names should be identical to how they are defined in the data set

```
cst_id INT,
cst_key NVARCHAR(50),
cst_firstname NVARCHAR(50),
cst_lastname NVARCHAR(50),
cst_marital_status NVARCHAR(50),
cst_gndr NVARCHAR(50),
cst_create_date DATE
```

4. Execute the query, then refresh the tables in the database to confirm creation.
5. Execute steps 1 - 3 for all remaining files for both source systems

Bronze Layer Table Creation



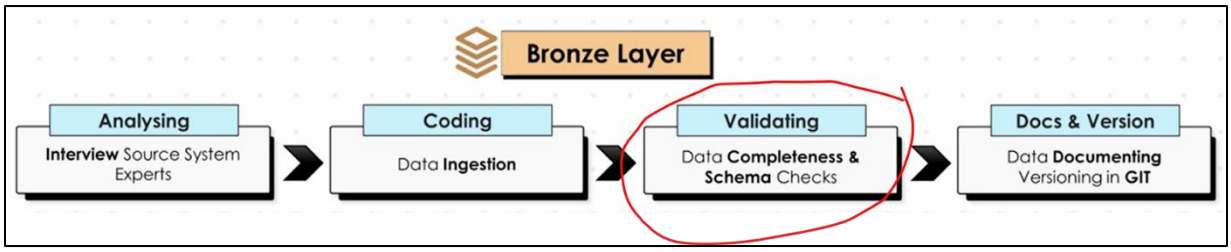
6. Loading the data: We are doing a bulk insert directly from files into the database. A single operation in one go. I wrote the script to load the cust_info data file into the bronze.crm_cust_info' table.

NOTE: I learned that this script will encounter an error if the file is open by a user. I had the file open when I attempted to execute the script and encountered just that. Closed the file and executed again successfully

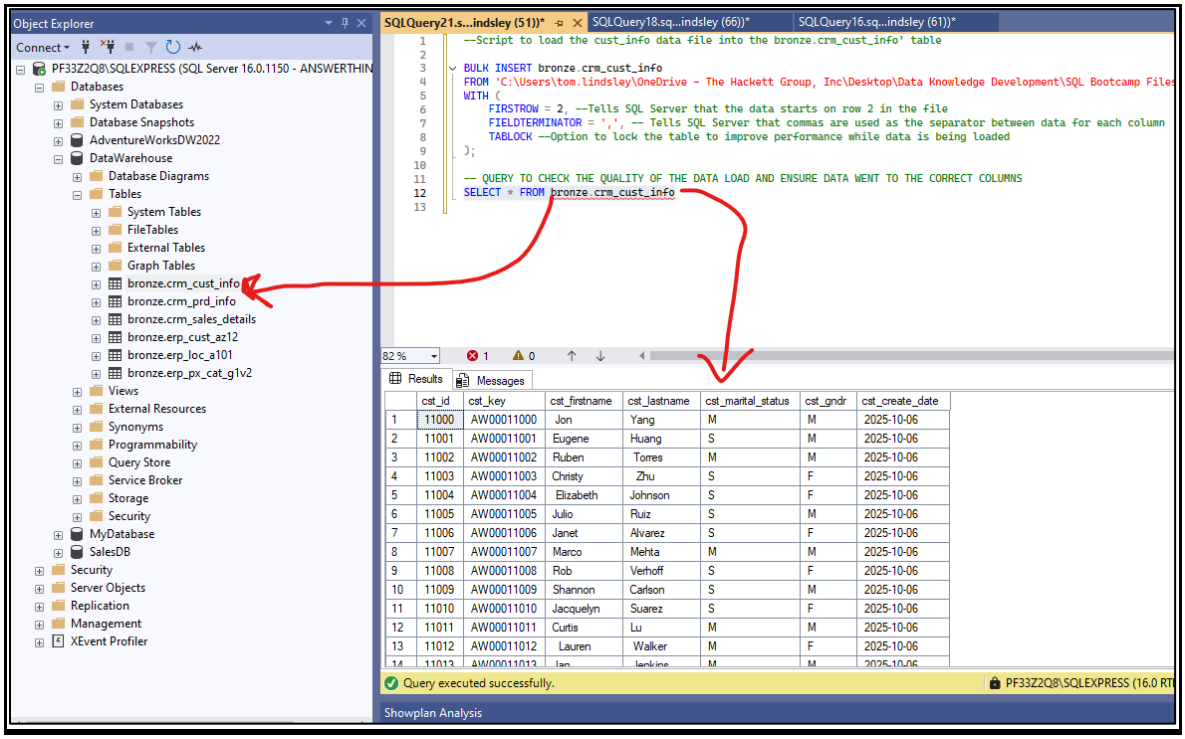
```
--Script to load the cust_info data file into the bronze.crm_cust_info' table

BULK INSERT bronze.crm_cust_info
FROM 'C:sql-data-warehouse-project\datasets\source_crm\cust_info.csv'
WITH (
        FIRSTROW = 2, --Tells SQL Server that the data starts on row 2 in the file
        FIELDTERMINATOR = ',', -- Tells SQL Server that commas are used as the separator between data for each column
        TABLOCK --Option to lock the table to improve performance while data is being loaded
);
```

c. Validating: Data Completeness & Schema Checks

This was essentially a continuation of the previous tasks conducted in 'Coding'



a. The next step was to run a query to view the data in the table to confirm that the data was pulled into the correct columns. I ran the following: SELECT * FROM bronze.crm_cust_info



	cst_id	cst_key	cst_firstname	cst_lastname	cst_marital_status	cst_gndr	cst_create_date
1	11000	AW00011000	Jon	Yang	M	M	2025-10-06
2	11001	AW00011001	Eugene	Huang	S	M	2025-10-06
3	11002	AW00011002	Ruben	Torres	M	M	2025-10-06
4	11003	AW00011003	Christy	Zhu	S	F	2025-10-06
5	11004	AW00011004	Elizabeth	Johnson	S	F	2025-10-06
6	11005	AW00011005	Julio	Ruiz	S	M	2025-10-06
7	11006	AW00011006	Janet	Alvarez	S	F	2025-10-06
8	11007	AW00011007	Marco	Mehta	M	M	2025-10-06
9	11008	AW00011008	Rob	Verhoff	S	F	2025-10-06
10	11009	AW00011009	Shannon	Carlson	S	M	2025-10-06
11	11010	AW00011010	Jacquelyn	Suarez	S	F	2025-10-06
12	11011	AW00011011	Curtis	Lu	M	M	2025-10-06
13	11012	AW00011012	Lauren	Walker	M	F	2025-10-06

b. The previous steps were only a portion of the full code that needed to be written. The command needed to include a 'TRUNCATE' command to enable bulk load if the data is refreshed in the source file

without adding to the existing data in the table. Truncating will replace existing data with the new / refreshed data source. So….the command was rewritten and run again

```
--Script to load the cust_info data file into the bronze.crm_cust_info' table

TRUNCATE TABLE bronze.crm_cust_info; --Empties existing data in the table, then loads with refreshed data from scratch.
--This is a full load.

BULK INSERT bronze.crm_cust_info
FROM 'C:\Users\tom.lindsley\OneDrive - The Hackett Group, Inc\Desktop\Data Knowledge Development\SQL Bootcamp Files\sql-data-warehouse-project\datasets\source_crm\cust_info.csv'
WITH (
        FIRSTROW = 2, --Tells SQL Server that the data starts on row 2 in the file
        FIELDTERMINATOR = ',', -- Tells SQL Server that commas are used as the separator between data for each column
        TABLOCK --Option to lock the table to improve performance while data is being loaded
);

-- QUERY TO CHECK THE QUALITY OF THE DATA LOAD AND ENSURE DATA WENT TO THE CORRECT COLUMNS
SELECT * FROM bronze.crm_cust_info -- QUERY TO CHECK THE QUALITY OF THE DATA LOAD AND ENSURE DATA WENT TO THE CORRECT COLUMNS

SELECT COUNT(*) FROM bronze.crm_cust_info ---- QUERY TO CHECK THAT THE TABLE ROW COUNT MATCHES THE CSV FILE
```

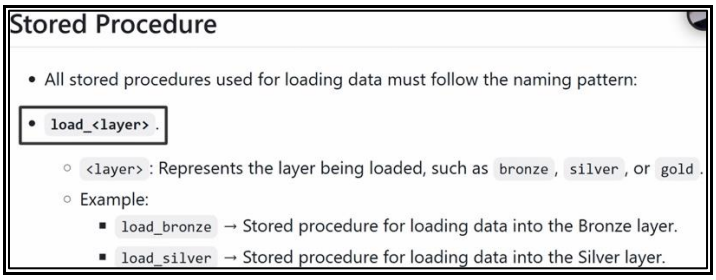
c. The same script was written and run for each data file for CRM and ERP data tables.

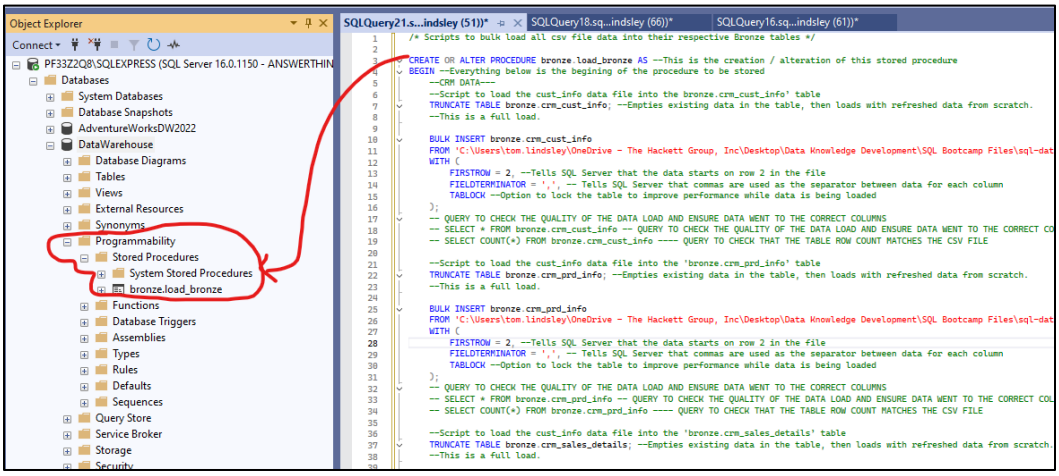
d. After running each script, I checked the results of the load against the source files to confirm that the data was placed into the correct columns and that the tables and the data sources had the same number of rows.

e. With the checks complete, I was good to move on to creating a stored procedure as this script would need to be run on a recurring basis.

f. **Creating a stored procedure:**

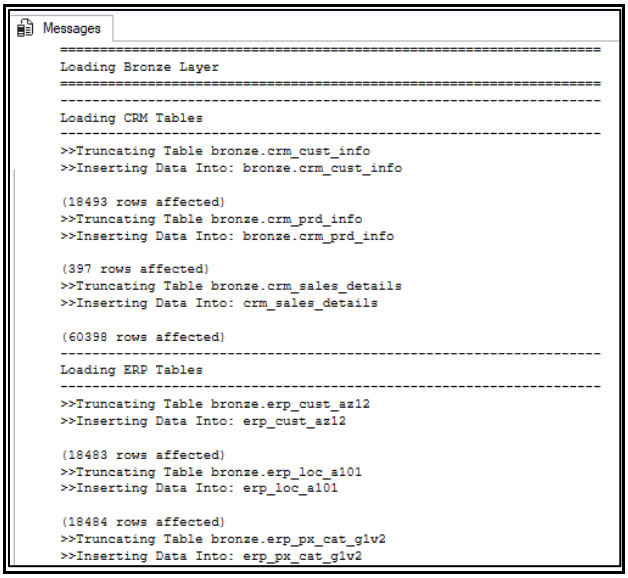
- I wrote a stored procedure for the loading of the data into the database as it is intended to be a recurring action to bring in the data.





- Once executing the stored procedure and refreshing the DB to visually confirm its existence, I then created a new query to test the execution of the procedure. The following SQL statement was used: EXEC bronze.load_bronze

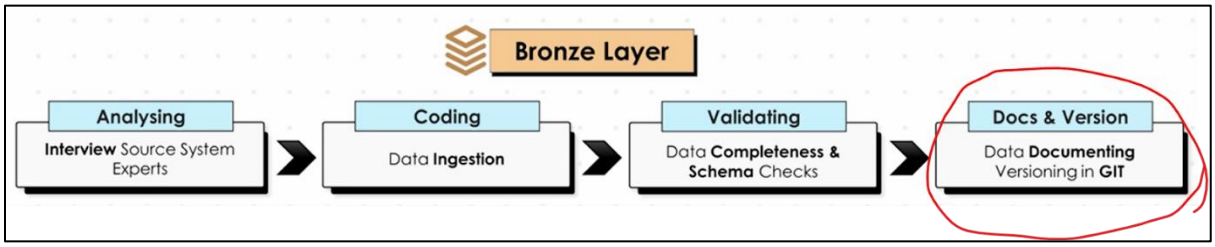
- The message output in SQL Server Studio is fairly vague. I edited the script in the procedure to enhance the information produced in the message output for greater clarity / confirmation of successful load to the intended tables. The result of the enhancement:



- The next step was to further enhance the stored procedure by adding a TRY CATCH command to handle errors that might occur in execution

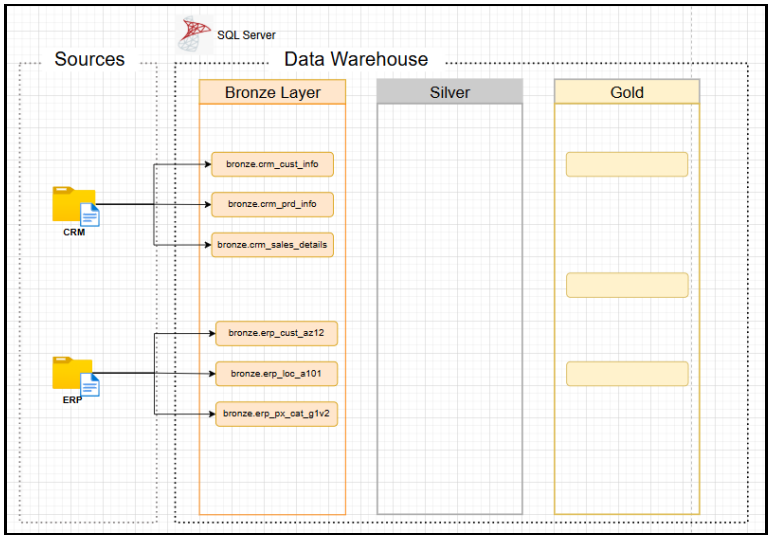
- I also added script to include the duration of the loading effort for each table. To do this I declared two variables just before the 'BEGIN TRY' command for start time and end time. These variables will be used to output the execution duration that will be included in the message detail

d. Document: Draw Data Flow (Draw.io)



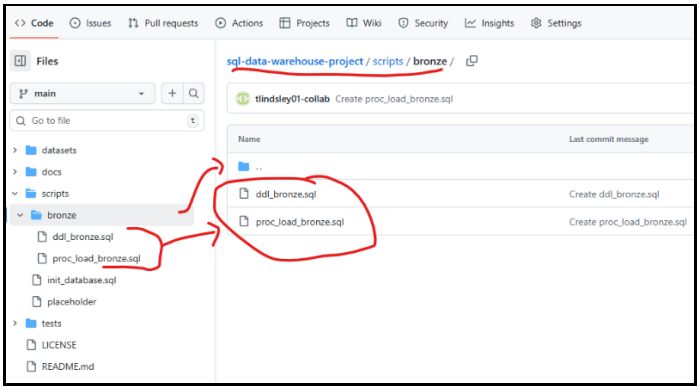
1. For the first step in this task, I drew a simple diagram in Draw.io of the flow of data from sources to the tables in the bronze layer to document data lineage. The output completed for the bronze layer is depicted below.

Initial Data Flow Diagram

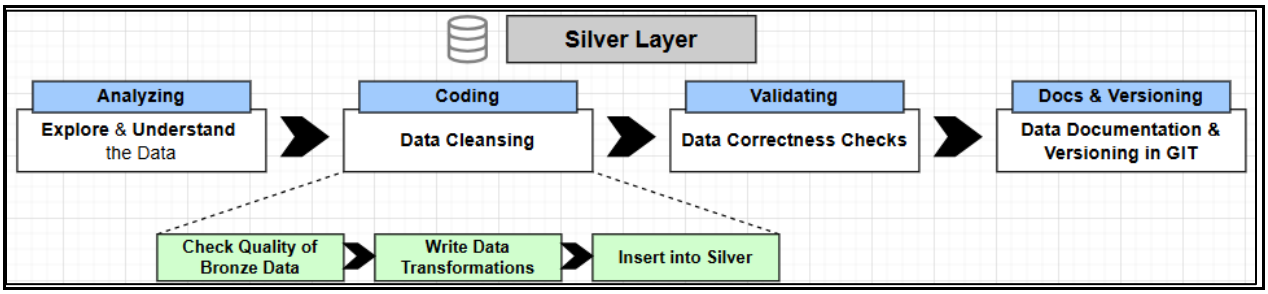


d. Commit Code to Git

The last task in this Epic is to commit my code into my Git repository. All of the code generated for table creation, and table loading for the bronze layer were added as separate files in the repository in a folder specific to the bronze layer.



7. Silver Layer Build

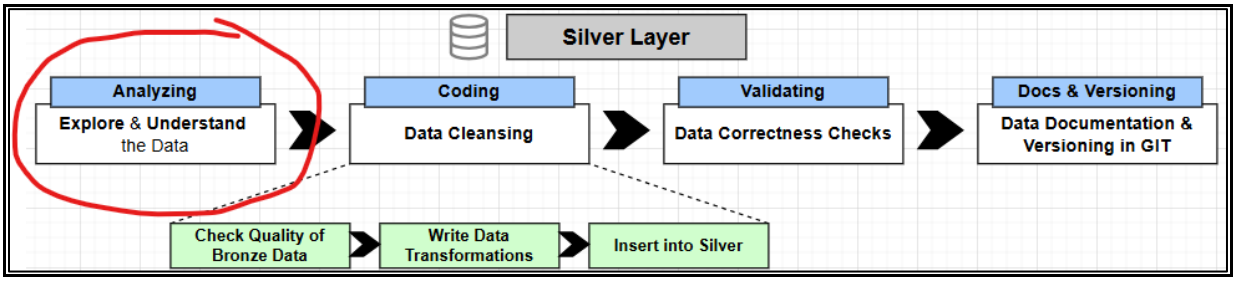


It is now time to build the Silver layer of the data warehouse now that the bronze layer has been built, documented, and its supporting code has committed to my Git repo.

This Epic is comprised of the following key tasks that will be completed in the order in which they are displayed below:

- Analyzing: Explore & Understand Data
- Coding: Data Cleansing
    - Check Quality of Bronze Data
    - Write Data Transformations
    - Insert into Silver Layer
- Validating: Data Correctness Checks
- Documenting & Versioning in GIT
- Commit Code in GIT Repo

a. Analyzing: Explore & Understand Data



Exploring and understanding the data is key to building out the Silver layer, the relationships, etc. To do this I explored each table one-by-one in the bronze layer.

For each table, I selected the top 1000 rows to get an initial understanding. This can be done by creating and executing the query below, or (in SQL Server) by new right clicking on the desired table and selecting 'Select Top 1000 Rows'.

A key step in the data exploration task is to build an integration model to document what I observe / assume about how the tables are related. This is important so that I would not later forget what I observed

CRM Data Tables

- **bronze.crm_cust_info**: This table contains customer info with standard customer detail such as first name, last name, customer creation date. It includes a customer id (cust_id) as the primary key as well as a customer key (cust_key) which seems duplicative. An initial view of this table indicates that both the cust_id and the cust_key could be used to define relationships with other tables.

- **bronze.crm_prd_info:** This table contains basic product information along with historical information (historical product pricing). It includes a product id (prd_id) that appears to serve as the primary key, but it also includes a product key (prd_key). Like the cust_info table, an initial view of this table indicates that two columns (prd_id and prd_key) could be used to define relationships with other tables. I will have to determine which is most appropriate as I review the other tables.

- **bronze.crm_sales_details:** This table contains historical transactional records about sales and orders. Each record includes an order number (sls_ord_num), a product key for the product in the order (sls_prd_key), the id of the customer that placed the order (sls_cust_id), as well as order date, ship date, total order sales, sales quantity, and sales price. It appears that the sls_ord_num is the primary key for this table.

  It appears that the sls_prd_key can be used to define the relationship with the crm_prd_info table but the value in the sls_prd_key for each record appears to only be a subset of the value listed for each record in the prd_key field of the crm_prd_info table.

  The data type and character count for records in the sls_cust_id field appear to be a match for the data type and character count for the cust_id field in the crm_cust_info table. Therefore I will use the sls_cust_id field to define the relationship with the crm_cust_info table matching to the cst_id field.

ERP Data Tables

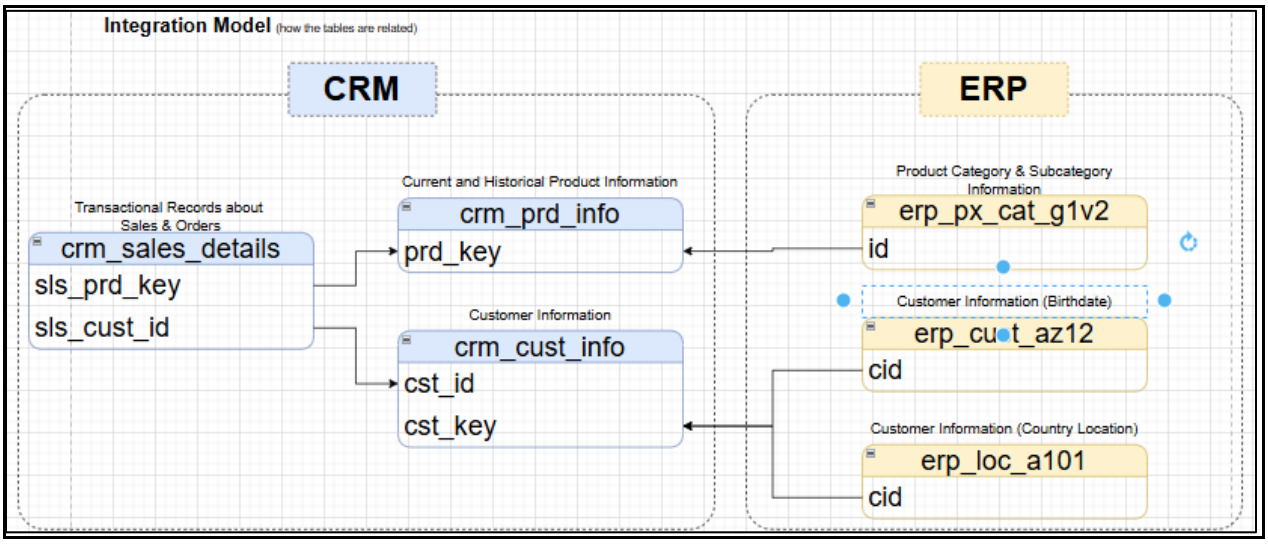
- **bronze.erp_cust_az12:** This table contains customer birth dates and gender. Its primary key appears to be the customer id field (cid). The values in this field for each record appear to be partial match to the customer key (cst_key) in the CRM table crm_cust_info, but it includes additional leading characters. No other field in this table could be used to create a relationship with any other table. So the customer id field in this table will be used to create a relationship with the crm_cust_info table by matching on its cst_key

- **bronze.erp_loc_a101:** This table contains country for each customer. Similar to the erp_cust_az12 table, its primary key appears to be the customer id field (cid) with partial match to the customer key (cst_key) in the CRM table crm_cust_info, but it includes "-" as separators for characters in this field. No other field in this table could be used to create a relationship with any other table. So the customer id (cid) field in this table will be used to create a relationship with the crm_cust_info table by matching on its cst_key

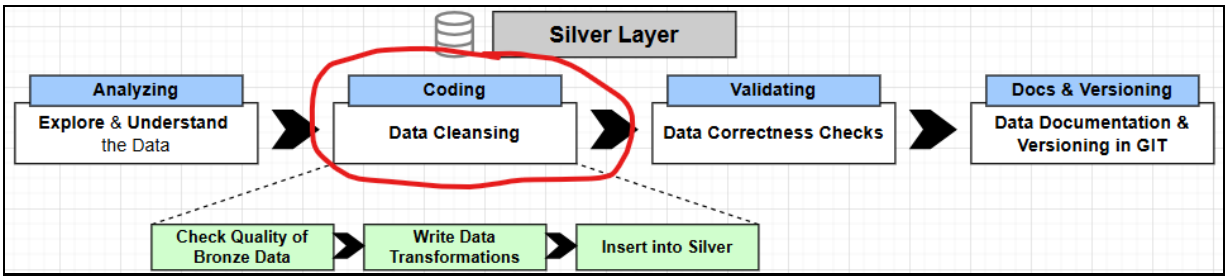
- **bronze.erp_px_cat_g1v2:** This table contains product categories, subcategories, and an identifier for "maintenance". Not sure what the "maintenance" field is used for, but it only includes "Yes" or "No" as values for each record. The primary key appears to be the "id" field and is comprised of five (5) characters →Example: 'AC_BR'. I assume this is the "id" for each product subcategory. The values in this field appear to be a match for the first five (5) characters in the prd_key field in the crm_prd_info table. This allows me to define a relationship between the two tables by matching on these fields. Doing so will require some transformation / manipulation first.

b. Document: High-Level Data Integration Diagram

As I explored the data in each table, I built the model below to help me visually understand the relationships that I will need to define.



c. Coding: Data Cleansing & Quality Checks



The first step in this task is to build the structure of the silver layer. It will be identical to the structure used for the Bronze layer. To accomplish this, I open the ddl script used for the bronze layer that I have saved in Git, paste it into a new query in SQL Server, then replace all references to ‘bronze’ with ‘silver.

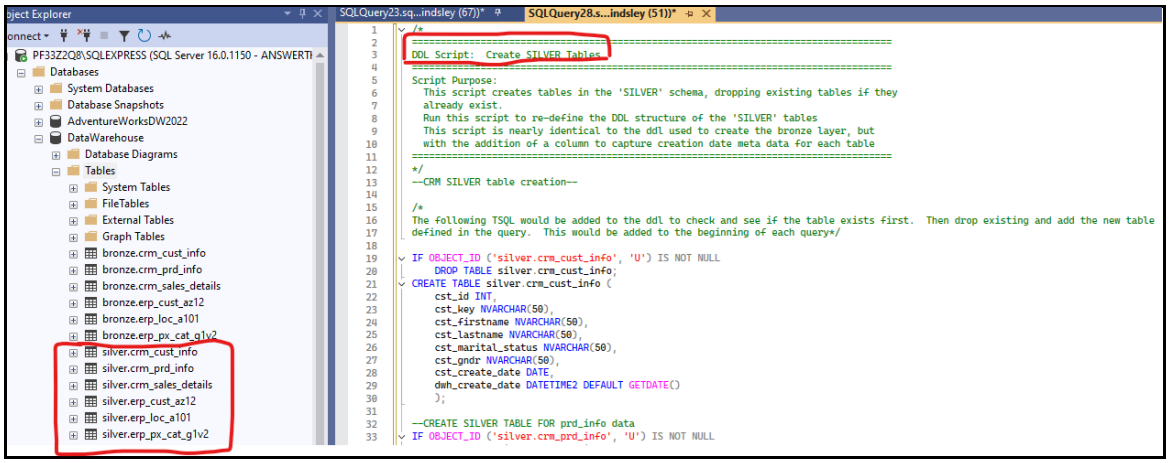
In addition to replicating the ddl to build the Silver layer, I also have to add metadata columns to each table to capture the following data. This information is needed to enable tracking and root cause analysis of corrupt data errors, data gaps in loads, etc:

- **create_date:** The record’s load timestamp. Added “**dwh_create_date DATETIME2 DEFAULT GETDATE()**” in each CREATE TABLE command as an additional column for every table

These columns were not created in this step for each table. Will need to determine if these will be added later.

- **update_date:** The records last update timestamp
- **source_system:** The origin system of the record
- **file_location:** The file source of the record (where it is stored and made available for loading into the db)

Creation and execution of the ddl script to build the silver layer successfully complete resulted in the silver layer tables being added to the Data Warehouse as depicted below:

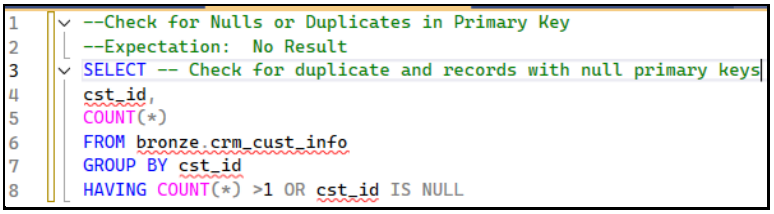


With the Silver layer tables built, the next step is to identify the data quality issues in each data set. This must be done before writing any transformation scripts. This exploration is done by evaluating each table in the bronze layer.

CRM → Customer Info Table

Checking for and Correcting Records with Duplicate and Null Primary Keys

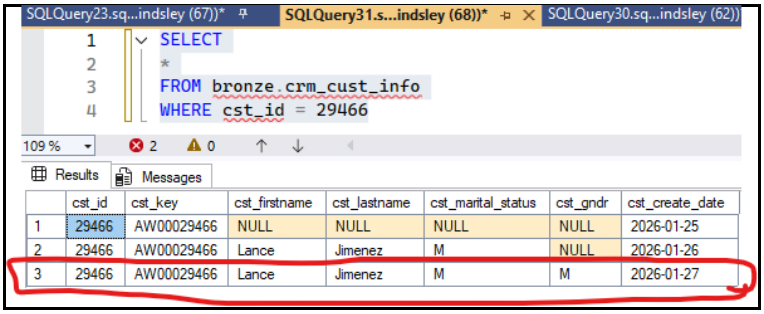
Primary Key Duplicate and Null For each table, the primary key was evaluated to determine whether or not there were NULL values or duplicates using the following script. The result was the identification of 6 customer ids with two or more duplicates. Three (3) of those records had null values in the cst_id field (no customer id)



```sql
--Check for Nulls or Duplicates in Primary Key
--Expectation: No Result
SELECT -- Check for duplicate and records with null primary keys
cst_id,
COUNT(*)
FROM bronze.crm_cust_info
GROUP BY cst_id
HAVING COUNT(*) >1 OR cst_id IS NULL
```

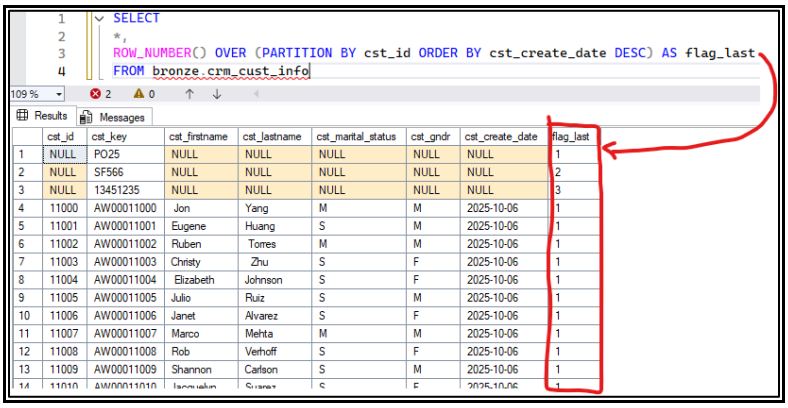
	cst_id	(No column name)
1	29449	2
2	29473	2
3	29433	2
4	NULL	3
5	29483	2
6	29466	3

Next I needed to create a new query to clean the data to correct / account for the duplicates and null values. Before starting the transformation query, I needed to explore the data a bit more. I started by focusing on one of the IDs (29466) that appears more than once and running a query to view the associated records. This review allowed me to see that cst_id 29466 appears for three different records in the data set. Each record has a different creation date. The most recent creation date was selected as the primary record in the dataset.



	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

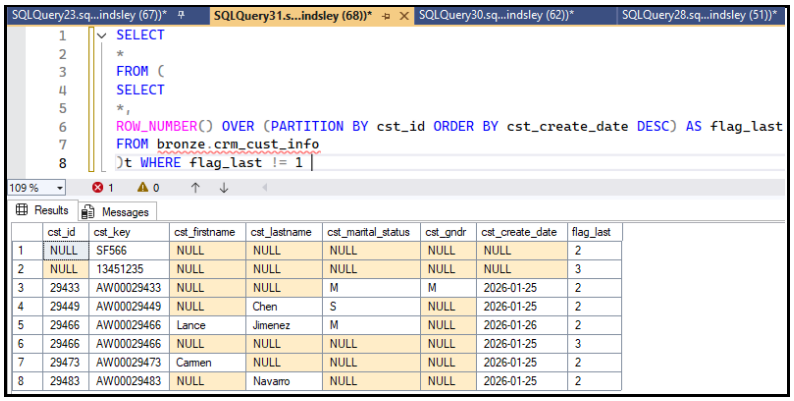
I assumed that all other duplicates likely had different creation dates. So, to set the most recently added records for duplicate cst_ids, I needed to rank them according to their creation date and assign a ranking number. The following query was used to rank all records in order by their creation date with the assignment of “1” for the most recent. Records with the assignment of “1” will be used as the primary records from the data set.



```
SELECT
    *,
    ROW_NUMBER() OVER (PARTITION BY cst_id ORDER BY cst_create_date DESC) AS flag_last
    FROM bronze.crm_cust_info
```

	cst_id	cst_key	cst_firstname	cst_lastname	cst_marital_status	cst_gndr	cst_create_date	flag_last
1	NULL	PO25	NULL	NULL	NULL	NULL	NULL	1
2	NULL	SF566	NULL	NULL	NULL	NULL	NULL	2
3	NULL	13451235	NULL	NULL	NULL	NULL	NULL	3
4	11000	AW00011000	Jon	Yang	M	M	2025-10-06	1
5	11001	AW00011001	Eugene	Huang	S	M	2025-10-06	1
6	11002	AW00011002	Ruben	Torres	M	M	2025-10-06	1
7	11003	AW00011003	Christy	Zhu	S	F	2025-10-06	1
8	11004	AW00011004	Elizabeth	Johnson	S	F	2025-10-06	1
9	11005	AW00011005	Julio	Ruiz	S	M	2025-10-06	1
10	11006	AW00011006	Janet	Alvarez	S	F	2025-10-06	1
11	11007	AW00011007	Marco	Mehta	M	M	2025-10-06	1
12	11008	AW00011008	Rob	Verhoff	S	F	2025-10-06	1
13	11009	AW00011009	Shannon	Carlson	S	M	2025-10-06	1

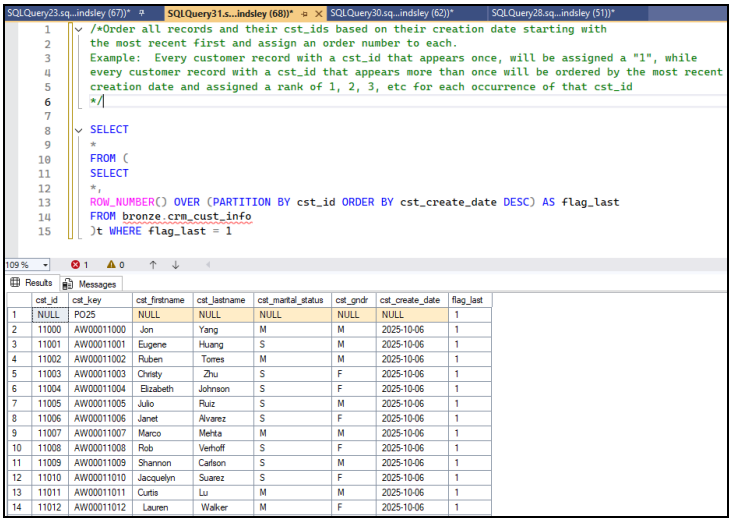
Creating and running the query below allowed me to see all records that are duplicates (or nulls) with ranking identifiers not equal to “1”. These will not be used in my data set for this table.



```
SELECT
    *
    FROM (
    SELECT
    *,
    ROW_NUMBER() OVER (PARTITION BY cst_id ORDER BY cst_create_date DESC) AS flag_last
    FROM bronze.crm_cust_info
    )t WHERE flag_last != 1
```

	cst_id	cst_key	cst_firstname	cst_lastname	cst_marital_status	cst_gndr	cst_create_date	flag_last
1	NULL	SF566	NULL	NULL	NULL	NULL	NULL	2
2	NULL	13451235	NULL	NULL	NULL	NULL	NULL	3
3	29433	AW00029433	NULL	NULL	M	M	2026-01-25	2
4	29449	AW00029449	NULL	Chen	S	NULL	2026-01-25	2
5	29466	AW00029466	Lance	Jimenez	M	NULL	2026-01-26	2
6	29466	AW00029466	NULL	NULL	NULL	NULL	2026-01-25	3
7	29473	AW00029473	Carmen	NULL	NULL	NULL	2026-01-25	2
8	29483	AW00029483	NULL	Navarro	NULL	NULL	2026-01-25	2

Creating and running the query below allowed me to see all records with ranking identifiers equal to “1”. These records will be used in my data set for this table.



```
/*Order all records and their cst_ids based on their creation date starting with
the most recent first and assign an order number to each.
Example:  Every customer record with a cst_id that appears once, will be assigned a "1", while
every customer record with a cst_id that appears more than once will be ordered by the most recent
creation date and assigned a rank of 1, 2, 3, etc for each occurrence of that cst_id
*/

SELECT
    *
    FROM (
    SELECT
    *,
    ROW_NUMBER() OVER (PARTITION BY cst_id ORDER BY cst_create_date DESC) AS flag_last
    FROM bronze.crm_cust_info
    )t WHERE flag_last = 1
```

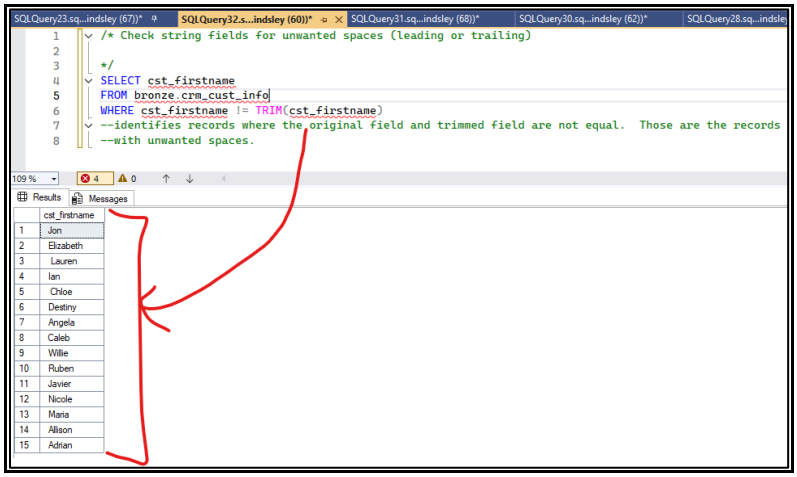
	cst_id	cst_key	cst_firstname	cst_lastname	cst_marital_status	cst_gndr	cst_create_date	flag_last
1	NULL	PO25	NULL	NULL	NULL	NULL	NULL	1
2	11000	AW00011000	Jon	Yang	M	M	2025-10-06	1
3	11001	AW00011001	Eugene	Huang	S	M	2025-10-06	1
4	11002	AW00011002	Ruben	Torres	M	M	2025-10-06	1
5	11003	AW00011003	Christy	Zhu	S	F	2025-10-06	1
6	11004	AW00011004	Elizabeth	Johnson	S	F	2025-10-06	1
7	11005	AW00011005	Julio	Ruiz	S	M	2025-10-06	1
8	11006	AW00011006	Janet	Alvarez	S	F	2025-10-06	1
9	11007	AW00011007	Marco	Mehta	M	M	2025-10-06	1
10	11008	AW00011008	Rob	Verhoff	S	F	2025-10-06	1
11	11009	AW00011009	Shannon	Carlson	S	M	2025-10-06	1
12	11010	AW00011010	Jacquelyn	Suarez	S	F	2025-10-06	1
13	11011	AW00011011	Curtis	Lu	M	M	2025-10-06	1
14	11012	AW00011012	Lauren	Walker	M	F	2025-10-06	1

Checking for Unwanted Spaces in String Values

Each field with strings as data types (NVARCHAR) needs to be checked for unwanted spaces and have unwanted spaces removed (leading or trailing spaces). The following columns were identified as being strings and needed to be checked and corrected if applicable.

cst_key NVARCHAR(50),
cst_firstname NVARCHAR(50),
cst_lastname NVARCHAR(50),
cst_marital_status NVARCHAR(50),
cst_gndr NVARCHAR(50),

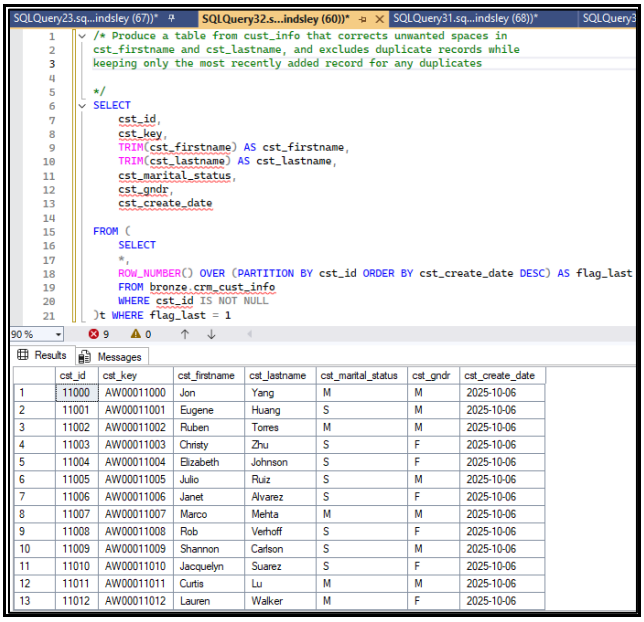
The following query was written and executed on the cst_firstname field to test the identification of records with unwanted spaces as depicted below. The same was done for each string value field. Only the 'cst_firstname' and 'cst_lastname' fields had unwanted spaces that required correction.



```
/* Check string fields for unwanted spaces (leading or trailing)

*/
SELECT cst_firstname
FROM bronze.crm_cust_info
WHERE cst_firstname != TRIM(cst_firstname)
--identifies records where the original field and trimmed field are not equal.  Those are the records
--with unwanted spaces.
```

	cst_firstname
1	Jon
2	Elizabeth
3	Lauren
4	Ian
5	Chloe
6	Destiny
7	Angela
8	Caleb
9	Willie
10	Ruben
11	Javier
12	Nicole
13	Maria
14	Allison
15	Adrian

The following query was written and executed to produce a table that included all fields from cust_info table, but corrects the wanted spaces in the cst_firstname and cst_last name fields, and includes only unique records records (non-duplicates) with a ranking of "1" based on their creation date. The ranking identifier is not included in the output.

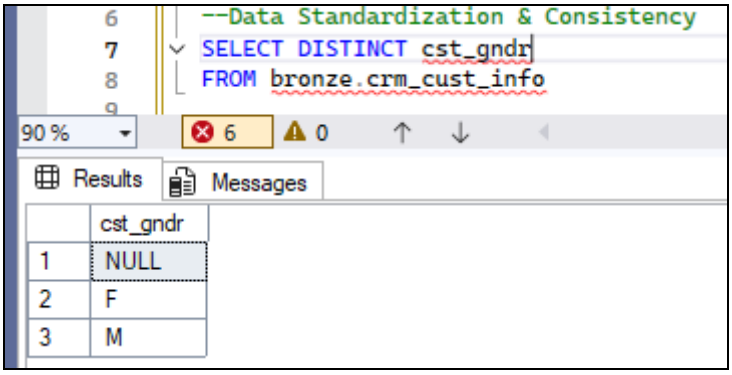


```
/* Produce a table from cust_info that corrects unwanted spaces in
cst_firstname and cst_lastname, and excludes duplicate records while
keeping only the most recently added record for any duplicates

*/
SELECT
    cst_id,
    cst_key,
    TRIM(cst_firstname) AS cst_firstname,
    TRIM(cst_lastname) AS cst_lastname,
    cst_marital_status,
    cst_gndr,
    cst_create_date

FROM (
    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 IS NOT NULL
)t WHERE flag_last = 1
```

	cst_id	cst_key	cst_firstname	cst_lastname	cst_marital_status	cst_gndr	cst_create_date
1	11000	AW00011000	Jon	Yang	M	M	2025-10-06
2	11001	AW00011001	Eugene	Huang	S	M	2025-10-06
3	11002	AW00011002	Ruben	Torres	M	M	2025-10-06
4	11003	AW00011003	Christy	Zhu	S	F	2025-10-06
5	11004	AW00011004	Elizabeth	Johnson	S	F	2025-10-06
6	11005	AW00011005	Julio	Ruiz	S	M	2025-10-06
7	11006	AW00011006	Janet	Alvarez	S	F	2025-10-06
8	11007	AW00011007	Marco	Mehta	M	M	2025-10-06
9	11008	AW00011008	Rob	Verhoff	S	F	2025-10-06
10	11009	AW00011009	Shannon	Carlson	S	M	2025-10-06
11	11010	AW00011010	Jacquelyn	Suarez	S	F	2025-10-06
12	11011	AW00011011	Curtis	Lu	M	M	2025-10-06
13	11012	AW00011012	Lauren	Walker	M	F	2025-10-06

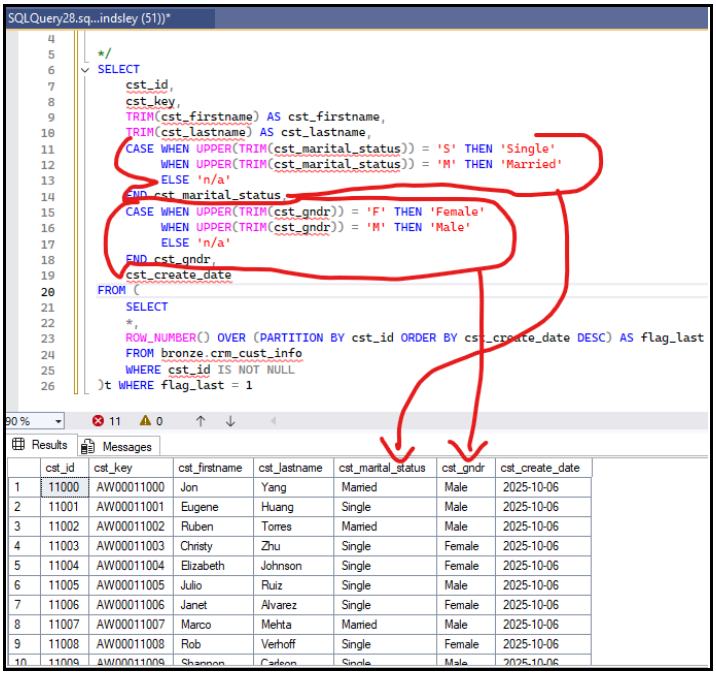
Checking for consistency of values in low cardinality columns (columns used for categorical data)
The columns capturing customer marital status and gender have low quantity of possible variations. The data in these columns needs to be assessed for consistency. Example: The cst_gndr column should only have two possible values → "M" for male and "F" for female. The values in this column should be "M" or "F" only (or maybe NULL). It should be list "Male", "Female", "Q", "Duck", or anything else. So I run the following query to identify inconsistencies.

The following query was run and produced results for three (3) possible values: M, F, and NULL as depicted below.



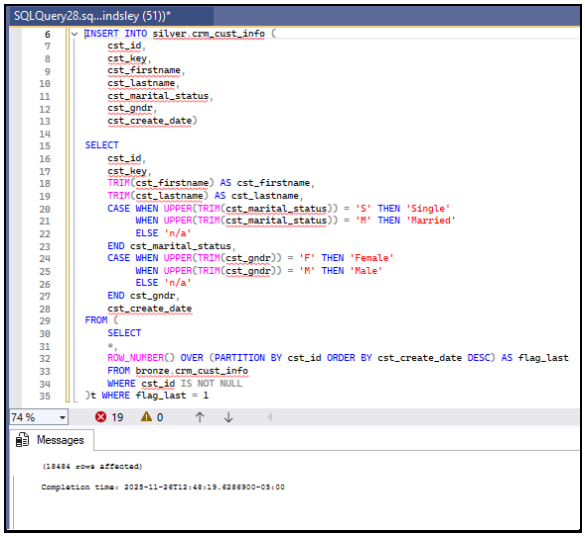
Now…to make this data business / reader friendly, we will adjust our primary transformation query for this table so that M is replaced with "Male", "F" is replaced with "Female" and "NULL" is replaced with "n/a". The query will also be written so that any unwanted spaces are trimmed and we account for any possible lower case occurrences of "M" or "F".

The same approach was applied to the marital status field (cst_marital_status) to replace "S", "M", and "NULL" with "Single", "Married", and "n/a" respectively.

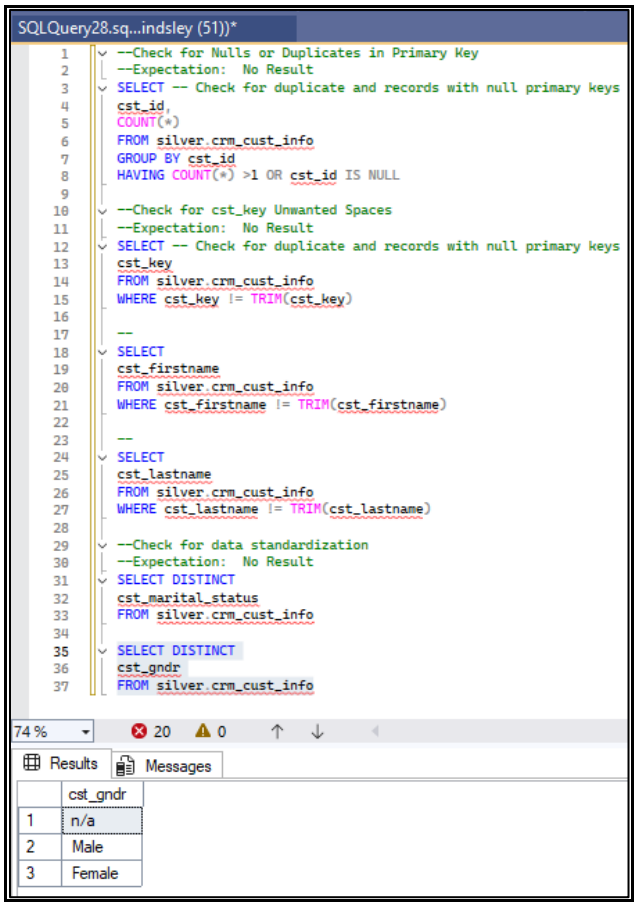


Inserting Customer Info data into Silver layer table

At this point, I am now ready to insert the cleansed data into the Silver layer table as the data quality checks and cleansing complete for the table crm_cust_info. An 'INSERT INTO" command was added to the data quality script so it would execute the load after the data cleansing script is complete.



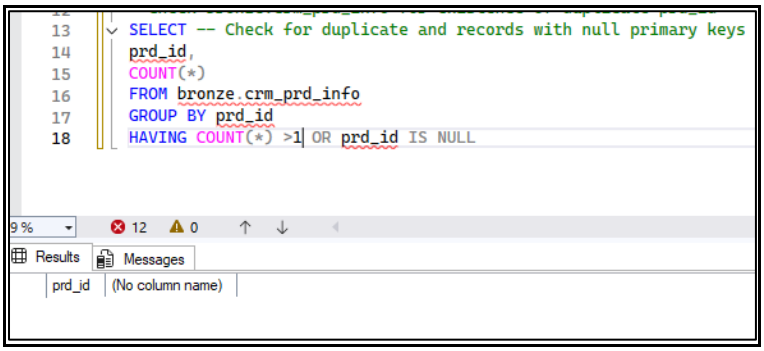
I now check that the data quality corrections and standardizations are correct by running a query for each correction / standardizationWith the load complete. The screenshot below depicts the results of the query to confirm values listed for cst_gndr are correctly converted from M, F, and NULL to "Male", "Female, and "n/a"....and that no other value types exist



CRM → Product Info Table

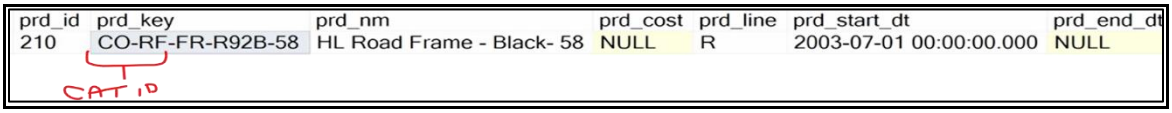
Checking for and Correcting Records with Duplicate and Null Primary Keys

Primary key was evaluated to determine whether or not there were NULL values or duplicates using the following script. This resulted in zero (0) duplicates or null values for the prd_id field



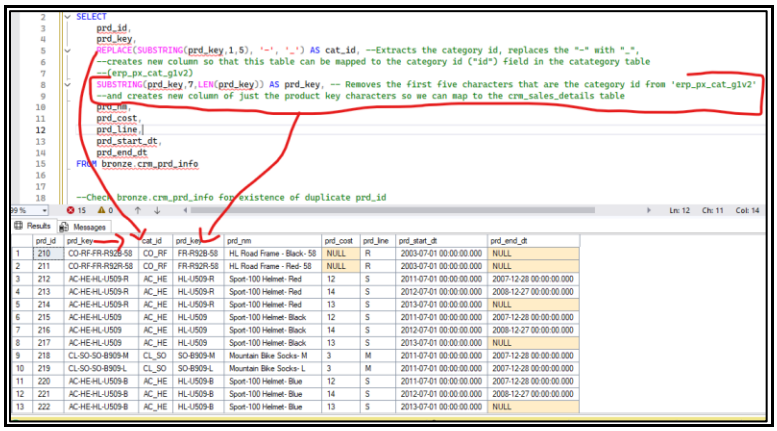
Deriving New Columns to Enable Mapping

During the data exploration, I found that the prd_key field is a concatenation of the category id 'id' field from the 'erp_px_cat_g1v2' table and additional characters native to this table. The first five characters of each value in this field are the category id and the remaining are just…..a bunch of other characters. So, this column needs to separate the first five characters into a new column so the 'erp_px_cat_g1v2' can be mapped to it.



Additionally, I have to correct for the difference between the characters as the "id" in the 'erp_px_cat_g1v2' are separated by a "_", whereas the characters in the prd_key from the prd_info table are separated by a "-".

I also need to create a separate column for the prd_key without the category id characters. So…some standardization is needed here. This is accomplished by the script (with results below) in the following image:



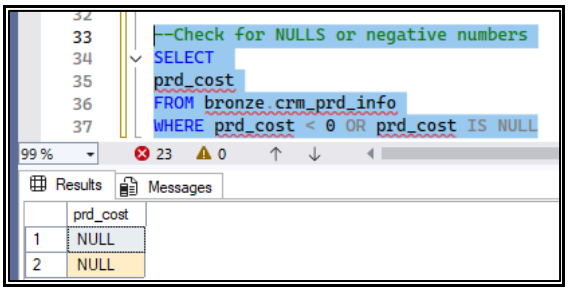
Checking for unwanted spaces

I checked the prd_nm field for unwanted spaces and found none using the following query

```
SELECT
prd_nm
FROM bronze.crm_prd_info
WHERE prd_nm != TRIM(prd_nm)
```

Checking for product costs <0 and Null values

I checked the prd_cost field to identify any products with negative values or nulls using the following query. No products had negative costs, but there are two products with null values for cost. In a business context, the business stakeholder would need to determine whether to retain the null or replace it with a "0" or some other value.

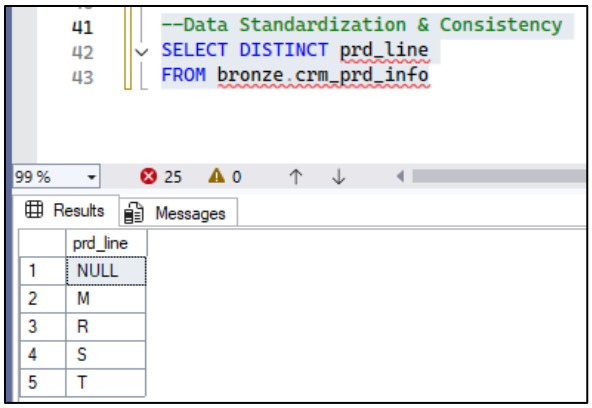


As this is a fictional situation without actual stakeholders, I chose to replace the NULL values with a "0". I did this by adding the following to the transformation script for the prd_info table:

```
ISNULL(prd_cost,0) AS prd_cost, --Replaces null values in cost with 0
```

Checking 'prd_line' field for consistency of values (similar to the check conducted for marital status and gender in the 'crm_cust_info' table

The 'prd_line' column has a low quantity of possible variations. The data in this column needs to be assessed for consistency. I ran 'SELECT DISTINCT' query on the prd_info table and found the following possible values to be: "NULL", "M", "R", "S", and "T".



In a real-world situation, I would ask the data owner / business stakeholder what those values represent so that I could convert them to a user / business friendly value. There are no stakeholders in this case as this is a fictional project. So…I added a CASE WHEN with UPPER(TRIM) to the transformation script to replace the values with the following per the instructor for this course:

```
CASE WHEN UPPER(TRIM(prd_line)) = 'M' THEN 'Mountain'
        WHEN UPPER(TRIM(prd_line)) = 'R' THEN 'Road'
        WHEN UPPER(TRIM(prd_line)) = 'S' THEN 'Other Sales'
        WHEN UPPER(TRIM(prd_line)) = 'T' THEN 'Touring'
        ELSE 'n/a'
END AS prd_line,
```

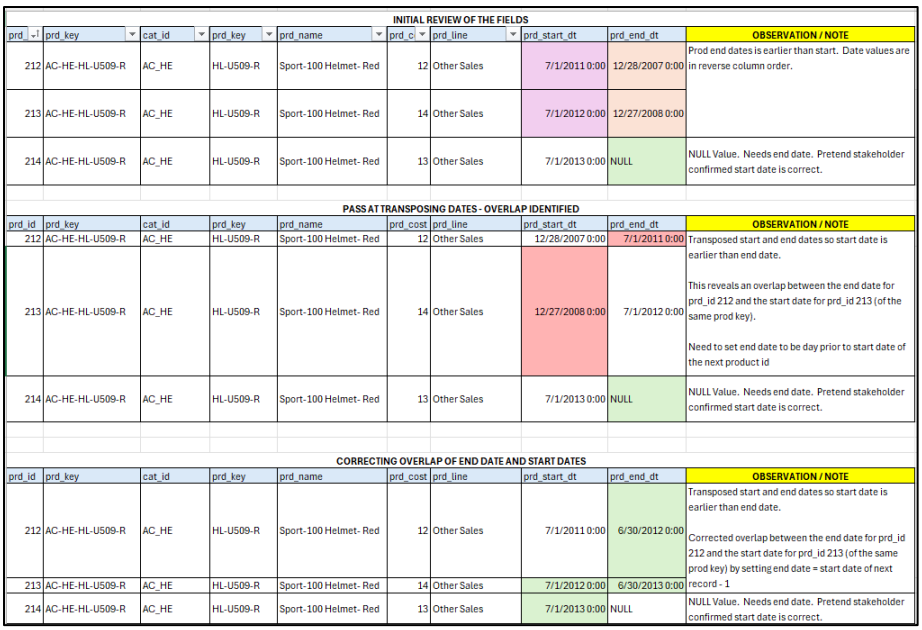
Checking the quality of Product Start and End Date fields and correcting quality issues.
The first part of this effort is to identify records that have a product end date that is earlier than the product start date. To do this, I ran the following query and found 201 records in which this occurs.

```
SELECT * FROM bronze.crm_prd_info
WHERE prd_end_dt < prd_start_dt
```

I assumed that the start and end dates for these records were simply reversed and in the incorrect field. So, the start date is actually the end date and the end date is actually the start date. Assuming this is the case, I also found that several records had product end dates that overlap with the start date of the product with the same product key but next prd_id sequence. So….I had some corrections to make.

In a real-world scenario, these observations would need to be reviewed and verified with the data owner before making any decisions on transformations / clean-up. In this case, I pretend that I reviewed the issues with a data owner / business stakeholder and had them confirm the dates were reversed, and that the end dates should not overlap but end the day prior to the start date of the next product id having the same value for 'prd_key'

The below image summarizes observations against a sample set of records with approach for correction:



INITIAL REVIEW OF THE FIELDS

prd_	prd_key	cat_id	prd_key	prd_name	prd_c	prd_line	prd_start_dt	prd_end_dt	OBSERVATION / NOTE
212	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	12	Other Sales	7/1/2011 0:00	12/28/2007 0:00	Prod end dates is earlier than start. Date values are in reverse column order.
213	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	14	Other Sales	7/1/2012 0:00	12/27/2008 0:00	
214	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	13	Other Sales	7/1/2013 0:00	NULL	NULL Value. Needs end date. Pretend stakeholder confirmed start date is correct.

PASS AT TRANSPOSING DATES - OVERLAP IDENTIFIED

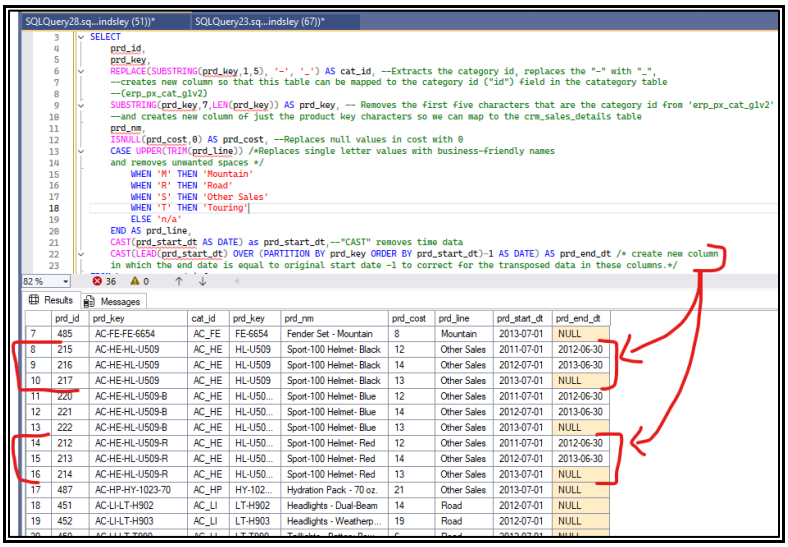
prd_id	prd_key	cat_id	prd_key	prd_name	prd_cost	prd_line	prd_start_dt	prd_end_dt	OBSERVATION / NOTE
212	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	12	Other Sales	12/28/2007 0:00	7/1/2011 0:00	Transposed start and end dates so start date is earlier than end date.
213	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	14	Other Sales	12/27/2008 0:00	7/1/2012 0:00	This reveals an overlap between the end date for prd_id 212 and the start date for prd_id 213 (of the same prod key). Need to set end date to be day prior to start date of the next product id
214	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	13	Other Sales	7/1/2013 0:00	NULL	NULL Value. Needs end date. Pretend stakeholder confirmed start date is correct.

CORRECTING OVERLAP OF END DATE AND START DATES

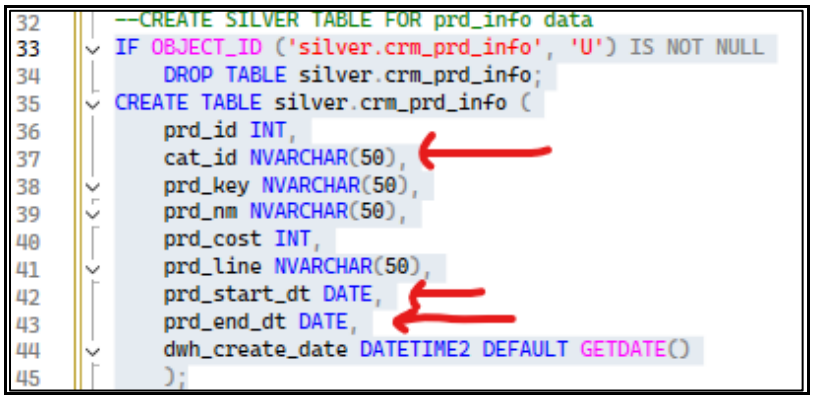
prd_id	prd_key	cat_id	prd_key	prd_name	prd_cost	prd_line	prd_start_dt	prd_end_dt	OBSERVATION / NOTE
212	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	12	Other Sales	7/1/2011 0:00	6/30/2012 0:00	Transposed start and end dates so start date is earlier than end date. Corrected overlap between the end date for prd_id 212 and the start date for prd_id 213 (of the same prod key) by setting end date = start date of next record - 1
213	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	14	Other Sales	7/1/2012 0:00	6/30/2013 0:00	
214	AC-HE-HL-U509-R	AC_HE	HL-U509-R	Sport-100 Helmet- Red	13	Other Sales	7/1/2013 0:00	NULL	NULL Value. Needs end date. Pretend stakeholder confirmed start date is correct.

The following commands were added to the script to correct the start and end dates, and to remove the time values from the date field as they were “00:00:00”. Below is a screenshot of the full script and results.

CAST(prd_start_dt AS DATE) as prd_start_dt,--"CAST" removes time data
CAST(LEAD(prd_start_dt) OVER (PARTITION BY prd_key ORDER BY prd_start_dt)-1 AS DATE)
AS prd_end_dt



I needed to update the ddl for the prd_info table in the Silver layer to account for the removal of time values from the start and and date columns in the transformation script. I also added the cat_id field to the ddl as that is being created in the transformation script. Below is the updated ddl to account for this.



Inserting and validating the transformation query results into the Silver layer prd_info table

It is now time to insert the transformed data now that the data quality corrections are complete for the prd_info table and the ddl has been updated.

To insert the corrected data, I wrote and executed an INSERT INTO script ahead of the transformation query for the prd_info table and executed checks against the output. Below is the INSERT INTO with the transformation script:

```
INSERT INTO silver.crm_prd_info (
    prd_id,
    cat_id,
    prd_key,
    prd_nm,
    prd_cost,
    prd_line,
    prd_start_dt,
    prd_end_dt
)

SELECT
    prd_id,
    REPLACE(SUBSTRING(prd_key,1,5), '-', '_') AS cat_id, --Extracts the category id, replaces the "-" with "_",
    --creates new column so that this table can be mapped to the category id ("id") field in the catategory table
    --(erp_px_cat_g1v2)
    SUBSTRING(prd_key,7,LEN(prd_key)) AS prd_key, -- Removes the first five characters that are the category id from 'erp_px_cat_g1v2'
    --and creates new column of just the product key characters so we can map to the crm_sales_details table
    prd_nm,
    ISNULL(prd_cost,0) AS prd_cost, --Replaces null values in cost with 0
    CASE UPPER(TRIM(prd_line)) /*Replaces single letter values with business-friendly names
    and removes unwanted spaces */
        WHEN 'M' THEN 'Mountain'
        WHEN 'R' THEN 'Road'
        WHEN 'S' THEN 'Other Sales'
        WHEN 'T' THEN 'Touring'
        ELSE 'n/a'
    END AS prd_line,
    CAST(prd_start_dt AS DATE) as prd_start_dt,--"CAST" removes time data
    CAST(LEAD(prd_start_dt) OVER (PARTITION BY prd_key ORDER BY prd_start_dt)-1 AS DATE) AS prd_end_dt /* create new column
    in which the end date is equal to original start date -1 to correct for the transposed data in these columns.*/
FROM bronze.crm_prd_info
```

The next step was to run checks against the transformed data and derived fields in the prd_info table in the silver table. Each of the following scripts were run and executed. All were successful.

```
--Check silver.crm_prd_info for existence of duplicate prd_id
SELECT -- Check for duplicate and records with null primary keys
prd_id,
COUNT(*)
FROM silver.crm_prd_info
GROUP BY prd_id
HAVING COUNT(*) >1 OR prd_id IS NULL

--Check for cst_key Unwanted Spaces
--Expectation:  No Result
SELECT
prd_nm
FROM silver.crm_prd_info
WHERE prd_nm != TRIM(prd_nm)

--Check for NULLS or negative numbers
--Expectation:  No Result
SELECT
prd_cost
FROM silver.crm_prd_info
WHERE prd_cost < 0 OR prd_cost IS NULL

--Data Standardization & Consistency
SELECT DISTINCT prd_line
FROM silver.crm_prd_info

--Checking for invalid product start and end dates
SELECT
*
FROM silver.crm_prd_info
WHERE prd_end_dt < prd_start_dt --Checks for records with end dates that are earlier that the prd start date

SELECT
*
FROM silver.crm_prd_info --View full data set in the silver table to get a warm fuzzy
```

74 % | 45 | 0

Results | Messages

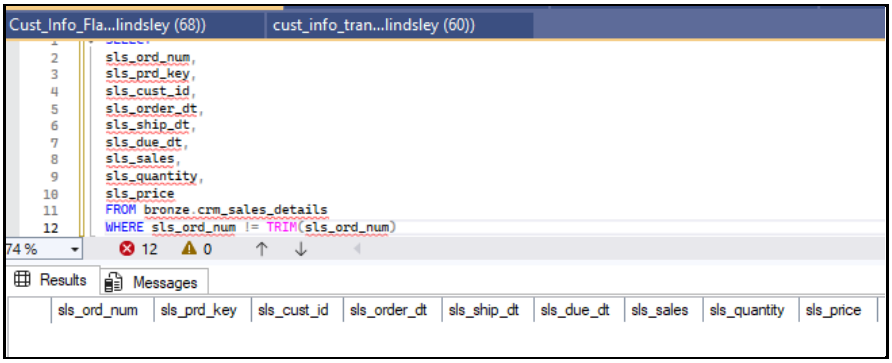
	prd_id	cat_id	prd_key	prd_nm	prd_cost	prd_line	prd_start_dt	prd_end_dt	dwh_create_date
1	478	AC_BC	BC-M005	Mountain Bottle Cage	4	Mountain	2013-07-01	NULL	2025-11-27 09:56:50.0500000
2	479	AC_BC	BC-R205	Road Bottle Cage	3	Road	2013-07-01	NULL	2025-11-27 09:56:50.0500000
3	477	AC_BC	WB-H098	Water Bottle - 30 oz.	2	Other Sales	2013-07-01	NULL	2025-11-27 09:56:50.0500000
4	483	AC_BR	RA-H123	Hitch Rack - 4-Bike	45	Other Sales	2013-07-01	NULL	2025-11-27 09:56:50.0500000
5	486	AC_BS	ST-1401	All-Purpose Bike Stand	59	Mountain	2013-07-01	NULL	2025-11-27 09:56:50.0500000
6	484	AC_CL	CL-9009	Bike Wash - Dissolver	3	Other Sales	2013-07-01	NULL	2025-11-27 09:56:50.0500000

DISCLAIMER: Now….I completely understand that this is best case scenario as I was following step-by-step with the instructor. I likely would have dorked this up completely and not known how to correct any issues if running this on my own. It would have been a disaster in all likelihood. But I'm learning….baby steps. This is fun…..so far.

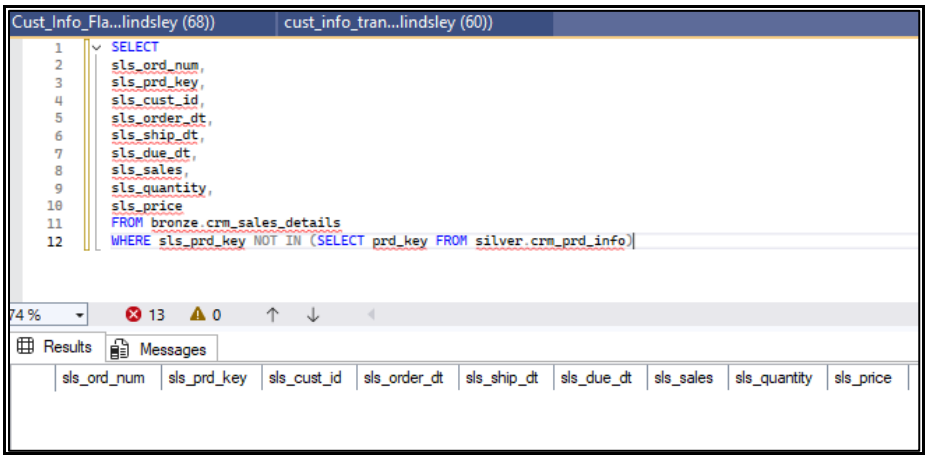
CRM → Sales Details Table

Checking primary key (sls_ord_num) for unwanted spaces:

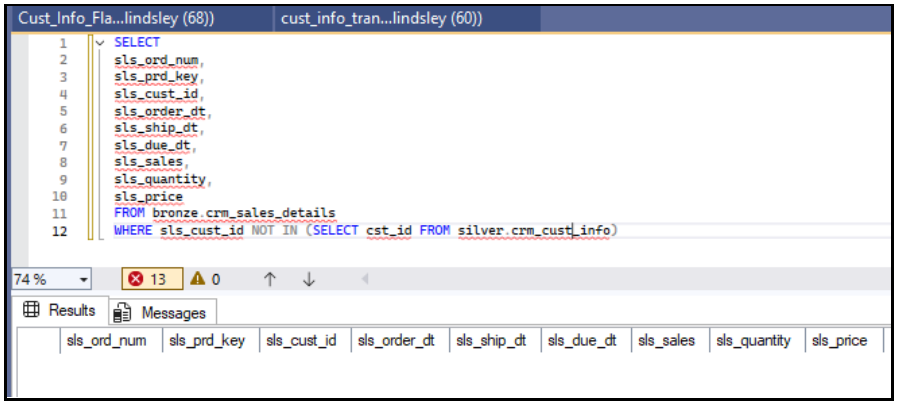
Executed the following query with no results for records with unwanted spaces:



Checking the sls_prd_key to determine whether or not there are matches in the prd_key of the the crm_prd_info table: This is necessary as I determined earlier in the data diagram that I would need to using a portion of the values in this column to match to the prd_key field in the crm_prd_info table. The following was written and executed…resulting in zero (0) matches.



Checking the sls_cust_id to determine whether or not there are matches in the cst_id field of the crm_cust_info table: This is necessary as I determined earlier in the data diagram that I would need to using a portion of the values in this column to match to the cst_id field in the crm_cust_info table. The following was written and executed…resulting in zero (0) matches.



Check values in the following fields to identify non-date values: sls_order_dt; sls_ship_dt; and sls_due_dt. I ran the following query and it returned 17 records with a “0” in the sls_order_dt field

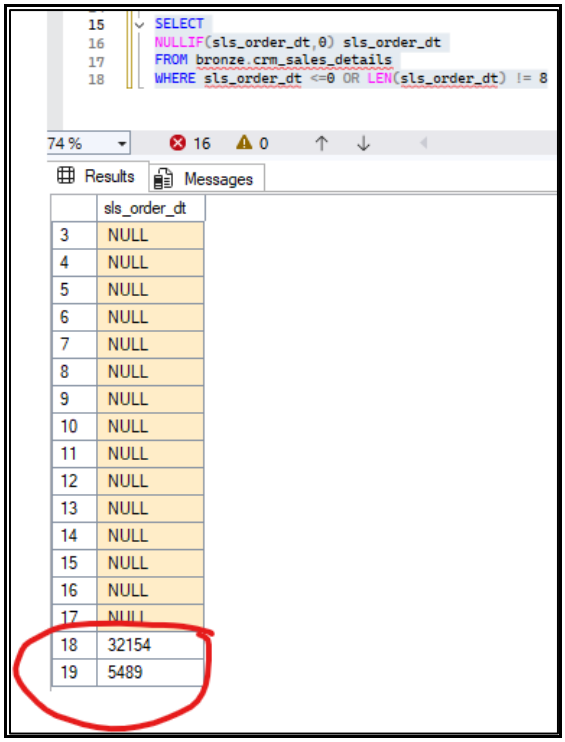
```
SELECT
sls_order_dt
FROM bronze.crm_sales_details
WHERE sls_order_dt <=0
```

I needed to convert the “0” for these records to NULL. The following was written to correct for this:

```
SELECT
NULLIF(sls_order_dt,0) sls_order_dt
FROM bronze.crm_sales_details
WHERE sls_order_dt <=0
```

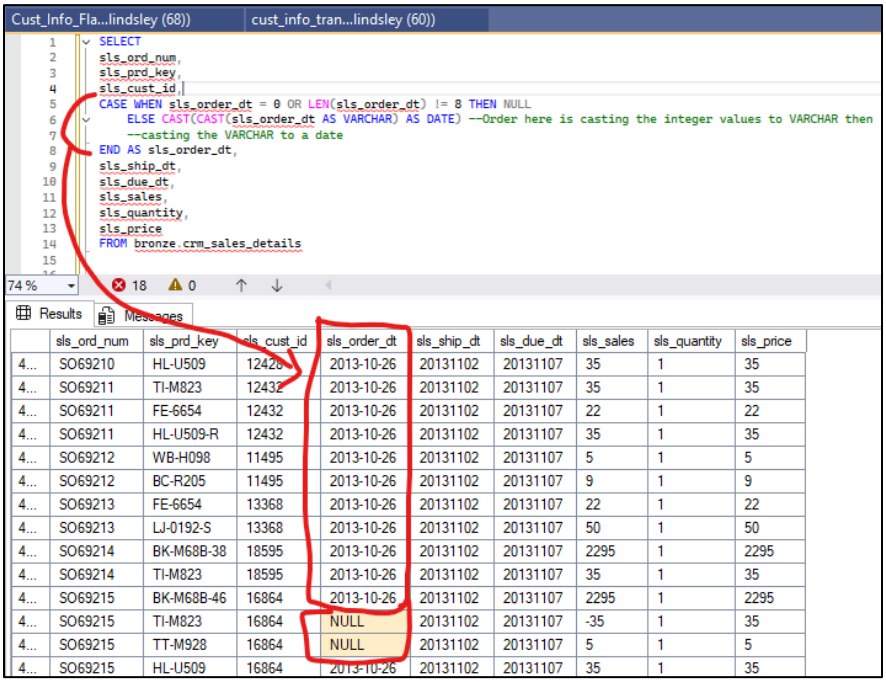
A further check of the sls_order_dt field was done to identify:
- Any additional non-date numerical values that can not be converted to a date
- Any dates beyond 20500101 (fictional upper boundary chosen for this exercise)
- Any dates earlier than 19990101 (fictional lower boundary chosen for this exercise).

This query was run and returned all NULLs as well as two records with non-date numerical values.



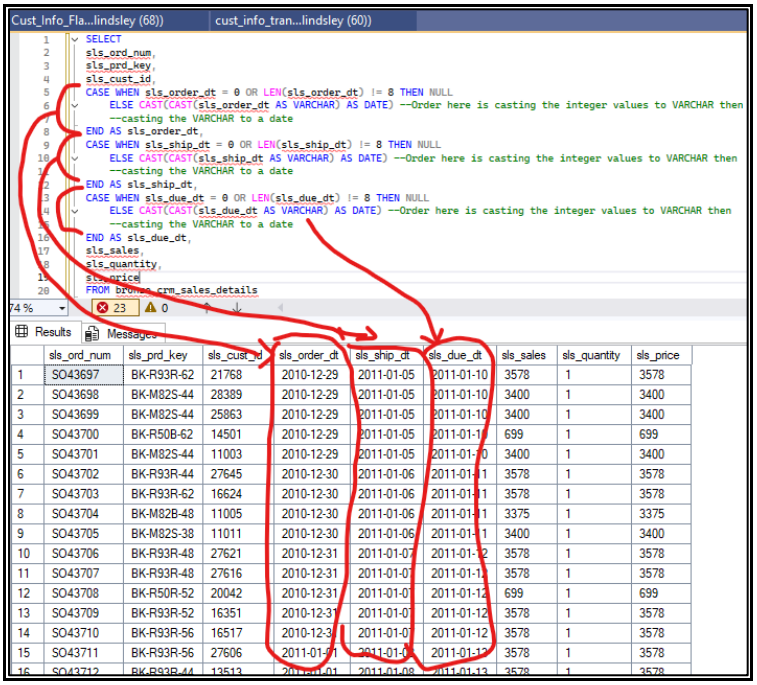
Converting the values in the following fields from INT to dates: sls_order_dt; sls_ship_dt; and sls_due_dt. This is necessary because the values should be dates but are written as “20101229” for example. They should be in this format “2010-12-29”

I added the following to the transformation query to convert non-date numerical values to NULL and set all others to dates:



	sls_ord_num	sls_prd_key	sls_cust_id	sls_order_dt	sls_ship_dt	sls_due_dt	sls_sales	sls_quantity	sls_price
4...	SO69210	HL-U509	12426	2013-10-26	20131102	20131107	35	1	35
4...	SO69211	TI-M823	12432	2013-10-26	20131102	20131107	35	1	35
4...	SO69211	FE-6654	12432	2013-10-26	20131102	20131107	22	1	22
4...	SO69211	HL-U509-R	12432	2013-10-26	20131102	20131107	35	1	35
4...	SO69212	WB-H098	11495	2013-10-26	20131102	20131107	5	1	5
4...	SO69212	BC-R205	11495	2013-10-26	20131102	20131107	9	1	9
4...	SO69213	FE-6654	13368	2013-10-26	20131102	20131107	22	1	22
4...	SO69213	LJ-0192-S	13368	2013-10-26	20131102	20131107	50	1	50
4...	SO69214	BK-M68B-38	18595	2013-10-26	20131102	20131107	2295	1	2295
4...	SO69214	TI-M823	18595	2013-10-26	20131102	20131107	35	1	35
4...	SO69215	BK-M68B-46	16864	2013-10-26	20131102	20131107	2295	1	2295
4...	SO69215	TI-M823	16864	NULL	20131102	20131107	-35	1	35
4...	SO69215	TT-M928	16864	NULL	20131102	20131107	5	1	5
4...	SO69215	HL-U509	16864	2013-10-26	20131102	20131107	35	1	35

The same series of checks and corrections done on the sls_order_dt field were conducted for the remaining date fields and added to the transformation query. No quality issues were identified, but I added the same CASE WHEN transformation script to each date field to ensure similar potential quality issues that might arise in future data sets are automatically corrected.



	sls_ord_num	sls_prd_key	sls_cust_id	sls_order_dt	sls_ship_dt	sls_due_dt	sls_sales	sls_quantity	sls_price
1	SO43697	BK-R93R-62	21768	2010-12-29	2011-01-05	2011-01-10	3578	1	3578
2	SO43698	BK-M82S-44	28389	2010-12-29	2011-01-05	2011-01-10	3400	1	3400
3	SO43699	BK-M82S-44	25863	2010-12-29	2011-01-05	2011-01-10	3400	1	3400
4	SO43700	BK-R50B-62	14501	2010-12-29	2011-01-05	2011-01-10	699	1	699
5	SO43701	BK-M82S-44	11003	2010-12-29	2011-01-05	2011-01-10	3400	1	3400
6	SO43702	BK-R93R-44	27645	2010-12-30	2011-01-06	2011-01-11	3578	1	3578
7	SO43703	BK-R93R-62	16624	2010-12-30	2011-01-06	2011-01-11	3578	1	3578
8	SO43704	BK-M82B-48	11005	2010-12-30	2011-01-06	2011-01-11	3375	1	3375
9	SO43705	BK-M82S-38	11011	2010-12-30	2011-01-06	2011-01-11	3400	1	3400
10	SO43706	BK-R93R-48	27621	2010-12-31	2011-01-07	2011-01-12	3578	1	3578
11	SO43707	BK-R93R-48	27616	2010-12-31	2011-01-07	2011-01-12	3578	1	3578
12	SO43708	BK-R50R-52	20042	2010-12-31	2011-01-07	2011-01-12	699	1	699
13	SO43709	BK-R93R-52	16351	2010-12-31	2011-01-07	2011-01-12	3578	1	3578
14	SO43710	BK-R93R-56	16517	2010-12-31	2011-01-07	2011-01-12	3578	1	3578
15	SO43711	BK-R93R-56	27606	2011-01-01	2011-01-08	2011-01-13	3578	1	3578
16	SO43712	BK-R93R-44	13513	2011-01-01	2011-01-08	2011-01-13	3578	1	3578

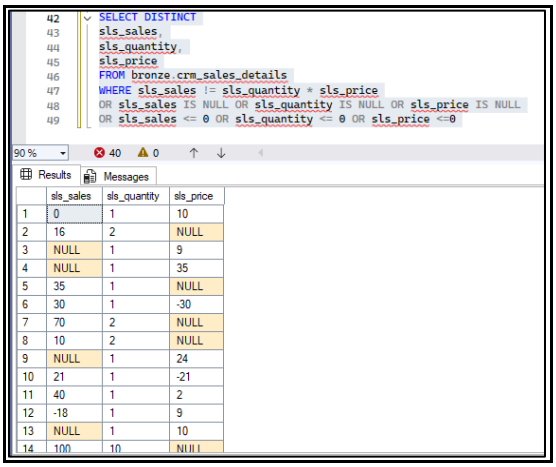
Checking for invalid dates. The purpose of this check is to ensure that the ship dates and due dates are not earlier than the order dates. The following query was run and returned no results - confirming that order dates are not later than ship or due dates.

```
SELECT
*
FROM bronze.crm_sales_details
WHERE sls_order_dt > sls_ship_dt OR sls_order_dt > sls_due_dt
```

Checking quality of data for sales, quantity and price. In this fictional project, the following business rules have been defined for these fields:

- Sales must be equal to the product of quantity and price (sales = quantity * price).
- Negative values, zeros, and Nulls are not allowed

To check for data consistency, I ran the following query that returned results with records in all three fields that required correction.



```
SELECT DISTINCT
sls_sales,
sls_quantity,
sls_price
FROM bronze.crm_sales_details
WHERE sls_sales != sls_quantity * sls_price
OR sls_sales IS NULL OR sls_quantity IS NULL OR sls_price IS NULL
OR sls_sales <= 0 OR sls_quantity <= 0 OR sls_price <=0
```

90 % | 40 | 0

Results | Messages

	sls_sales	sls_quantity	sls_price
1	0	1	10
2	16	2	NULL
3	NULL	1	9
4	NULL	1	35
5	35	1	NULL
6	30	1	-30
7	70	2	NULL
8	10	2	NULL
9	NULL	1	24
10	21	1	-21
11	40	1	2
12	-18	1	9
13	NULL	1	10
14	100	10	NULL

In a real-world situation, I would need to meet with the data owners / business stakeholders to review the data quality issues and gain alignment on a decision to resolve them. The decision could be for them to fix the data in the source system or to correct them through transformations in the data warehouse until they are able to correct them in the source system.

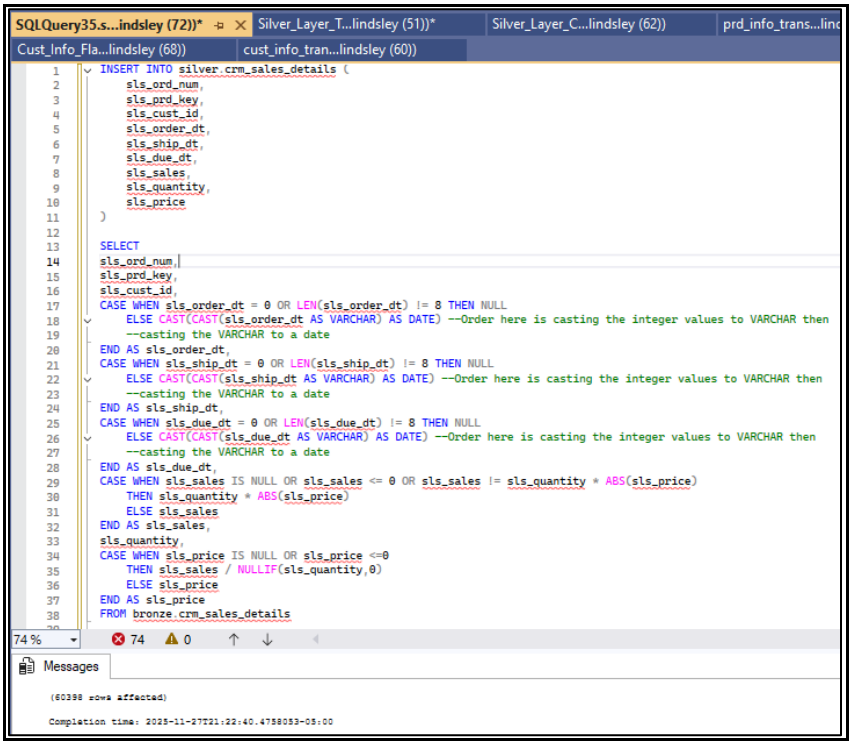
As this is a fictional situation, the following rules were applied to correct the data quality issues identified for sales, price, and quantity.

- If Sales is negative, zero, or null, derive it using Quantity * Price
- If Price is zero or Null, calculate it using sales / quantity
- If Price is negative, convert to a positive number

The following corrections were added to the transformation query using the above rules.

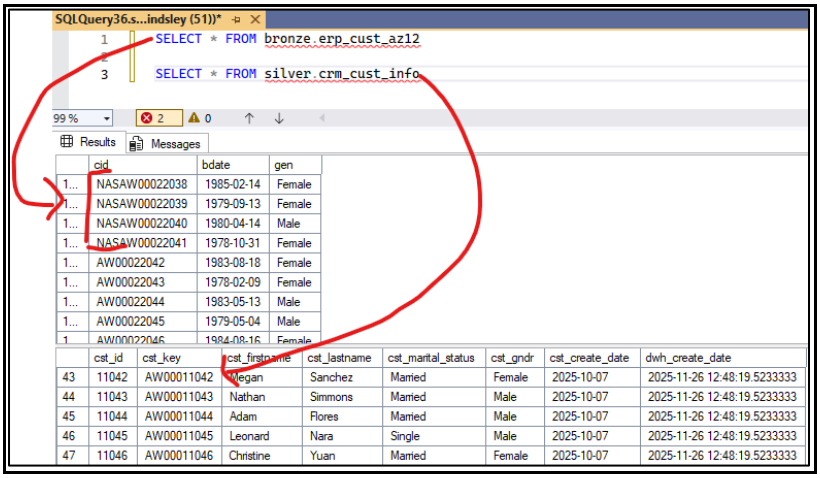
```
sls_price AS old_sls_price,
CASE WHEN sls_sales IS NULL OR sls_sales <= 0 OR sls_sales != sls_quantity * ABS(sls_price)
        THEN sls_quantity * ABS(sls_price)
        ELSE sls_sales
END AS sls_sales,
sls_quantity,
CASE WHEN sls_price IS NULL OR sls_price <=0
        THEN sls_sales / NULLIF(sls_quantity,0)
        ELSE sls_price
END AS sls_price
```

Finalize transformation query for crm_sales_details, update ddl, and INSERT into the silver table. After adding the above to the transformation query, I also had to update the data types for the date fields in the ddl script for the sales_details table from INT to DATE and re-execute. I Then prepared the INSERT INTO script to insert the corrected data into the silver table

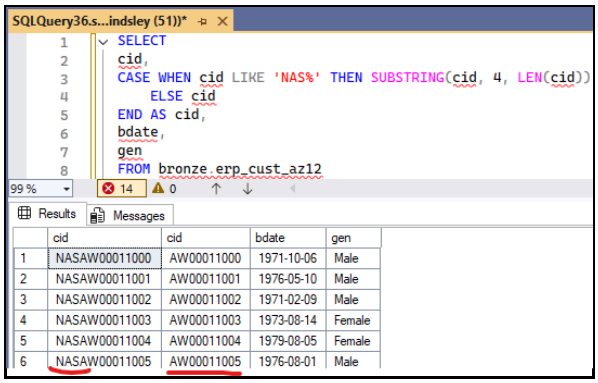


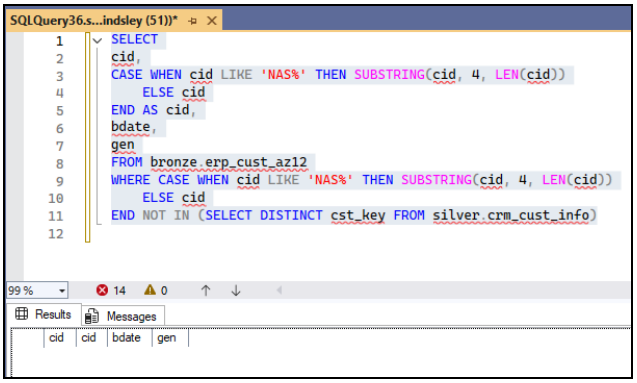
Check quality of data inserted into the the crm_sales_details table post transformation. All of the same quality checks run against the bronze table for crm_sales_details were executed on the silver table for crm_sales_details to ensure that all transformations works, and that there are no additional quality issues requiring correction.

ERP → cust_az12 (The table with customer ids, birthdates and gender)
Checking primary key (cid) to determine what needs to be done to connect to crm_cust_info via the cst_key
Running separate queries against both tables allows me to see that some of the records in the cid field in the cust_az12 have extra characters (“NAS”) that do not appear in the cst_key field. I will need to account for this to connect both tables.



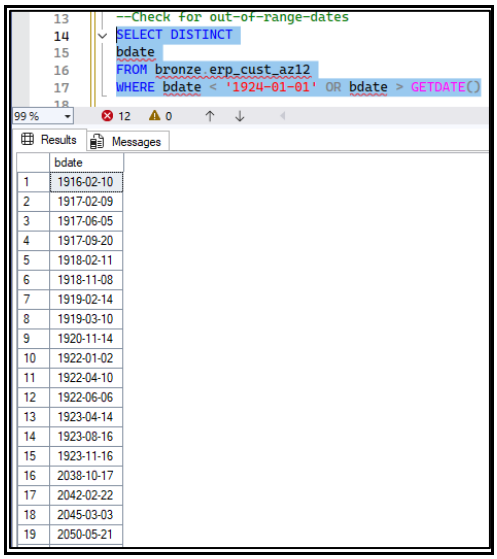
I wrote the following transformation query to remove the “NAS” from any records that included it in the cid field, then added the CASE WHEN to a WHERE statement to check if there were any records from the correction that did not appear in the silver table of crm_cust_info.





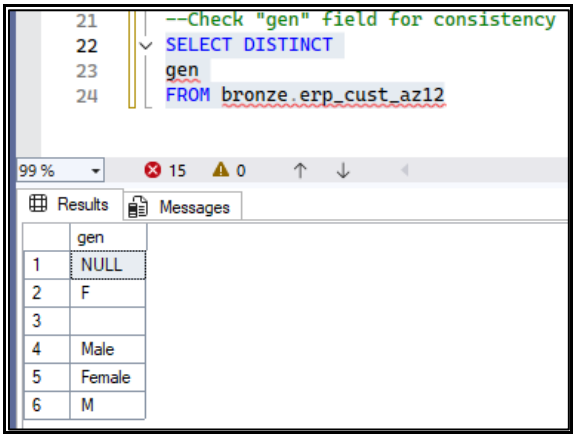
Check birthdates for values outside of upper and lower date boundaries

The following query was run to identify anyone that is > = 100 years old or have birthdates in the future. The results returned records that meet both criteria and would require correction. In a real world situation, I would need to meet with the data owner / business stakeholder to determine how these should be treated. Since this is a fictional situation, I will pretend that I met with the stakeholder and was informed that only birthdates in the future would be adjusted in the data set and they would be replaced with NULL. This was accomplished using a CASE WHEN and GETDATE.

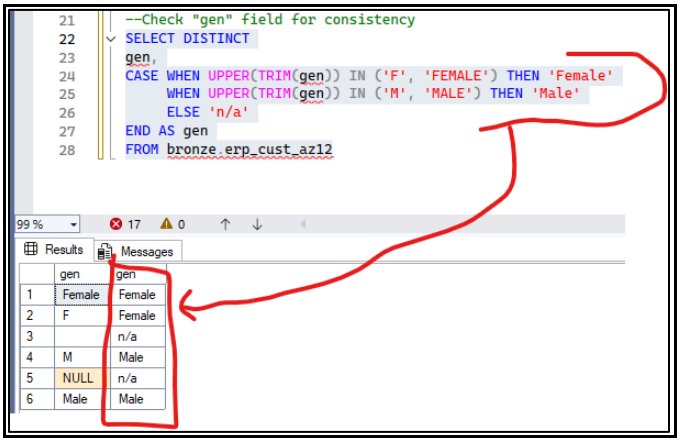


Check gen field for data standardization

This is a categorization field and should have standard values. A quick query on distinct values in the gen field resulted in inconsistent values that will require standardization (“F”, “Female”, “M”, “Male”, etc.). Additionally, there are empty values and nulls. In a real world situation, I would meet with the business stakeholder to obtain a decision on the appropriate business-friendly correction and how to account for / correct the NULLs and empty fields.

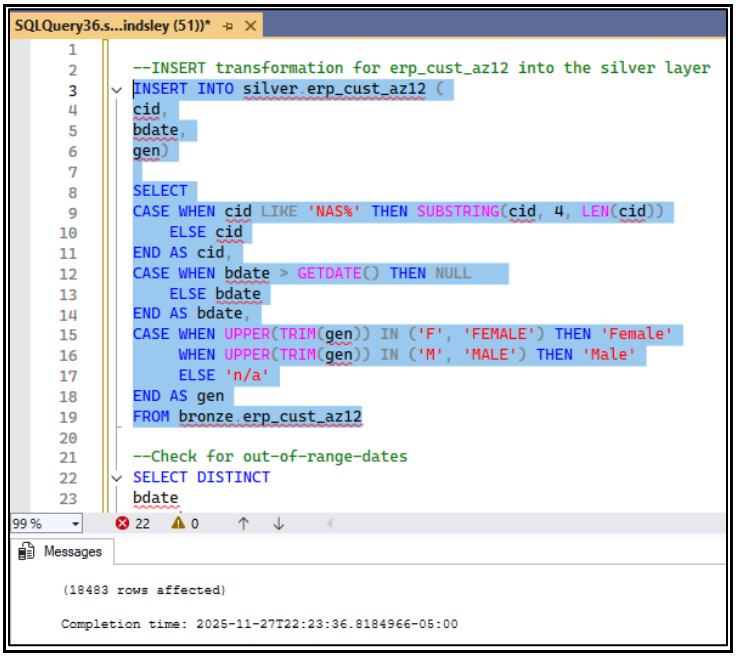


The following query was written to standardize values in the gen field.



Insert erp_cust_az12 into Silver Layer

The corrections identified for erp_cust_az12 were consolidated into a single transformation query along with an INSERT INTO command to insert the corrected data into the silver layer with quality checks completed to ensure data fed correctly

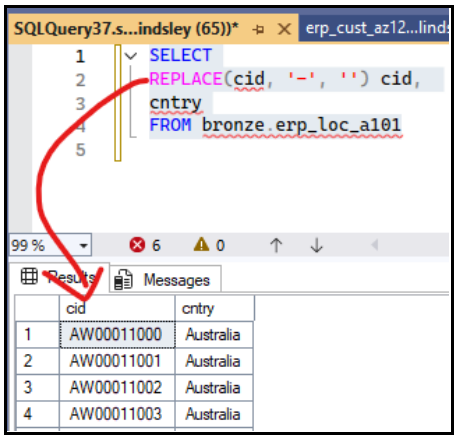


ERP → loca101 (The table with customer id and country)

Checking primary key (cid) to determine what needs to be done to connect to crm_cust_info via the cst_key

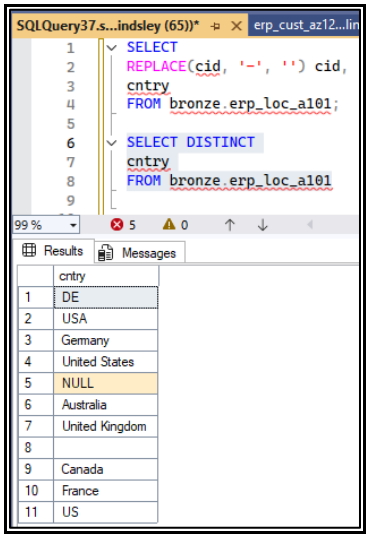
Running separate queries against both tables allows me to see that the cid field in the loc_a101 table has a "-" between the first two string characters and the remaining numerical characters in the field. The "-" will need to be removed to enable mapping between the two tables.

The following script was written to remove the "-" from the cid field in loc_a101. I also ran the same a query with a NOT IN subquery to determine whether any of the transformed values did not have matches in the cust_info_table



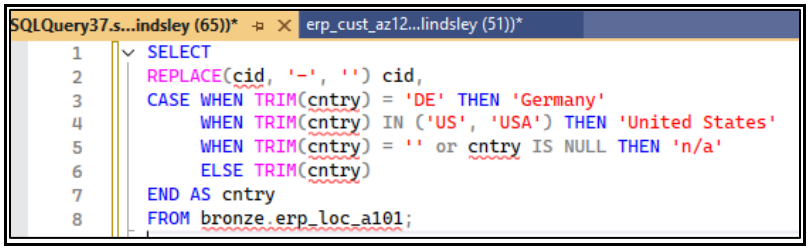
Checking the country field to identify consistency / needs for standardization

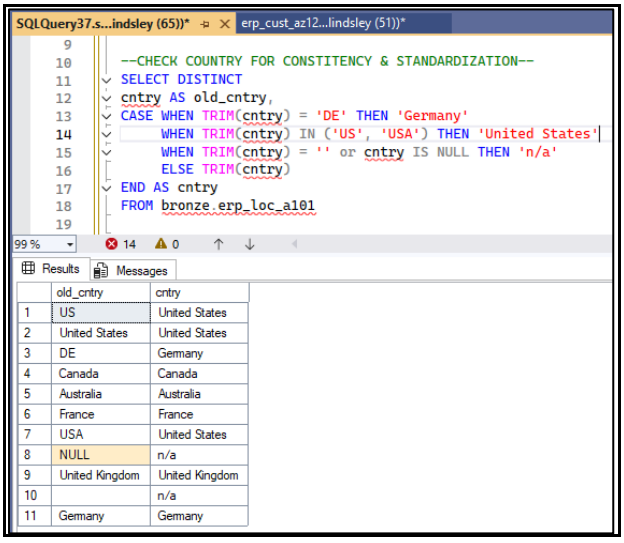
I ran a SELECT DISTINCT query to identify all possible values and identify needs for standardization. There is a mix of abbreviations, fully spelled out country names, NULLs, and empty values.



	cntry
1	DE
2	USA
3	Germany
4	United States
5	NULL
6	Australia
7	United Kingdom
8	
9	Canada
10	France
11	US

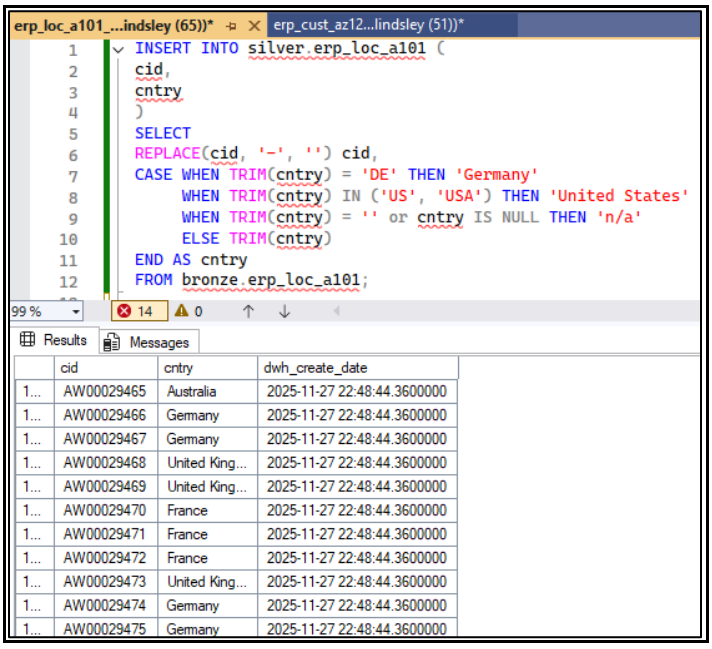
The following query was written and added to the transformation query to correct for the inconsistent country identifiers. I also added the same CASE WHEN to the previous query to identify how the matches mapped to their original values to confirm correction.





	old_cntry	cntry
1	US	United States
2	United States	United States
3	DE	Germany
4	Canada	Canada
5	Australia	Australia
6	France	France
7	USA	United States
8	NULL	n/a
9	United Kingdom	United Kingdom
10		n/a
11	Germany	Germany

Insert loc_a101 into silver layer
I wrote an INSERT INTO query to insert the transformed data from the bronze layer into the silver layer and ran a query to review and confirm that corrections to cid and cntry fields appeared correctly in the silver layer



	cid	cntry	dwh_create_date
1...	AW00029465	Australia	2025-11-27 22:48:44.3600000
1...	AW00029466	Germany	2025-11-27 22:48:44.3600000
1...	AW00029467	Germany	2025-11-27 22:48:44.3600000
1...	AW00029468	United King...	2025-11-27 22:48:44.3600000
1...	AW00029469	United King...	2025-11-27 22:48:44.3600000
1...	AW00029470	France	2025-11-27 22:48:44.3600000
1...	AW00029471	France	2025-11-27 22:48:44.3600000
1...	AW00029472	France	2025-11-27 22:48:44.3600000
1...	AW00029473	United King...	2025-11-27 22:48:44.3600000
1...	AW00029474	Germany	2025-11-27 22:48:44.3600000
1...	AW00029475	Germany	2025-11-27 22:48:44.3600000

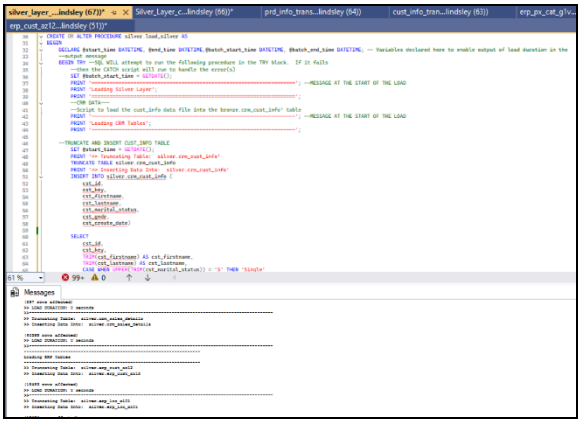
ERP → px_cat_g1v2 (The table with product category ids, product categories, subcategories, and maintenance identifier)
Checking primary key (id) to determine what needs to be done to connect to crm_prod_info via the cat_id that was created earlier in for the silver layer of this table
a. Running separate queries against both tables allows me to see the values cat_id in crm_prod_info appears to be in the same format and structure as the values listed in the id field of px_cat_g1v2 and is ready to be used for mapping

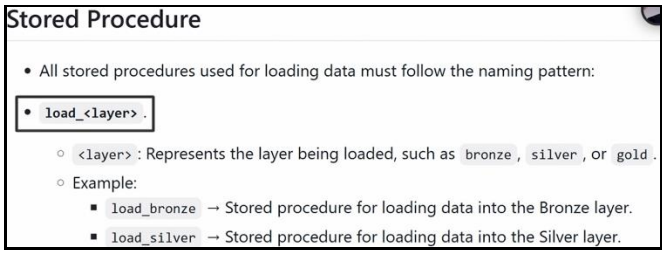
Checking the "cat" field to ensure consistency, removal of unwanted spaces, etc.
a. Running a query to identify unwanted spaces across all fields returned zero results
b. Running a query to identify inconsistencies in the cat field resulted in no consistent value usage
c. Running a query to identify inconsistencies in the subcat field resulted in no consistent value usage
c. Running a query to identify inconsistencies in the maintenance field resulted in no consistent value usage

With all checks complete, I wrote the INSERT INTO query to place the data into the silver layer. Then built a query that consolidated the INSERT queries for all tables in the silver layer with TRUNCATE TABLE commands for each.

Creating a stored procedure:

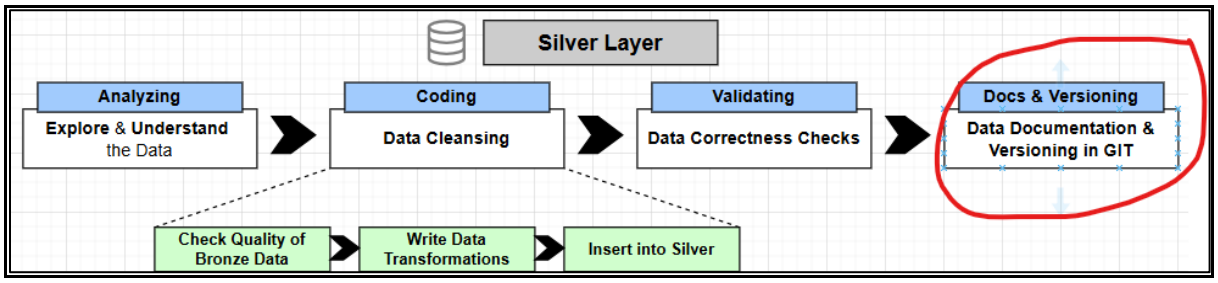
a. I wrote a stored procedure for the loading of the data into the database as it is intended to be a recurring action to bring in the data.



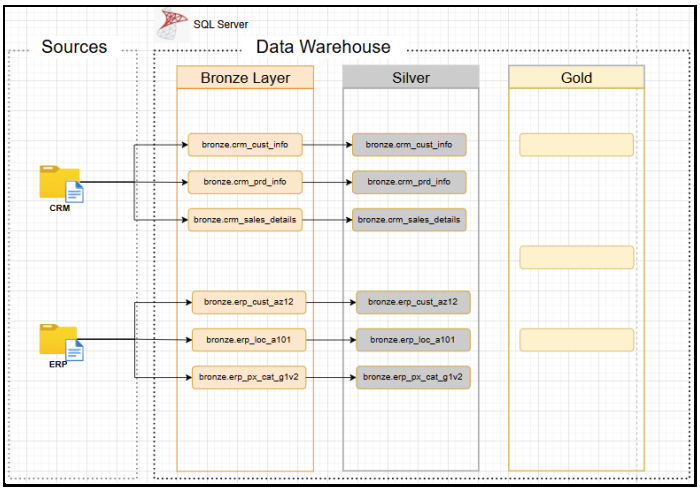


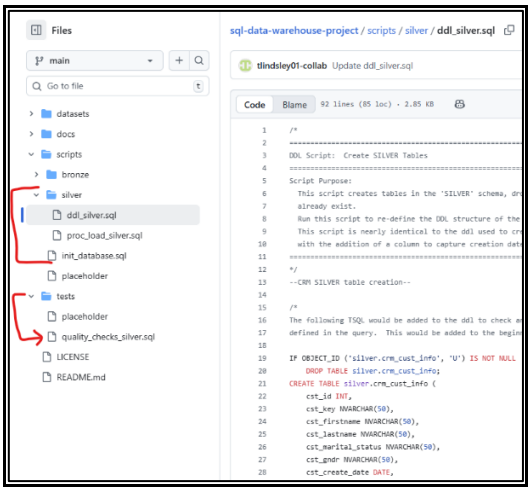
Once executing the stored procedure and refreshing the DB to visually confirm its existence, I then created a new query to test the execution of the procedure. The following SQL statement was used: EXEC silver..load_silver

d. Documenting & Commit Code in GIT

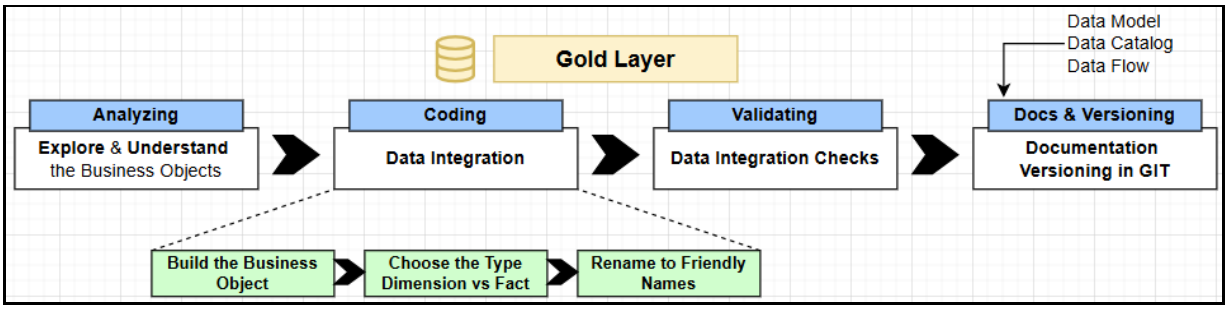


I updated the data flow to reflect the lineage of data from source systems to bronze to silver. I also loaded and committed my ddl script, stored procedure, and quality checks for the silver layer to my Git repo.





8. Build Gold Layer



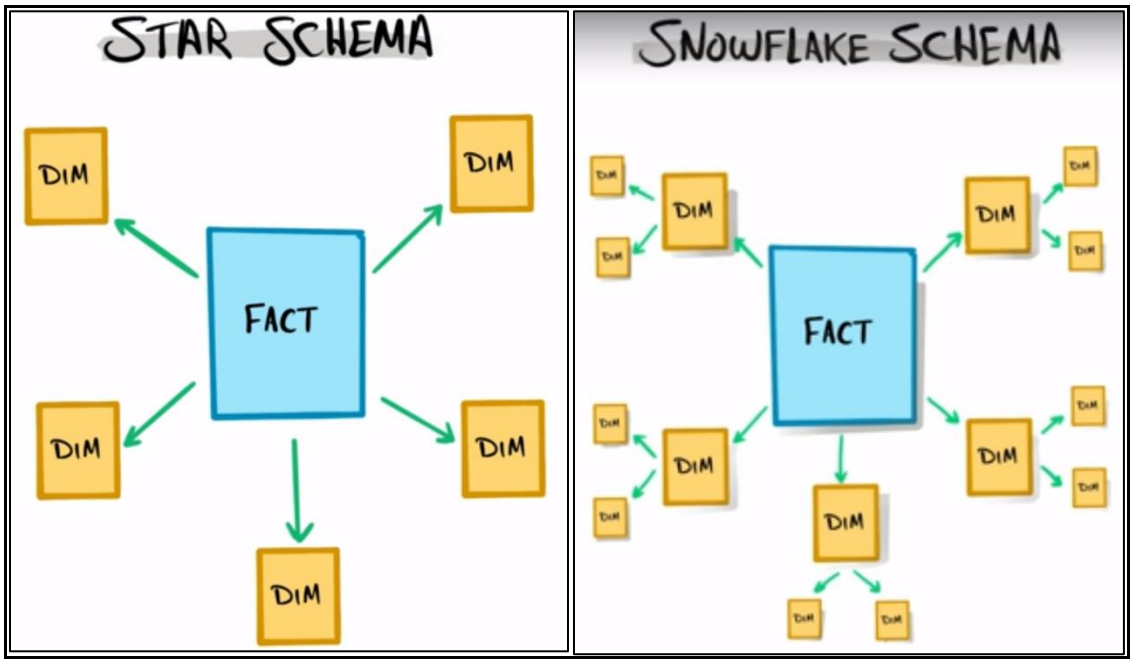
a. Analyzing: Explore Business Objects

Data Model Selection
The first step in this task is to select the logical data model to enable development of relationships between data sets. Additionally, it serves as a reference for future users of the data warehouse.

The data model needs to be optimized for data analytics. It also needs to be flexible, scalable and easy to understand. The types of data models typically used are Star Schema and Snowflake Schema.

Star Schema: A central fact table in the center and surrounded by dimensions. The fact table contains transactional / event-oriented data, and the dimensions contain descriptive information and the relationship between the fact table and the dimensions around it form a star shape when depicted visually….hence "Start Schema". It is simple in design and easy to query; however dimensions might have duplicates and they get larger over time. Commonly used and ideal for BI tools like Tableau. Star Schema will be used for this project

Snowflake Schema: Similar to the Star Schema in that the fact table is the central component of the model and is surrounded by dimensions; however, each dimension is broken out further into sub-dimensions. This extension into sub-dimensions results in a snowflake pattern when depicted visually. Hence…the name "Snowflake Schema". The Snowflake Schema is more complex, requires deeper knowledge of the underlying data sets and requires more effort to query. The main advantage comes through normalization from breaking redundancies into smaller tables which optimizes storage.

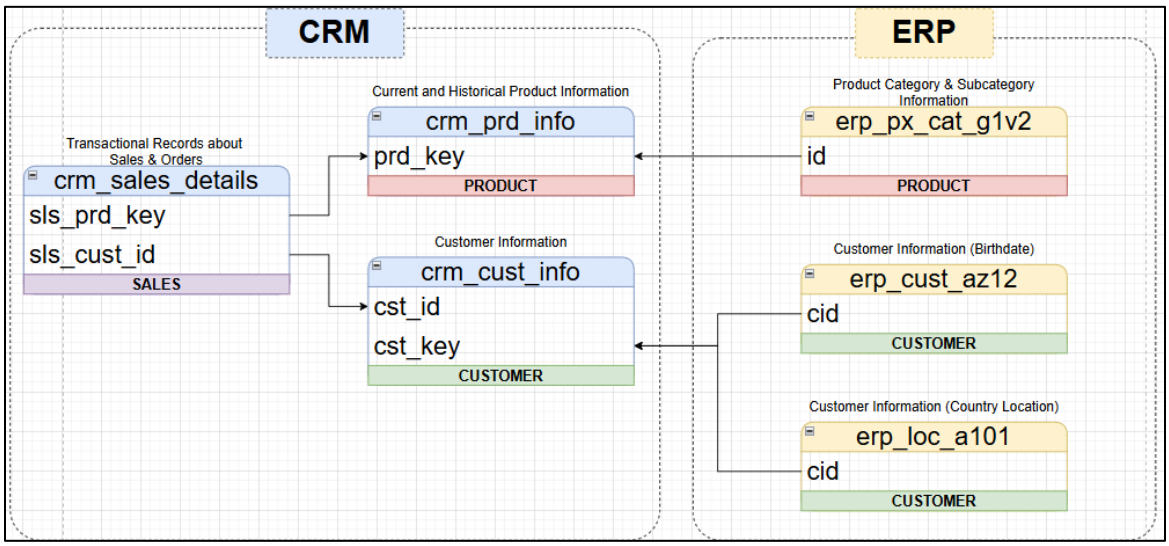


Data Model Design & Build

The dimensions and facts need to be defined now that Star Schema has been selected. Dimension tables will contain descriptive information to give context to the data (example: product info). If the data is oriented around who, what, when and where…then it belongs in the dimension table. Fact tables contain quantitative information that represent events / transactions (Example: sales orders, shipments, returns with dates, and other numerical data). If the data is oriented around how much, when, etc.…then it belongs in the Fact table.

The first step here is to understand the business context of the data to ensure data is correctly identified / assigned to dimension tables and fact table. In a real-world situation, I would have met with the business stakeholders / process SMEs / process owners to walk-through the data and its alignment to relevant business processes.

Before doing so, it is important to explore the data and develop an initial understanding and documenting it to enable review and validation with the business SMEs. As this is a fictional situation without stakeholders to review, I merely prepared the below visual representation of the data objects and their relationships that will serve as the foundation for building out the Star Schema for the Gold Layer.



b. Coding & Validating: Data Integration & Quality Checks

Build of the Gold Layer is focused on creating the Star Schema in the data warehouse with dimensions and facts defined to build Views. This will be done by integrating Silver Layer data into business-friendly objects to enable data consumption for reporting & analytics.

Integrating Customer Data

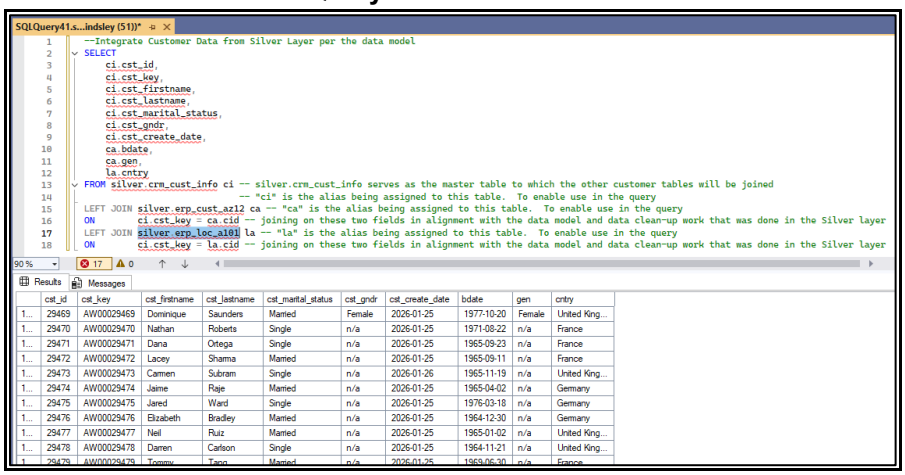
There are three tables that need to be integrated to bring customer data together into a single view in the Gold layer from the Silver layer. Those are:

- silver.crm_cust_info
- silver.erp_cust_az12
- silver.erp_loc_a101

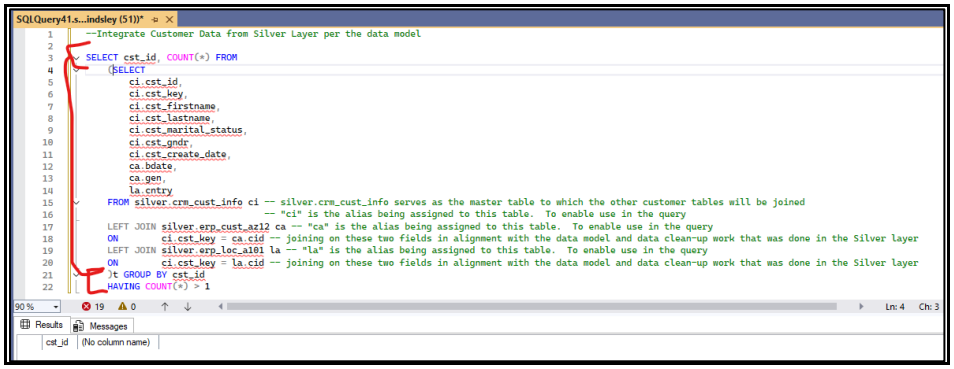
I created a join of these three tables with 'silver.crm_cust_info' serving as the master table.

Once created, I ran the query and ran a check to ensure there are no duplicates in the primary keys. If duplicates appear as a result of the check, then that is an indication that there is an issue with the Silver layer data that would need to be resolved before moving forward with the Gold Layer.

Initial Query to Join Customer Data



Query & Results Checking for Duplicate Primary Keys



The fields that contain customer gender in tables ‘crm_cust_info’ and ‘silver.erp_cust_az12’ were not cleansed for consistency during the build of the Silver layer. I would think that the correct approach would be to go back to the silver layer and make the corrections there, but….as I am merely following along with the instructor, I will cleanse them for consistency in the gold layer as he does.

Below is a view of the inconsistency that needs to be corrected. In a real world situation, I would have the business stakeholder / data owner confirm the system that serves as the source of truth for customer gender. In this fictitious situation, the crm is the source of truth. So, I will need to create a query that applies consistency and build it into the primary query for customer data.

The query written to apply consistency to customer gender data will leverage a CASE WHEN to use the value in silver.crm_cust_info if the value is not ‘n/a’. If the value is ‘n/a’, the it will pull the gender from silver.erp_cust_az12. If it is n/a in both, then the value remains as ‘n/a’.

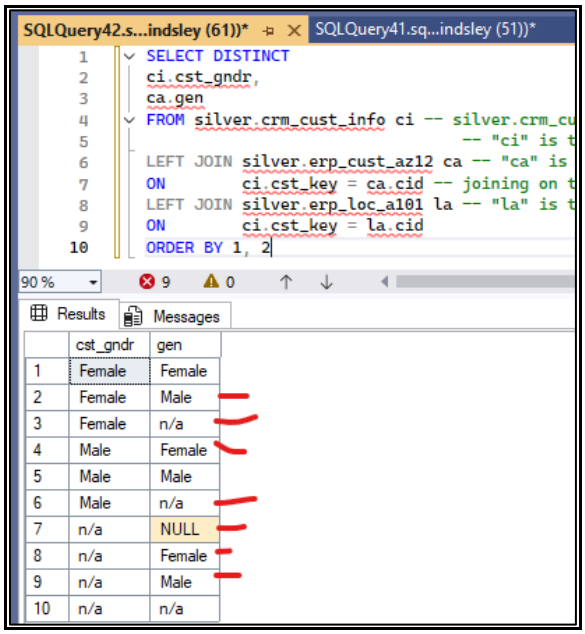
The query built to apply consistency was added to the customer info data integration query. I then applied business-friendly names to each field, reordered the columns to improve readability.

The integrated customer info is a dimension as it contains descriptive data rather than events, transactions, or any measurements.

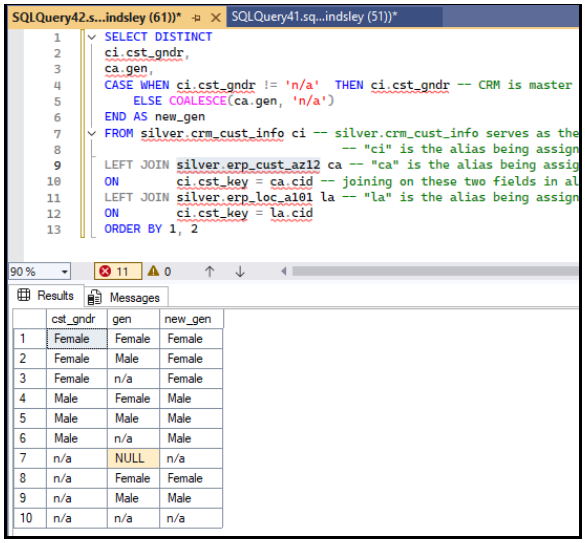
As dimension object, it will need a primary key defined for each record. In this project, a surrogate key will be created using a Window function (Row_Number) so that there is no dependency on use of the primary keys of integrated tables.

Surrogate Key = System-generated unique identifier assigned to each record in a table. They can be defined in the ddl or they can be query based using a Window function (Row_Number)

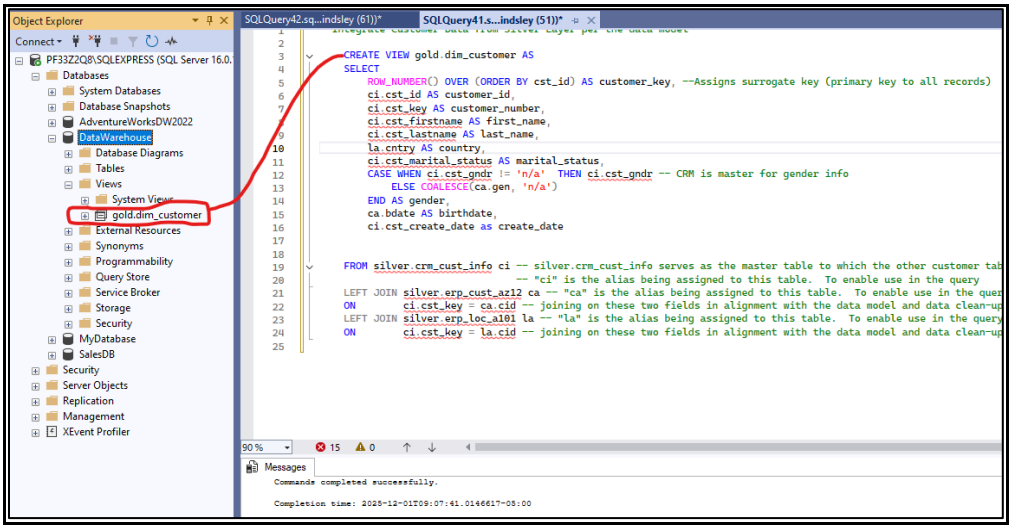
Query Identifying Inconsistency in Gender Identifiers



Query Applying Consistency to Gender Data



CREATE CUSTOMER VIEW



CREATE CUSTOMER VIEW

The final step in the creation of the Customer View is to conduct a quality check of the data. To do so, I ran a 'SELECT *' query and conducted a quick scan to ensure all data appeared in the correct order and that there are no unexpected NULL values. I also ran a 'SELECT DISTINCT' query on the 'gender' field to ensure that there are no values in the field other than 'Male', 'Female', or 'n/a'. Results of both queries indicated success.

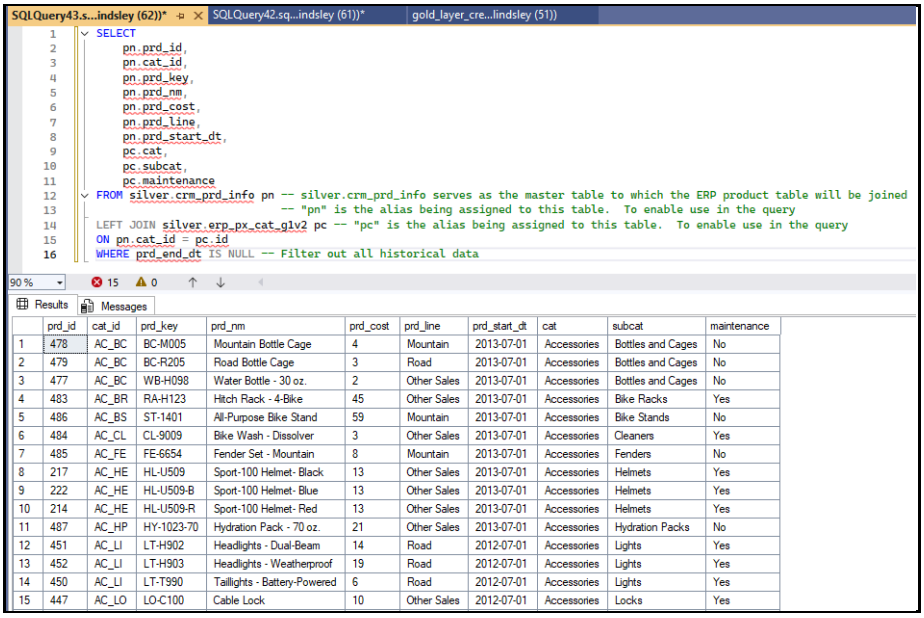
Integrating Product Data

1. Product data exists in both the CRM and the ERP data sources. Integrating these into a single, usable view is necessary to enable analytics on the data.

The product info table contains historical and current product information. In a real-world scenario, a decision would be needed as to whether the view must contain both historical and current product information for analysis. In this fictitious case, only the current product data will be used in the view. The product key is being used in the mapping of product data between tables. This also allow for the historical data to be filtered out so only current product data is included.

To filter out historical data, a WHERE condition was applied to the query on the prd_end_dt field to filter out records where the value != NULL. All records with NULL in the prd_end_dt field are active products because the product end date has not occurred.

Query Joining Product Data from CRM and ERP tables



```
SELECT
    pn.prd_id,
    pn.cat_id,
    pn.prd_key,
    pn.prd_nm,
    pn.prd_cost,
    pn.prd_line,
    pn.prd_start_dt,
    pc.cat,
    pc.subcat,
    pc.maintenance
FROM silver.crm_prd_info pn -- silver.crm_prd_info serves as the master table to which the ERP product table will be joined
                            -- "pn" is the alias being assigned to this table.  To enable use in the query
LEFT JOIN silver.erp_px_cat_g1v2 pc -- "pc" is the alias being assigned to this table.  To enable use in the query
ON pn.cat_id = pc.id
WHERE prd_end_dt IS NULL -- Filter out all historical data
```

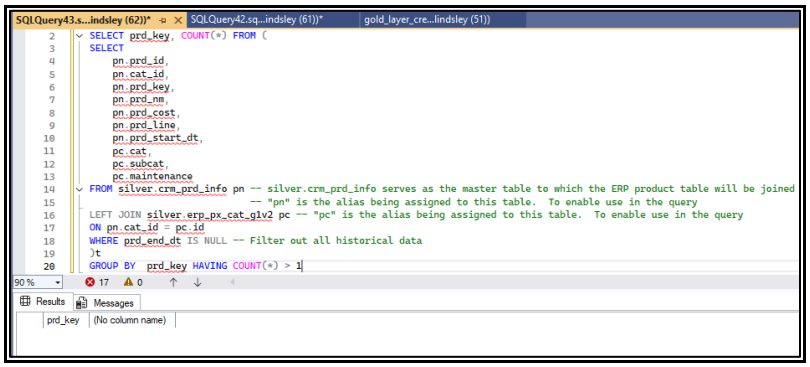
Results | Messages

	prd_id	cat_id	prd_key	prd_nm	prd_cost	prd_line	prd_start_dt	cat	subcat	maintenance
1	478	AC_BC	BC-M005	Mountain Bottle Cage	4	Mountain	2013-07-01	Accessories	Bottles and Cages	No
2	479	AC_BC	BC-R205	Road Bottle Cage	3	Road	2013-07-01	Accessories	Bottles and Cages	No
3	477	AC_BC	WB-H098	Water Bottle - 30 oz.	2	Other Sales	2013-07-01	Accessories	Bottles and Cages	No
4	483	AC_BR	RA-H123	Hitch Rack - 4-Bike	45	Other Sales	2013-07-01	Accessories	Bike Racks	Yes
5	486	AC_BS	ST-1401	All-Purpose Bike Stand	59	Mountain	2013-07-01	Accessories	Bike Stands	No
6	484	AC_CL	CL-9009	Bike Wash - Dissolver	3	Other Sales	2013-07-01	Accessories	Cleaners	Yes
7	485	AC_FE	FE-6654	Fender Set - Mountain	8	Mountain	2013-07-01	Accessories	Fenders	No
8	217	AC_HE	HL-U509	Sport-100 Helmet- Black	13	Other Sales	2013-07-01	Accessories	Helmets	Yes
9	222	AC_HE	HL-U509-B	Sport-100 Helmet- Blue	13	Other Sales	2013-07-01	Accessories	Helmets	Yes
10	214	AC_HE	HL-U509-R	Sport-100 Helmet- Red	13	Other Sales	2013-07-01	Accessories	Helmets	Yes
11	487	AC_HP	HY-1023-70	Hydration Pack - 70 oz.	21	Other Sales	2013-07-01	Accessories	Hydration Packs	No
12	451	AC_LI	LT-H902	Headlights - Dual-Beam	14	Road	2012-07-01	Accessories	Lights	Yes
13	452	AC_LI	LT-H903	Headlights - Weatherproof	19	Road	2012-07-01	Accessories	Lights	Yes
14	450	AC_LI	LT-T990	Taillights - Battery-Powered	6	Road	2012-07-01	Accessories	Lights	Yes
15	447	AC_LO	LO-C100	Cable Lock	10	Other Sales	2012-07-01	Accessories	Locks	Yes

Conduct Data Quality Checks

The check for data quality, I wrote a SELECT COUNT(*) query with grouping by prd_key for records with prd_key appearing more than once. There were no records displayed in results the quality check query. That tells me that:

- There are no duplicates resulting from the join
- Each product has only one record
- There are no historical data



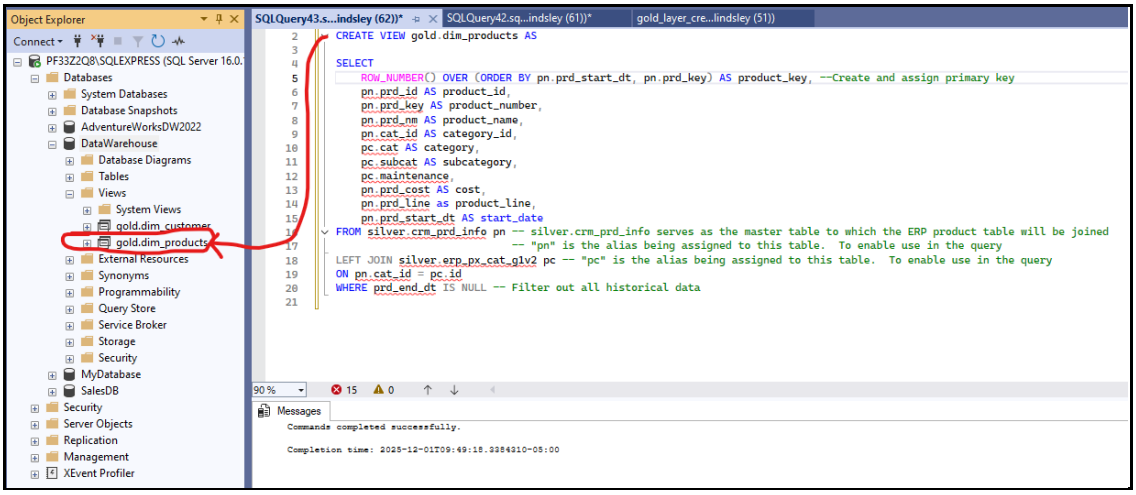
```
SELECT prd_key, COUNT(*) FROM (
SELECT
    pn.prd_id,
    pn.cat_id,
    pn.prd_key,
    pn.prd_nm,
    pn.prd_cost,
    pn.prd_line,
    pn.prd_start_dt,
    pc.cat,
    pc.subcat,
    pc.maintenance
FROM silver.crm_prd_info pn -- silver.crm_prd_info serves as the master table to which the ERP product table will be joined
                            -- "pn" is the alias being assigned to this table.  To enable use in the query
LEFT JOIN silver.erp_px_cat_g1v2 pc -- "pc" is the alias being assigned to this table.  To enable use in the query
ON pn.cat_id = pc.id
WHERE prd_end_dt IS NULL -- Filter out all historical data
)t
GROUP BY  prd_key HAVING COUNT(*) > 1
```

Results | Messages

prd_key	(No column name)

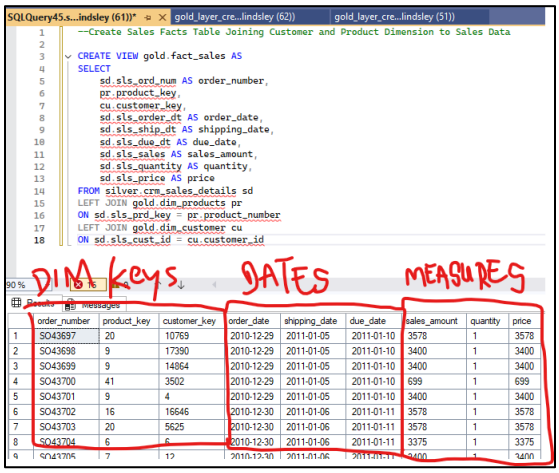
The field names were then assigned business-friendly names and columns were reordered for readability. Additionally a the query was enhanced to include the creation of a surrogate key (primary key) for each product using the ROW_NUMBER window function (similar to what was done for the customer info dimension.

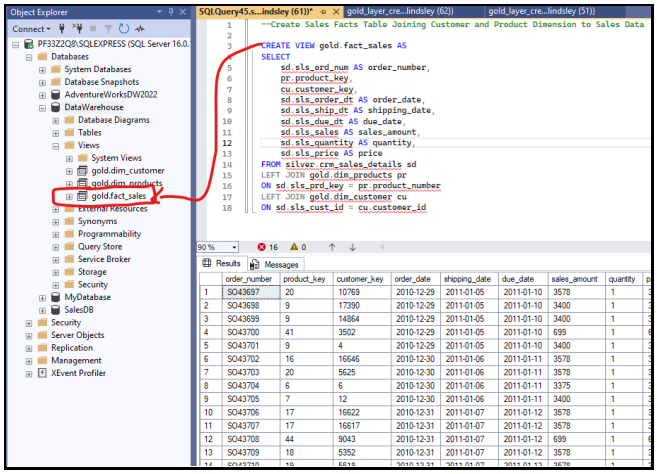
CREATE VIEW was added to the query, then the query was executed…resulting in the creation of the view in the data warehouse.



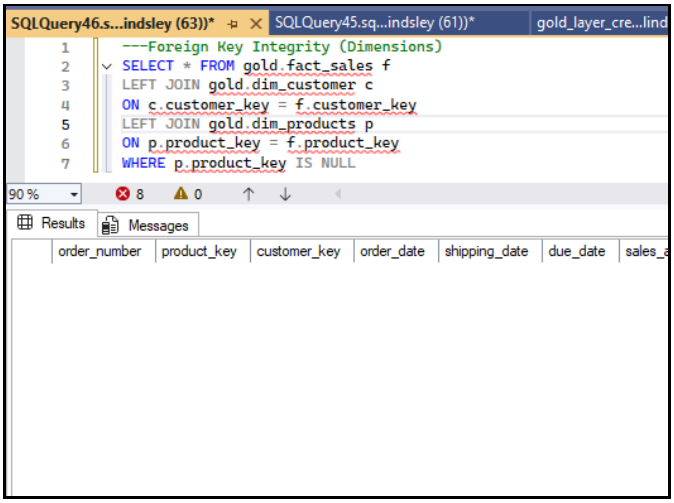
Building the View of Sales Data as a Facts Table

What I did - connected Customer and Product dimensions (product ID, customer ID) to the sales data





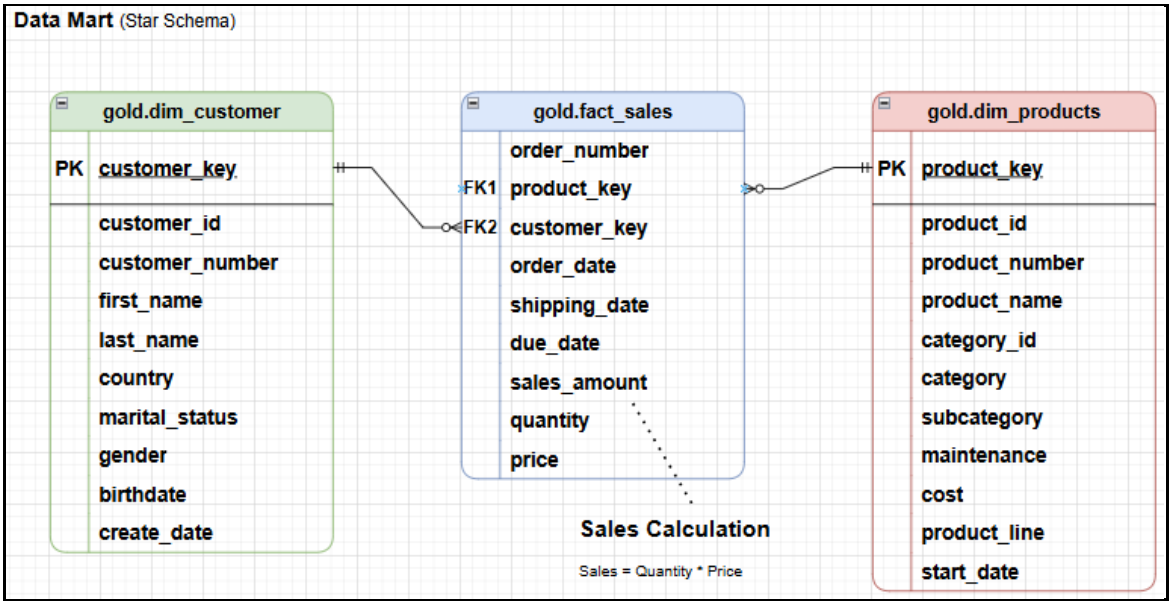
Checking Foreign Key Integrity (dimensions) to see if all dimension tables can successfully join to the fact table: No bad results….good to go



c. Document & Commit Code to Git

Document Star Schema

The Star Schema was modeled to visually depict the relationships defined between the customer dimension, the products dimension and the sales Facts views in the Gold Layer. It includes identification of primary keys, foreign keys, and the relationships (many to one / one to many)

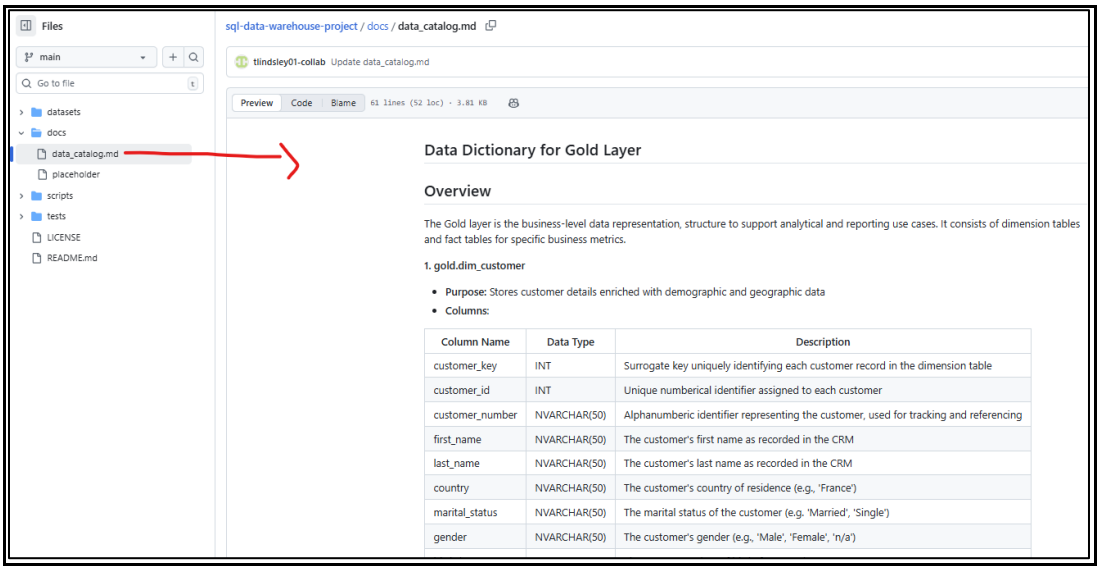


Create Data Catalog & Document Versioning in Git

A Data Catalog is needed to support the data product we have created (the data warehouse) to be developed based on the data warehouse. The Data Catalog describes the data model (it's columns, tables, relationships, etc) to enable users to derive meaningful insights and to avoid repeat questions from users. The data catalog is documented in an md file in my Git repository.

The Data Catalog has a section for each table in the Gold layer (gold.dim_customer, gold.dim_products, and gold.facts_sales). Each section includes the purpose of the table, the column names, data types, and column descriptions (with examples).

I defined and documented the data catalog in a markdown file and committed it to my Git repo for this project under the ‘docs’ folder.



Data Dictionary for Gold Layer

Overview

The Gold layer is the business-level data representation, structure to support analytical and reporting use cases. It consists of dimension tables and fact tables for specific business metrics.

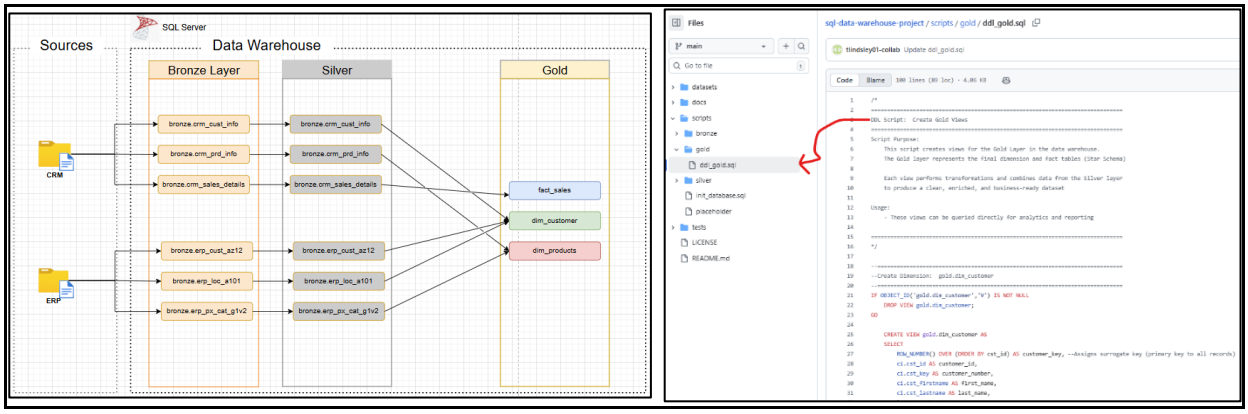
1. gold.dim_customer

- **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 numberical identifier assigned to each customer
customer_number	NVARCHAR(50)	Alphanumberic identifier representing the customer, used for tracking and referencing
first_name	NVARCHAR(50)	The customer's first name as recorded in the CRM
last_name	NVARCHAR(50)	The customer's last name as recorded in the CRM
country	NVARCHAR(50)	The customer's country of residence (e.g., 'France')
marital_status	NVARCHAR(50)	The marital status of the customer (e.g. 'Married', 'Single')
gender	NVARCHAR(50)	The customer's gender (e.g., 'Male', 'Female', 'n/a')

Update the Data Flow Diagram to show Gold Layer and commit Gold Layer view ddl to Git: I updated the data flow diagram to include the dimension tables for customer and product data along with the fact table for sales data. This view provides a visual representation of the data lineage for the data warehouse.

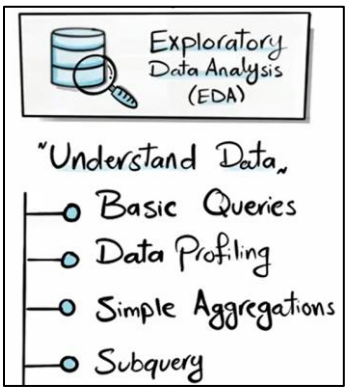
I also combined all gold layer creation scripts into a single ddl sql and uploaded to git with the scripts written for quality checks in the gold layer and all final diagrams and data flows created throughout this project.



9. Exploratory SQL Data Analysis

a. Overview

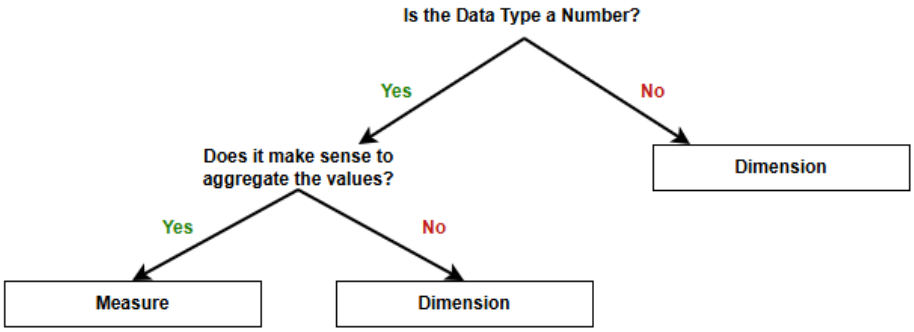
In this stage of the project, I will leverage the basic SQL skills I've learned throughout this effort to conduct data profiling, simple aggregation, and use of subqueries to further understand the data sets.



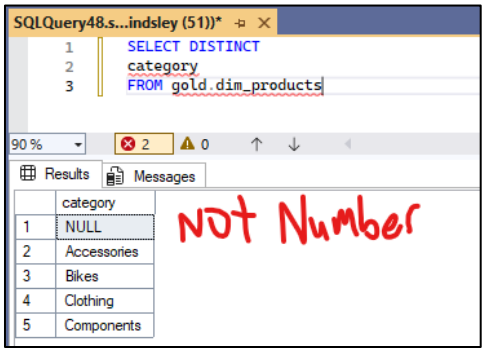
b. Dimensions vs Measures

The data in the gold layer of my database is separated into dimensions (dim tables) and measures (the fact table).

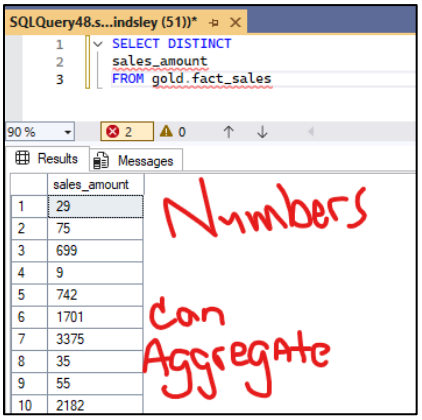
The data sets were evaluated according to the following to enable determination as to whether they should be defined as dimensions or tables.



Dimension Example: Looking at the 'category' field in the gold.dim_products view within the Gold layer, each of the values are strings (not numbers). Therefore, these are correctly categorized as dimension values and reside in a 'dim' view of the Gold layer.



Measure Example: Looking at the 'sales_amount' field in the 'gold.fact_sales' view within the Gold layer, each of the values are numbers. As such, they can be aggregated and are therefore correctly categorized as measures and reside in a 'fact' view of the Gold layer for analysis.



I reviewed data all fields across each data set in the gold layer to ensure I have each field correctly identified as a dimension or measure. The table below provides a consolidated view for my reference.

View	Column Name	Number?	Can be Aggregated?	Dimension or Measure	Rationale
gold.dim_customer	customer_key	Yes	Yes	Dimension	Values are numbers, but not measurable values.
	customer_id	Yes	Yes	Dimension	Values are numbers, but not measurable values.
	customer_number	No	No	Dimension	Values are combination of text and numbers. Not measurable values.
	first_name	No	No	Dimension	Values are text. Cannot measure.
	last_name	No	No	Dimension	Values are text. Cannot measure.
	country	No	No	Dimension	Values are text. Cannot measure, but can categorized / grouped for analysis
	marital_status	No	No	Dimension	Values are text. Cannot measure, but can categorized / grouped for analysis
	gender	No	No	Dimension	Values are text. Cannot measure, but can categorized / grouped for analysis
	birthdate	No	No	Dimension	Values are dates. Cannot measure, but can derive 'age' from this column to measure
	create_date	No	No	Dimension	Values are dates. Cannot measure, but can derive 'years as customer' from this column to measure
gold.dim_products	product_key	Yes	Yes	Dimension	Values are numbers, but not measurable values.
	product_id	Yes	Yes	Dimension	Values are numbers, but not measurable values.
	product_number	No	No	Dimension	Values are combination of text and numbers. Not measurable values.
	product_name	No	No	Dimension	Values are combination of text and numbers. Not measurable values.
	category_id	No	No	Dimension	Values are text. Cannot measure, but can categorized / grouped for analysis
	category	No	No	Dimension	Values are text. Cannot measure, but can categorized / grouped for analysis
	subcategory	No	No	Dimension	Values are text. Cannot measure, but can categorized / grouped for analysis
	maintenance	No	No	Dimension	Values are text. Cannot measure, but can categorized / grouped for analysis
	cost	Yes	Yes	Measure	Values are numbers and measurable values
	product_line	No	No	Dimension	Values are text. Cannot measure, but can categorized / grouped for analysis
	start_date	No	No	Dimension	Values are dates. Cannot measure, but can derive 'years as active product' from this column to measure
gold.fact_sales	order_number	No	No	Dimension	Values are combination of text and numbers. Not measurable values.

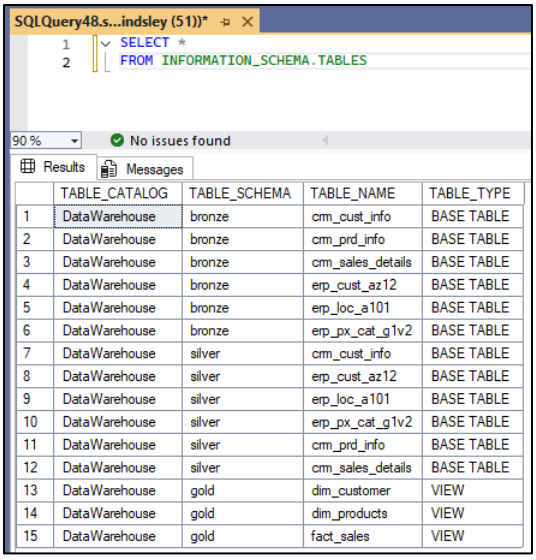
View	Column Name	Number?	Can be Aggregated?	Dimension or Measure	Rationale
	product_key	Yes	Yes	Dimension	Values are numbers, but not measurable values.
	customer_key	Yes	Yes	Dimension	Values are numbers, but not measurable values.
	order_date	No	No	Dimension	Values are dates. Cannot measure, but can derive 'days from order to ship' and 'days from order to due date' this column to measure
	shipping_date	No	No	Dimension	Values are dates. Cannot measure, but can derive 'days from order to ship' and 'days aging past due' from this column to measure
	due_date	No	No	Dimension	Values are dates. Cannot measure, but can derive days from order to due date' and 'days aging past due' from this column to measure
	sales_amount	Yes	Yes	Measure	Can measure sales across a variety of dimensions and derive additional measures
	quantity	Yes	Yes	Measure	Can measure quantity sold across a variety of dimensions and derive additional measures
	price	Yes	Yes	Measure	Can measure sales price across a variety of dimensions and derive additional measures (revenue using cost)

Identifying dimensions and measures allows me to identify which fields can be used to group / categorize data to answer questions on measurable values. For example, if I want to know the total sales by product category….I know that 'category' is a dimension and 'sales_amount' is the measure. Having this clarity enables effective analysis.

c. Database Exploration

I already know the layout and underlying structure of the data warehouse, however, if I were not familiar with it…I would need to explore it to develop a quick understanding. Running a handful of queries and reviewing the content and structure would give me this understanding.

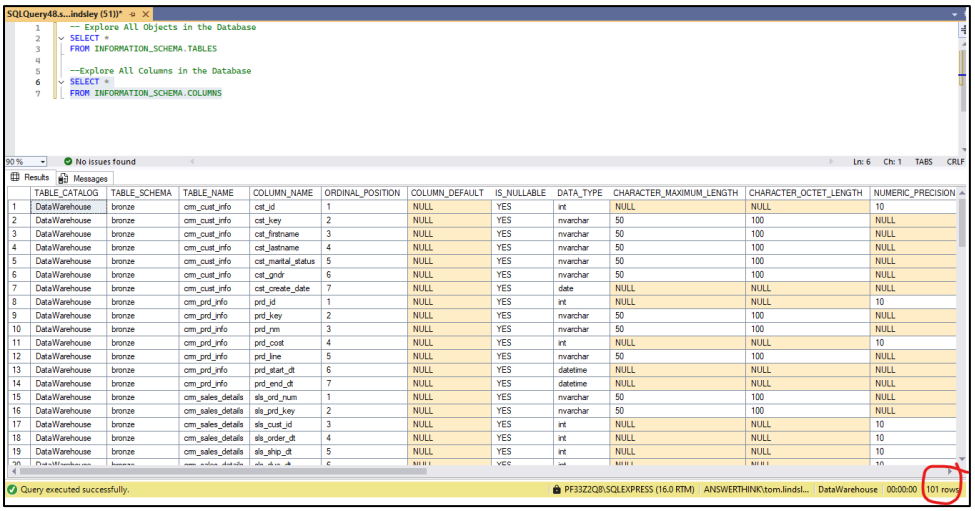
Running a 'SELECT *' query on the 'INFORMATION_SCHEMA.TABLES' gives me a high-level understanding of the tables, schema, and table types. Here I can see the medallion architecture, the tables within each layer, and the object type within each layer (base vs view).



	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE
1	DataWarehouse	bronze	crm_cust_info	BASE TABLE
2	DataWarehouse	bronze	crm_prd_info	BASE TABLE
3	DataWarehouse	bronze	crm_sales_details	BASE TABLE
4	DataWarehouse	bronze	erp_cust_az12	BASE TABLE
5	DataWarehouse	bronze	erp_loc_a101	BASE TABLE
6	DataWarehouse	bronze	erp_px_cat_g1v2	BASE TABLE
7	DataWarehouse	silver	crm_cust_info	BASE TABLE
8	DataWarehouse	silver	erp_cust_az12	BASE TABLE
9	DataWarehouse	silver	erp_loc_a101	BASE TABLE
10	DataWarehouse	silver	erp_px_cat_g1v2	BASE TABLE
11	DataWarehouse	silver	crm_prd_info	BASE TABLE
12	DataWarehouse	silver	crm_sales_details	BASE TABLE
13	DataWarehouse	gold	dim_customer	VIEW
14	DataWarehouse	gold	dim_products	VIEW
15	DataWarehouse	gold	fact_sales	VIEW

I can also explore all of the columns in the database by executing a ‘SELECT *’ query on the ‘INFORMATION_SCHEMA.COLUMNS’.

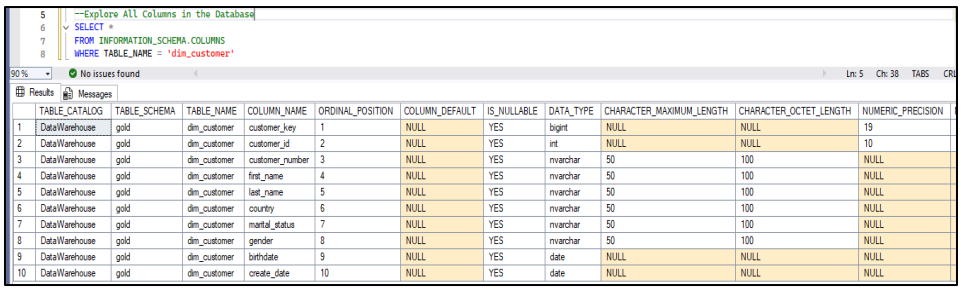
The results of this query provides me with the full inventory of all columns in the database with key metadata for each column, and I can see that there are 101 columns across the entire data warehouse.



```
-- Explore All Objects in the Database
SELECT *
FROM INFORMATION_SCHEMA.TABLES

--Explore All Columns in the Database
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
```

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	CHARACTER_MAXIMUM_LENGTH	CHARACTER_OCTET_LENGTH	NUMERIC_PRECISION
1	DataWarehouse	bronze	crm_cust_info	cst_id	1	NULL	YES	int	NULL	NULL	10
2	DataWarehouse	bronze	crm_cust_info	cst_key	2	NULL	YES	nvarchar	50	100	NULL
3	DataWarehouse	bronze	crm_cust_info	cst_firstname	3	NULL	YES	nvarchar	50	100	NULL
4	DataWarehouse	bronze	crm_cust_info	cst_lastname	4	NULL	YES	nvarchar	50	100	NULL
5	DataWarehouse	bronze	crm_cust_info	cst_marital_status	5	NULL	YES	nvarchar	50	100	NULL
6	DataWarehouse	bronze	crm_cust_info	cst_gndr	6	NULL	YES	nvarchar	50	100	NULL
7	DataWarehouse	bronze	crm_cust_info	cst_create_date	7	NULL	YES	date	NULL	NULL	NULL
8	DataWarehouse	bronze	crm_prd_info	prd_id	1	NULL	YES	int	NULL	NULL	10
9	DataWarehouse	bronze	crm_prd_info	prd_key	2	NULL	YES	nvarchar	50	100	NULL
10	DataWarehouse	bronze	crm_prd_info	prd_nm	3	NULL	YES	nvarchar	50	100	NULL
11	DataWarehouse	bronze	crm_prd_info	prd_cost	4	NULL	YES	int	NULL	NULL	10
12	DataWarehouse	bronze	crm_prd_info	prd_line	5	NULL	YES	nvarchar	50	100	NULL
13	DataWarehouse	bronze	crm_prd_info	prd_start_dt	6	NULL	YES	datetime	NULL	NULL	NULL
14	DataWarehouse	bronze	crm_prd_info	prd_end_dt	7	NULL	YES	datetime	NULL	NULL	NULL
15	DataWarehouse	bronze	crm_sales_details	sls_ord_num	1	NULL	YES	nvarchar	50	100	NULL
16	DataWarehouse	bronze	crm_sales_details	sls_prd_key	2	NULL	YES	nvarchar	50	100	NULL
17	DataWarehouse	bronze	crm_sales_details	sls_cust_id	3	NULL	YES	int	NULL	NULL	10
18	DataWarehouse	bronze	crm_sales_details	sls_order_dt	4	NULL	YES	int	NULL	NULL	10
19	DataWarehouse	bronze	crm_sales_details	sls_ship_dt	5	NULL	YES	int	NULL	NULL	10

To see column metadata for a specific table I can filter using a WHERE clause as depicted in the example below for the ‘dim_customer’ table.



```
--Explore All Columns in the Database
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'dim_customer'
```

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	CHARACTER_MAXIMUM_LENGTH	CHARACTER_OCTET_LENGTH	NUMERIC_PRECISION
1	DataWarehouse	gold	dim_customer	customer_key	1	NULL	YES	bigint	NULL	NULL	19
2	DataWarehouse	gold	dim_customer	customer_id	2	NULL	YES	int	NULL	NULL	10
3	DataWarehouse	gold	dim_customer	customer_number	3	NULL	YES	nvarchar	50	100	NULL
4	DataWarehouse	gold	dim_customer	first_name	4	NULL	YES	nvarchar	50	100	NULL
5	DataWarehouse	gold	dim_customer	last_name	5	NULL	YES	nvarchar	50	100	NULL
6	DataWarehouse	gold	dim_customer	country	6	NULL	YES	nvarchar	50	100	NULL
7	DataWarehouse	gold	dim_customer	marital_status	7	NULL	YES	nvarchar	50	100	NULL
8	DataWarehouse	gold	dim_customer	gender	8	NULL	YES	nvarchar	50	100	NULL
9	DataWarehouse	gold	dim_customer	birthdate	9	NULL	YES	date	NULL	NULL	NULL
10	DataWarehouse	gold	dim_customer	create_date	10	NULL	YES	date	NULL	NULL	NULL

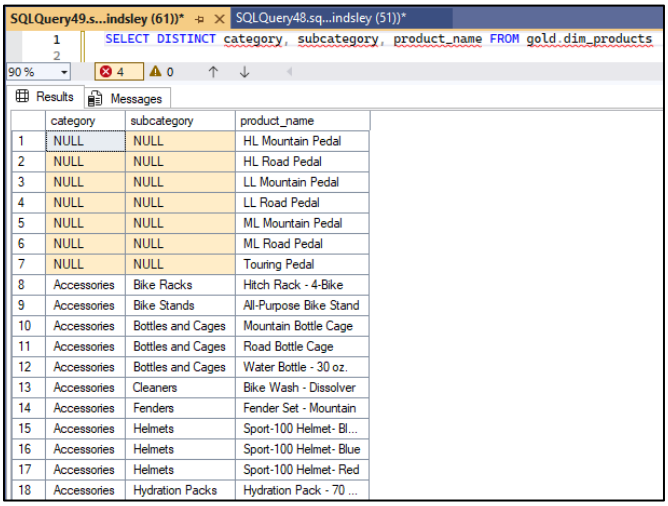
d. Dimensions Exploration

The purpose of dimension exploration is to identify the unique values (or categories) within each dimension so that you can develop an understanding of how data can be grouped or segmented for analysis.

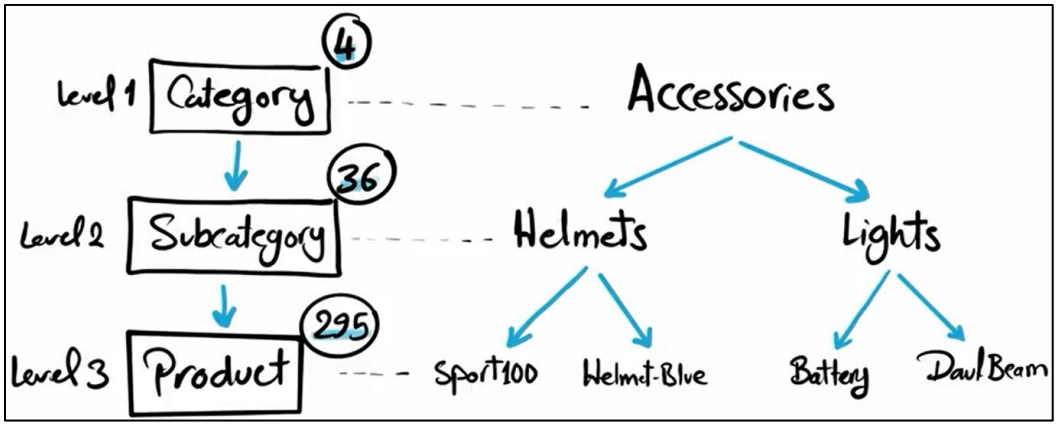
I explored each of the dimensions identified earlier using ‘SELECT DISTINCT’, and prepared the table below that consolidates the results of customer dimensions for my own reference.

Table	Dimension (column name)	Distinct Values
gold.dim_customer	country	n/a
		Australia
		Canada
		France
		Germany
		United Kingdom
		United States
	marital_status	Single
		Married
	gender	n/a
		Male
		Female

A similar exploration was done on the gold.dim_products table to understand the underlying distinct values, their relationships and hierarchy. This was accomplished by including the 'category', subcategory', and product_name' columns in the SELECT DISTINCT query. Results of the query revealed the various distinct values, the hierarchy, and relationships.



	category	subcategory	product_name
1	NULL	NULL	HL Mountain Pedal
2	NULL	NULL	HL Road Pedal
3	NULL	NULL	LL Mountain Pedal
4	NULL	NULL	LL Road Pedal
5	NULL	NULL	ML Mountain Pedal
6	NULL	NULL	ML Road Pedal
7	NULL	NULL	Touring Pedal
8	Accessories	Bike Racks	Hitch Rack - 4-Bike
9	Accessories	Bike Stands	All-Purpose Bike Stand
10	Accessories	Bottles and Cages	Mountain Bottle Cage
11	Accessories	Bottles and Cages	Road Bottle Cage
12	Accessories	Bottles and Cages	Water Bottle - 30 oz.
13	Accessories	Cleaners	Bike Wash - Dissolver
14	Accessories	Fenders	Fender Set - Mountain
15	Accessories	Helmets	Sport-100 Helmet- Bl...
16	Accessories	Helmets	Sport-100 Helmet- Blue
17	Accessories	Helmets	Sport-100 Helmet- Red
18	Accessories	Hydration Packs	Hydration Pack - 70 ...

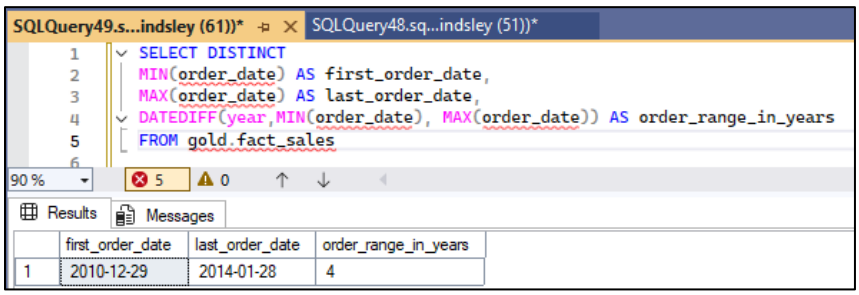


e. Date Exploration

Date exploration involves identifying the time boundaries in the data set (the earliest and latest dates), and understanding the scope of the data and timespan.

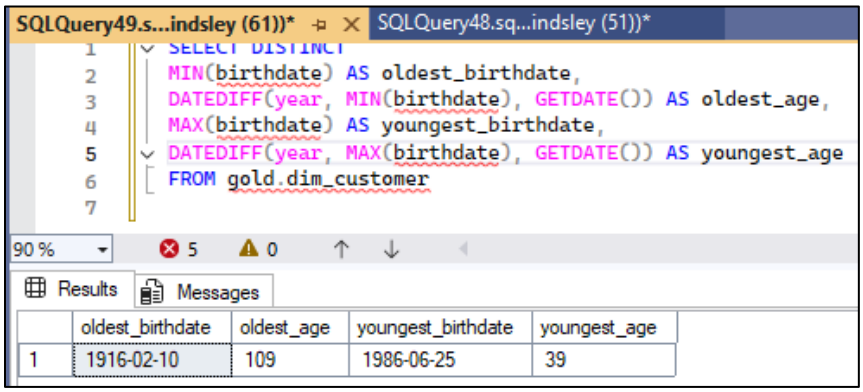
The MIN and MAX SQL functions applied to date columns will be used to identify date boundaries, while the DATEDIFF will be used to identify the timespan.

Identifying order date boundaries (first and last orders) and time span in years

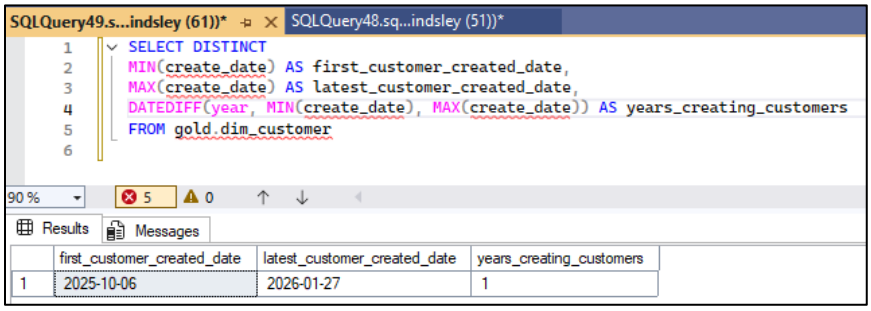


	first_order_date	last_order_date	order_range_in_years
1	2010-12-29	2014-01-28	4

Identifying date boundaries for customer birthdates (oldest and youngest customer)



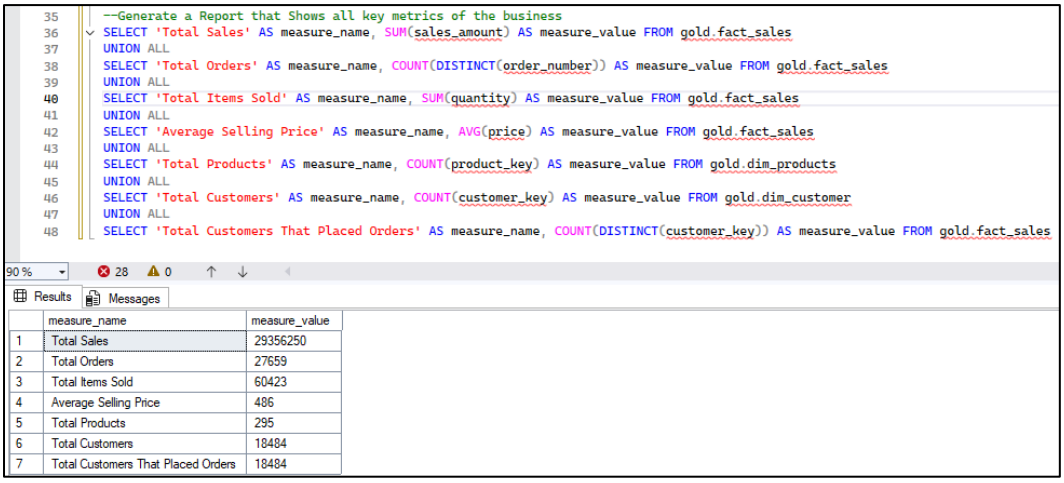
Identifying date boundaries for customer creation (first customer and most recent customer)



f. Measure Exploration

The purpose of this exploration is to calculate and identify the key metrics of the business at the highest and lowest levels of aggregation.

The SUM, AVG, COUNT, and COUNT(DISTINCT functions will be used will be used to identify the following at the highest level of aggregation. Each was written separately as an initial pass to identify and pressure test values, then consolidated into a single query using UNION ALL to produce key business metrics into a single view:



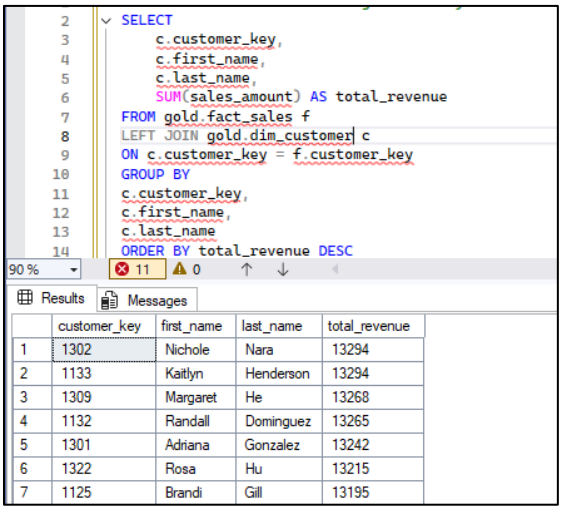
g. Magnitude Analysis

Magnitude analysis involves drilling further into the data at a lower level. As an example, let's say I wanted to identify the total revenue generated by each customer, the total revenue generated for each category, or the distribution of sold items across countries. Those questions would be answered in this type of analysis.

Total Sales by Customer

Viewing the total revenue for each customer requires building a LEFT JOIN from gold.fact_sales on the gold.dim_customer table with matching on the customer_key from both fields. It also requires obtaining the sum of sales for each customer from the fact_sales table and the customer_key, first_name, and last_name from the dim_customer table….then grouping by customer and ordering by total sales in DESC. Below is the query I wrote to generate the desired view. Results included all 18,484 customers and their total sales.

Total Sales by Customer



```
SELECT
    c.customer_key,
    c.first_name,
    c.last_name,
    SUM(sales_amount) AS total_revenue
FROM gold.fact_sales f
LEFT JOIN gold.dim_customer c
ON c.customer_key = f.customer_key
GROUP BY
c.customer_key,
c.first_name,
c.last_name
ORDER BY total_revenue DESC
```

90 % | 11 | 0

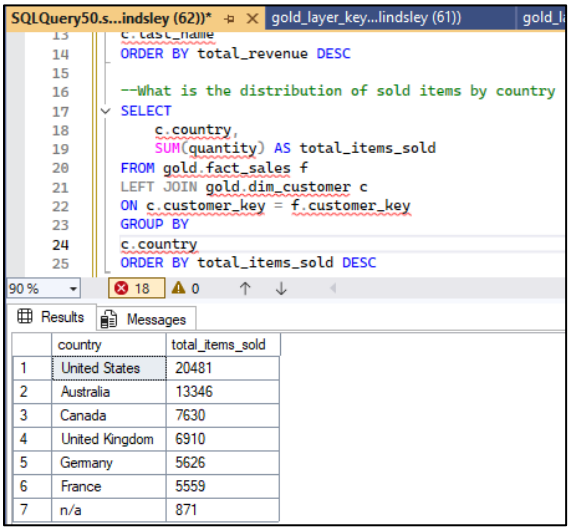
Results | Messages

	customer_key	first_name	last_name	total_revenue
1	1302	Nichole	Nara	13294
2	1133	Kaitlyn	Henderson	13294
3	1309	Margaret	He	13268
4	1132	Randall	Dominguez	13265
5	1301	Adriana	Gonzalez	13242
6	1322	Rosa	Hu	13215
7	1125	Brandi	Gill	13195

Total Items Sold by Country

Viewing the total items sold by country involves a similar approach using a LEFT JOIN. The following query was written to provide the desired view of total items sold by country.

Total Items Sold by Country



SQLQuery50.s...indsley (62))* | gold_layer_key...lindsley (61)) | gold_l

```
ORDER BY total_revenue DESC

--What is the distribution of sold items by country
SELECT
    c.country,
    SUM(quantity) AS total_items_sold
FROM gold.fact_sales f
LEFT JOIN gold.dim_customer c
ON c.customer_key = f.customer_key
GROUP BY
c.country
ORDER BY total_items_sold DESC
```

90 % | 18 | 0

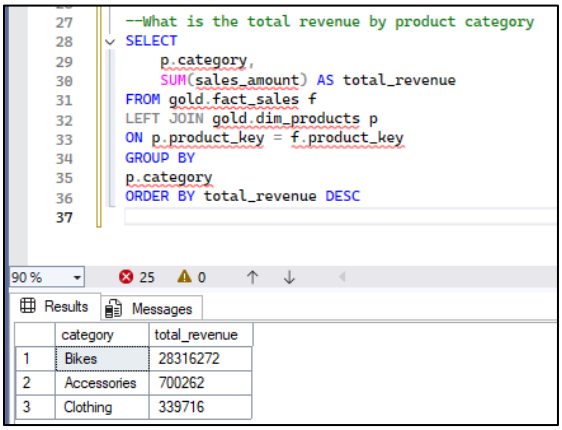
Results | Messages

	country	total_items_sold
1	United States	20481
2	Australia	13346
3	Canada	7630
4	United Kingdom	6910
5	Germany	5626
6	France	5559
7	n/a	871

Total Revenue by Product Category

Viewing the total items sold by product category is similar the previous analysis using a LEFT JOIN. The following query was written to provide the desired view of total revenue by product category.

Total Revenue by Product Category



```
--What is the total revenue by product category
SELECT
    p.category,
    SUM(sales_amount) AS total_revenue
FROM gold.fact_sales f
LEFT JOIN gold.dim_products p
ON p.product_key = f.product_key
GROUP BY
p.category
ORDER BY total_revenue DESC
```

90 % | 25 | 0

Results | Messages

	category	total_revenue
1	Bikes	28316272
2	Accessories	700262
3	Clothing	339716

h. Ranking Analysis

Ranking analysis involves ordering the values of dimensions by measure to identify top N and bottom N within the domain of interest.

Example:

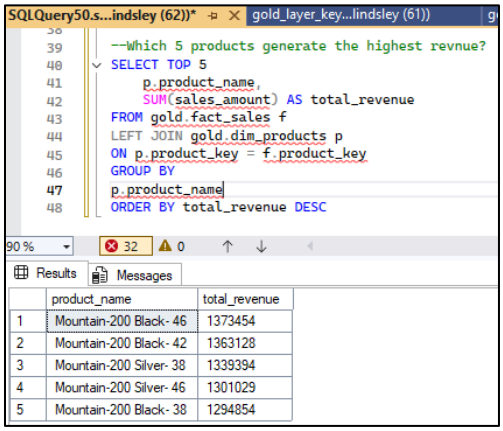
- Top 5 Products by Revenue Generation
- Rank Countries by Total Sales and view the top 3
- Bottom 3 Customers by Total Orders

Which Top 5 Products Generate the Highest / Lowest Revenue

In this case, the measure is revenue which is captured as 'sales_amount' in the gold.fact_sales' table. The dimension is products which is captured as the 'product_name' in the gold.dim_products' table. Both tables are related as the 'product_key' is found in both tables. To limit the results of the query only requires including 'TOP 5' immediately after 'SELECT' in the query.

In a similar manner, identifying the top 5 products with the lowest sales requires sorting in ascending order by total_revenue or just leaving it blank as SQL will order in ascending order by default.

Identifying Top 5 Products by Total Sales / Revenue



SQLQuery50.s...indsley (62))* | gold_layer_key...lindsley (61))

```
--Which 5 products generate the highest revnue?
SELECT TOP 5
    p.product_name,
    SUM(sales_amount) AS total_revenue
FROM gold.fact_sales f
LEFT JOIN gold.dim_products p
ON p.product_key = f.product_key
GROUP BY
p.product_name
ORDER BY total_revenue DESC
```

90 % | 32 | 0

Results | Messages

	product_name	total_revenue
1	Mountain-200 Black- 46	1373454
2	Mountain-200 Black- 42	1363128
3	Mountain-200 Silver- 38	1339394
4	Mountain-200 Silver- 46	1301029
5	Mountain-200 Black- 38	1294854

Assigning Rank identifiers to Enable More Complex Analysis

Assigning a rank ID to each record in the results enables more complex analysis. Doing so involves using WINDOW functions and generating a subquery.

```sql
--Assigning Rank Identifiers to View Products by Specific Rank Range
SELECT *
FROM (
    SELECT
        p.product_name,
        SUM(sales_amount) AS total_revenue,
        ROW_NUMBER() OVER (ORDER BY SUM(sales_amount) DESC) AS rank_products

    FROM gold.fact_sales f
    LEFT JOIN gold.dim_products p
    ON p.product_key = f.product_key
    GROUP BY p.product_name
    )t
WHERE rank_products <= 5
```

90 % 46 0

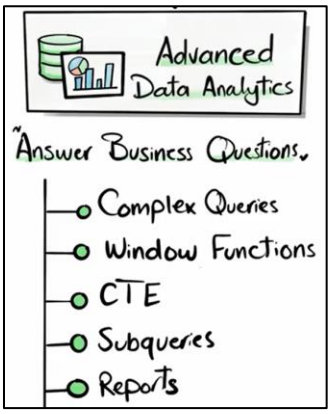
Results Messages

	product_name	total_revenue	rank_products
1	Mountain-200 Black- 46	1373454	1
2	Mountain-200 Black- 42	1363128	2
3	Mountain-200 Silver- 38	1339394	3
4	Mountain-200 Silver- 46	1301029	4
5	Mountain-200 Black- 38	1294854	5

10. SQL Advanced Analytics

a. Overview

The purpose of this stage of the project is to use and understand SQL syntax that enables answering advanced business questions. I will leverage the SQL skills I've learned throughout up to this point to further my knowledge of analytics in SQL. I learned to apply the following in this portion of the project:



b. Change Over Time Analysis

This is a technique for analyzing how a measure evolves over time and helps track trends and identify seasonality in data. Doing so involves aggregating a measure by a date dimension to understand how they change over time.

For example:

- Total Sales By Year
- Average Cost by Month

Analyzing Sales Performance Over Time

Bringing together the order dates, sum of sales, customer count, and item quantities ordered into a single query allows me to see to sales performance over time by year. The results reveal that sales increased from 2010 to 2013, then had a sharp drop in 2014. Additionally, you can see that total customers and total products sold took a nose dive between 2013 and 2014.

Sales Performance Over Time

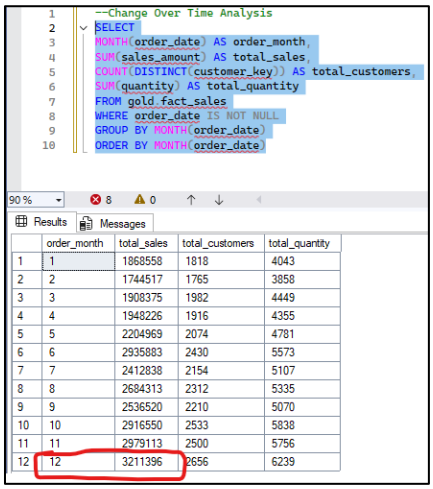
```sql
--Change Over Time Analysis
SELECT
YEAR(order_date) AS order_year,
SUM(sales_amount) AS total_sales,
COUNT(DISTINCT(customer_key)) AS total_customers,
SUM(quantity) AS total_quantity
FROM gold.fact_sales
WHERE order_date IS NOT NULL
GROUP BY YEAR(order_date)
ORDER BY YEAR(order_date)
```

Results | Messages

	order_year	total_sales	total_customers	total_quantity
1	2010	43419	14	14
2	2011	7075088	2216	2216
3	2012	5842231	3255	3397
4	2013	16344878	17427	52807
5	2014	45642	834	1970

The sales performance data can be further broken-down by month to understand the seasonality of sales and provide greater clarity / insight on the data. By adjusting the query to include the month and the year, it becomes obvious that we were wrong to assume that sales took a nose-dive in 2014 as the data only captures the first month of 2014 rather than a full year. So….it would only make sense to compare the sales of the first month of 2014 with the sales of the same month for previous years.

Additionally, it becomes apparent that December and February are the best and worst sales months respectively. This makes sense as December tends to be a month in which purchasing activity increases in consumer goods due to Christmas and other holidays during this period. Sales typically decrease sharply after Christmas…so…it makes sense that they are lowest in in February.



```sql
--Change Over Time Analysis
SELECT
MONTH(order_date) AS order_month,
SUM(sales_amount) AS total_sales,
COUNT(DISTINCT(customer_key)) AS total_customers,
SUM(quantity) AS total_quantity
FROM gold.fact_sales
WHERE order_date IS NOT NULL
GROUP BY MONTH(order_date)
ORDER BY MONTH(order_date)
```

90 % 8 0

Results Messages

	order_month	total_sales	total_customers	total_quantity
1	1	1868558	1818	4043
2	2	1744517	1765	3858
3	3	1908375	1982	4449
4	4	1948226	1916	4355
5	5	2204969	2074	4781
6	6	2935883	2430	5573
7	7	2412838	2154	5107
8	8	2684313	2312	5335
9	9	2536520	2210	5070
10	10	2916550	2533	5838
11	11	2979113	2500	5756
12	12	3211396	2656	6239

c. Cumulative Analysis

Cumulative analysis involves aggregating data progressively over time to understand how the values grow or decline. Aggregate WINDOW functions are used for this in SQL

Example:

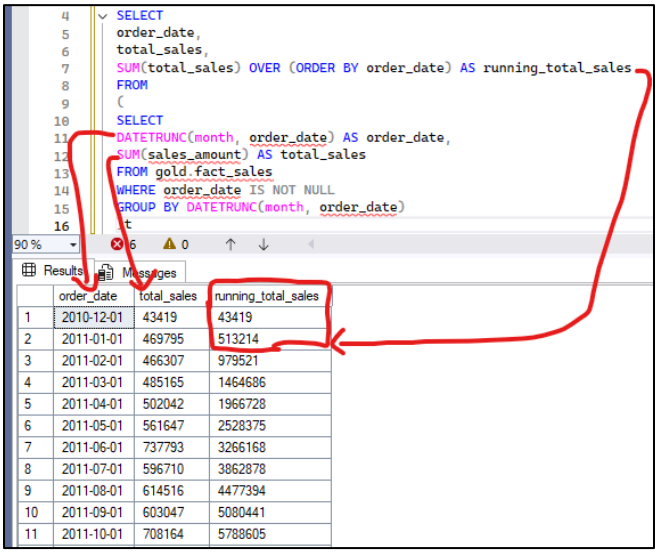
- Running total sales
- Moving average sales

Running Total Sales

Creating the following query with subsquery generates a view of total sales for each month and a view of the running total sales by month.

Running Total Sales (cont'd)

The image below shows the query and the results that show cumulative running total sales. The total sales for Dec 2010 (2010-12-01) was 43,419 and the running total sales for that month is the same as the total sales generated as it is the first month in which sales occurred. The following month, Jan 2011 (2011-01-01) has a monthly total sales of 469,795 and a running total sales of 513,214. This running total sales for Jan 2011 is the sum of sales for Dec 2010 and Jan 2011. The running total sum calculation continues for all remaining rows to provide visibility into the change in total sales over time.



```
SELECT
order_date,
total_sales,
SUM(total_sales) OVER (ORDER BY order_date) AS running_total_sales
FROM
(
SELECT
DATETRUNC(month, order_date) AS order_date,
SUM(sales_amount) AS total_sales
FROM gold.fact_sales
WHERE order_date IS NOT NULL
GROUP BY DATETRUNC(month, order_date)
)t
```

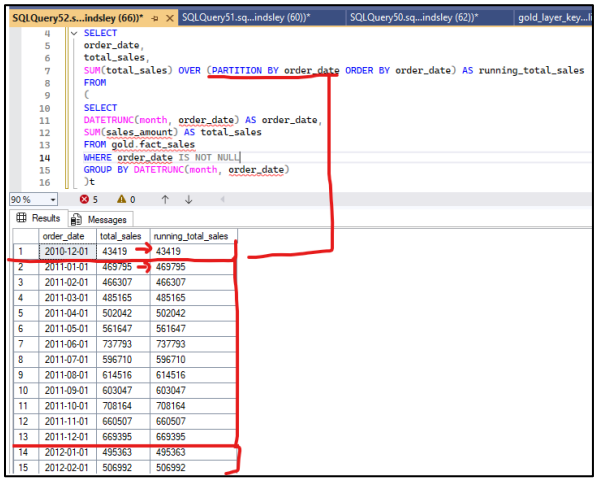
90 % | 6 | 0

Results | Messages

	order_date	total_sales	running_total_sales
1	2010-12-01	43419	43419
2	2011-01-01	469795	513214
3	2011-02-01	466307	979521
4	2011-03-01	485165	1464686
5	2011-04-01	502042	1966728
6	2011-05-01	561647	2528375
7	2011-06-01	737793	3266168
8	2011-07-01	596710	3862878
9	2011-08-01	614516	4477394
10	2011-09-01	603047	5080441
11	2011-10-01	708164	5788605

The query used to view running total sales provides continuous running total sales from the first month and year of sales through the current month and year of sales. This query can be refined by using PARTITION BY to view the running total by month for each year. Doing so will reset the cumulative sum of sales each at the beginning of each year and only run through the 12 months of that year. So, analysis can be done to compare cumulative sales over a 12 month period for each year. The image below depicts the query adjustment and the cumulative results produced through the PARTITION BY.

Cumulative Sales Partitioned by Year



```
SELECT
order_date,
total_sales,
SUM(total_sales) OVER (PARTITION BY order_date ORDER BY order_date) AS running_total_sales
FROM
(
SELECT
DATETRUNC(month, order_date) AS order_date,
SUM(sales_amount) AS total_sales
FROM gold.fact_sales
WHERE order_date IS NOT NULL
GROUP BY DATETRUNC(month, order_date)
)t
```

90 % | 5 | 0

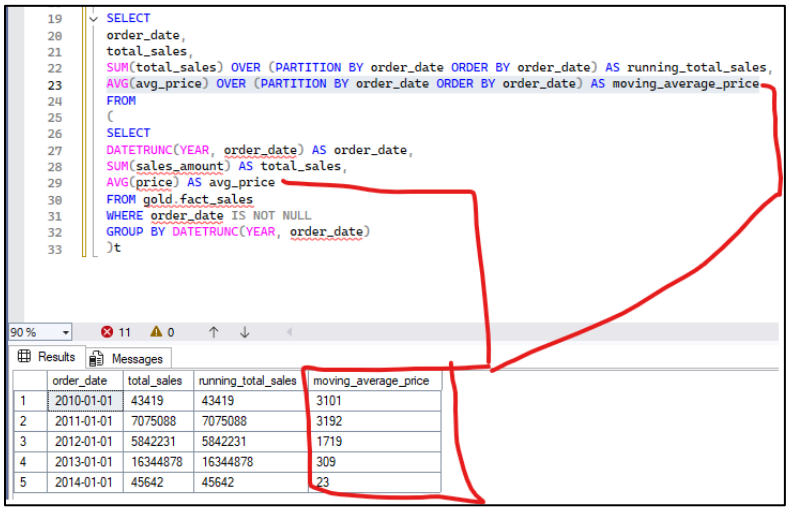
Results | Messages

	order_date	total_sales	running_total_sales
1	2010-12-01	43419	43419
2	2011-01-01	469795	469795
3	2011-02-01	466307	466307
4	2011-03-01	485165	485165
5	2011-04-01	502042	502042
6	2011-05-01	561647	561647
7	2011-06-01	737793	737793
8	2011-07-01	596710	596710
9	2011-08-01	614516	614516
10	2011-09-01	603047	603047
11	2011-10-01	708164	708164
12	2011-11-01	660507	660507
13	2011-12-01	669395	669395
14	2012-01-01	495363	495363
15	2012-02-01	506992	506992

Moving Average of Price

Adding the following to the query provides a view of the moving average price over time partitioned by year. The newly added column ‘avg_price’ must also be declared / created in the subquery for this to work.

```
AVG(avg_price) OVER (PARTITION BY order_date ORDER BY order_date) AS moving_average_price’
```



```
SELECT
order_date,
total_sales,
SUM(total_sales) OVER (PARTITION BY order_date ORDER BY order_date) AS running_total_sales,
AVG(avg_price) OVER (PARTITION BY order_date ORDER BY order_date) AS moving_average_price
FROM
(
SELECT
DATETRUNC(YEAR, order_date) AS order_date,
SUM(sales_amount) AS total_sales,
AVG(price) AS avg_price
FROM gold.fact_sales
WHERE order_date IS NOT NULL
GROUP BY DATETRUNC(YEAR, order_date)
)t
```

90 % 11 0

Results Messages

	order_date	total_sales	running_total_sales	moving_average_price
1	2010-01-01	43419	43419	3101
2	2011-01-01	7075088	7075088	3192
3	2012-01-01	5842231	5842231	1719
4	2013-01-01	16344878	16344878	309
5	2014-01-01	45642	45642	23

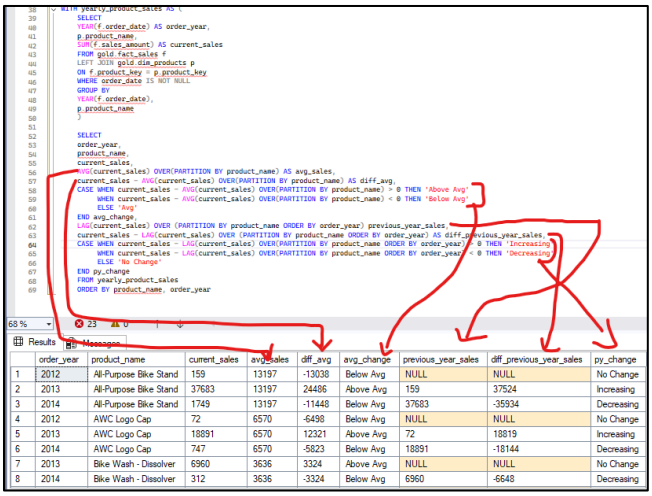
d. Performance Analysis

Performance analysis is the comparison of current values to a target value. This is helpful when evaluating the performance of a specific category / dimension. The formula here is:
Current[Measure] – Target[Measure]

Examples:

- Current Sales – Average Sales
- Current Year sales – Previous Year Sales ← Year over Year Analysis
- Current Sales – Lowest Sales

Window Functions such as MAX, MIN, LEAD, and LAG are typically used here in conjunction with SUM, and AVG. Additionally, CASE WHEN is also needed for this.



```
WITH yearly_product_sales AS (
    SELECT
    YEAR(f_order_date) AS order_year,
    p_product_name,
    SUM(f_sales_amount) AS current_sales
    FROM gold.fact_sales f
    LEFT JOIN gold.dim_products p
    ON f_product_key = p_product_key
    WHERE order_date IS NOT NULL
    GROUP BY
    YEAR(f_order_date),
    p_product_name
    )

    SELECT
    order_year,
    product_name,
    current_sales,
    AVG(current_sales) OVER(PARTITION BY product_name) AS avg_sales,
    current_sales - AVG(current_sales) OVER(PARTITION BY product_name) AS diff_avg,
    CASE WHEN current_sales - AVG(current_sales) OVER(PARTITION BY product_name) > 0 THEN 'Above Avg'
        WHEN current_sales - AVG(current_sales) OVER(PARTITION BY product_name) < 0 THEN 'Below Avg'
        ELSE 'Avg'
    END avg_change,
    LAG(current_sales) OVER (PARTITION BY product_name ORDER BY order_year) previous_year_sales,
    current_sales - LAG(current_sales) OVER (PARTITION BY product_name ORDER BY order_year) AS diff_previous_year_sales,
    CASE WHEN current_sales - LAG(current_sales) OVER(PARTITION BY product_name ORDER BY order_year) > 0 THEN 'Increasing'
        WHEN current_sales - LAG(current_sales) OVER(PARTITION BY product_name ORDER BY order_year) < 0 THEN 'Decreasing'
        ELSE 'No Change'
    END py_change
    FROM yearly_product_sales
    ORDER BY product_name, order_year
```

68 % 23 0

Results Messages

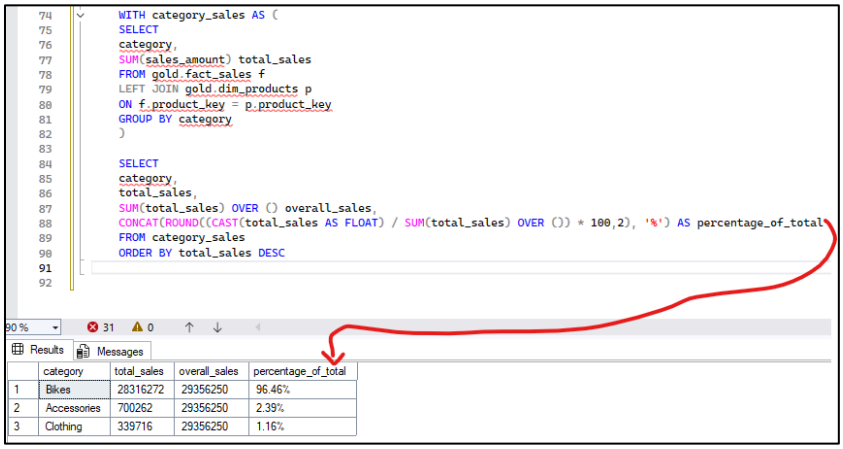
	order_year	product_name	current_sales	avg_sales	diff_avg	avg_change	previous_year_sales	diff_previous_year_sales	py_change
1	2012	All-Purpose Bike Stand	159	13197	-13038	Below Avg	NULL	NULL	No Change
2	2013	All-Purpose Bike Stand	37683	13197	24486	Above Avg	159	37524	Increasing
3	2014	All-Purpose Bike Stand	1749	13197	-11448	Below Avg	37683	-35934	Decreasing
4	2012	AWC Logo Cap	72	6570	-6498	Below Avg	NULL	NULL	No Change
5	2013	AWC Logo Cap	18891	6570	12321	Above Avg	72	18819	Increasing
6	2014	AWC Logo Cap	747	6570	-5823	Below Avg	18891	-18144	Decreasing
7	2013	Bike Wash - Dissolver	6960	3636	3324	Above Avg	NULL	NULL	No Change
8	2014	Bike Wash - Dissolver	312	3636	-3324	Below Avg	6960	-6648	Decreasing

e. Part-to-Whole Analysis

This type of analysis is used to analyze how an individual part is performing compared to the overall. For example: What is the total sales for the Bikes category as a percentage of all product categories? Think…pie chart. This is helpful for understanding the contribution of each category to the whole.

The formula here is simple:
([Measure] / Total[Measure]) * 100 by Dimension

The screenshot below shows how I wrote a query that provides total sales for each product category, the overall sales for all product categories, and the percentage of sales that each product category contributes to total sales.



```
WITH category_sales AS (
SELECT
category,
SUM(sales_amount) total_sales
FROM gold.fact_sales f
LEFT JOIN gold.dim_products p
ON f.product_key = p.product_key
GROUP BY category
)

SELECT
category,
total_sales,
SUM(total_sales) OVER () overall_sales,
CONCAT(ROUND((CAST(total_sales AS FLOAT) / SUM(total_sales) OVER ()) * 100,2), '%') AS percentage_of_total
FROM category_sales
ORDER BY total_sales DESC
```

Results | Messages

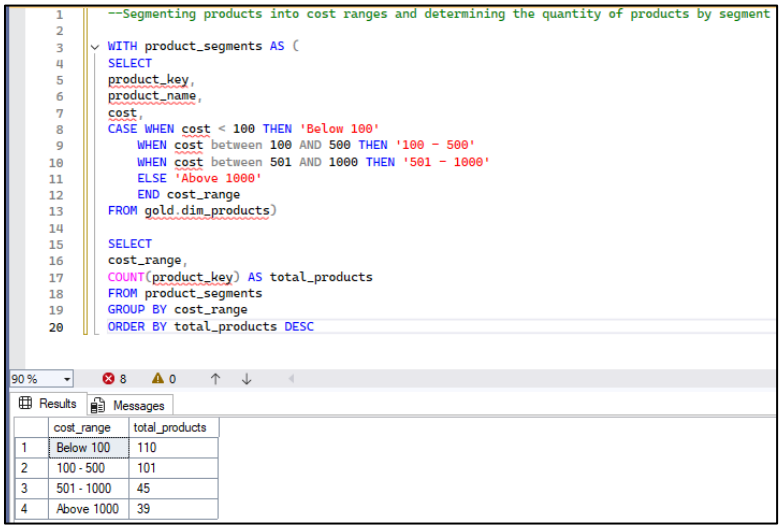
	category	total_sales	overall_sales	percentage_of_total
1	Bikes	28316272	29356250	96.46%
2	Accessories	700262	29356250	2.39%
3	Clothing	339716	29356250	1.16%

f. Data Segmentation

Data segmentation groups data into defined ranges to reveal relationships between measures. For example, total products by sales range or customers by age. For dimension items, you first create a measure (such as total products) and then segment it by a value range, effectively turning the measure into a dimension. I'll share a more detailed example built on the gold-layer data next.

Segmenting products into cost ranges and determining the quantity of products by segment
In this case I wrote the base query to categorize products into cost range categories using a CASE WHEN. I then built a CTE using WITH to generate a query on the base query to obtain the total number of products for each cost range category.

Segmenting Costs into Cost Range Categories



```
--Segmenting products into cost ranges and determining the quantity of products by segment

WITH product_segments AS (
SELECT
product_key,
product_name,
cost,
CASE WHEN cost < 100 THEN 'Below 100'
    WHEN cost between 100 AND 500 THEN '100 - 500'
    WHEN cost between 501 AND 1000 THEN '501 - 1000'
    ELSE 'Above 1000'
    END cost_range
FROM gold.dim_products)

SELECT
cost_range,
COUNT(product_key) AS total_products
FROM product_segments
GROUP BY cost_range
ORDER BY total_products DESC
```

Results | Messages

	cost_range	total_products
1	Below 100	110
2	100 - 500	101
3	501 - 1000	45
4	Above 1000	39

f. Data Segmentation (cont'd)

Segmenting Customers by Lifespan and Total Spending

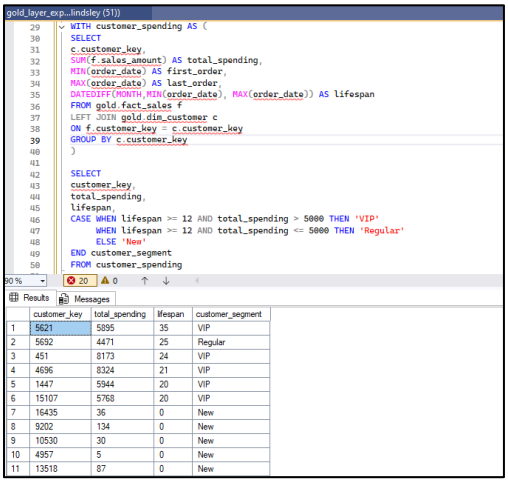
In this example, I am segmenting customers into three group according to the number of months between their first and most recent order, and the total sales from all orders. The segmentation is:

- **VIP**: Custs with >= 12 months of history and spending more than 5,000
- **Regular**: Customeres with >=12 months of history but spending <= 5,000
- **New**: Customers with a lifespan less than 12 months

The first step in accomplishing this is a CTE that has the following (see image below for initial results):

- A left join on gold.fact_sales and gold.dim_customer tables on 'customer_key'
- Determining the customer lifespan using DATEDIFF, MIN, and MAX on 'order_date'
- Capturing customer sum of sales
- Assigning segmentation to VIP, Regular, or New using CASE WHEN on the lifespan criteria and the total sales criteria defined above.

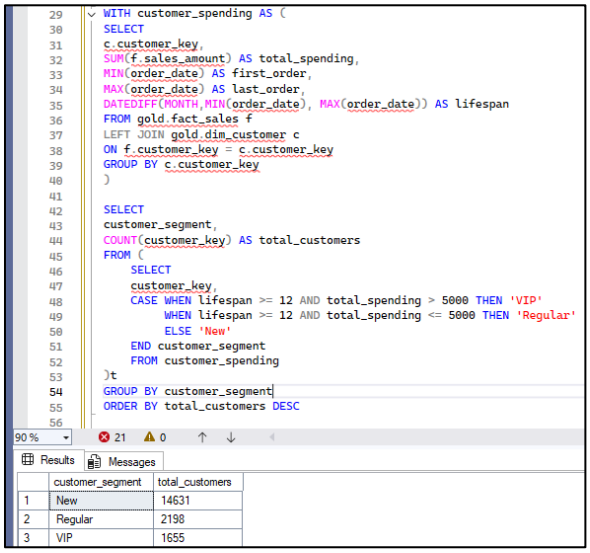
Query to and Results for Initial Segmentation



```
WITH customer_spending AS (
SELECT
c.customer_key,
SUM(f.sales_amount) AS total_spending,
MIN(order_date) AS first_order,
MAX(order_date) AS last_order,
DATEDIFF(MONTH,MIN(order_date), MAX(order_date)) AS lifespan
FROM gold.fact_sales f
LEFT JOIN gold.dim_customer c
ON f.customer_key = c.customer_key
GROUP BY c.customer_key
)

SELECT
customer_key,
total_spending,
lifespan,
CASE WHEN lifespan >= 12 AND total_spending > 5000 THEN 'VIP'
     WHEN lifespan >= 12 AND total_spending <= 5000 THEN 'Regular'
     ELSE 'New'
END customer_segment
FROM customer_spending
```

	customer_key	total_spending	lifespan	customer_segment
1	5621	5895	35	VIP
2	5692	4471	25	Regular
3	451	8173	24	VIP
4	4696	8324	21	VIP
5	1447	5944	20	VIP
6	15107	5768	20	VIP
7	16435	36	0	New
8	9202	134	0	New
9	10530	30	0	New
10	4957	5	0	New
11	13518	87	0	New

The next step is to find the total number of customers for each category. To accomplish this, I revised the CTE by adding a subquery, adding 'COUNT(customer_key) AS total_customers' and grouping by customer_segment to capture total customers by customer segment. This allows me to see the total quantity of customers by the categories I defined in the CTE.



```
WITH customer_spending AS (
SELECT
c.customer_key,
SUM(f.sales_amount) AS total_spending,
MIN(order_date) AS first_order,
MAX(order_date) AS last_order,
DATEDIFF(MONTH,MIN(order_date), MAX(order_date)) AS lifespan
FROM gold.fact_sales f
LEFT JOIN gold.dim_customer c
ON f.customer_key = c.customer_key
GROUP BY c.customer_key
)

SELECT
customer_segment,
COUNT(customer_key) AS total_customers
FROM (
    SELECT
    customer_key,
    CASE WHEN lifespan >= 12 AND total_spending > 5000 THEN 'VIP'
         WHEN lifespan >= 12 AND total_spending <= 5000 THEN 'Regular'
         ELSE 'New'
    END customer_segment
    FROM customer_spending
)t
GROUP BY customer_segment
ORDER BY total_customers DESC
```

	customer_segment	total_customers
1	New	14631
2	Regular	2198
3	VIP	1655

g. Build Customers Report

The purpose of this report is to consolidate key customer metrics and behaviors within the data sets in the data warehouse.

It should:

- Gather essential fields such as names, ages, and transaction details.
- Segments customers into categories (VIP, Regular, New) and age groups.
- Aggregate customer-level metrics:
  - total orders
  - total sales
  - total quantity purchased
  - total products
  - lifespan (in months)
- Calculate valuable KPIs:
  - recency (months since last order)
  - average order value
  - average monthly spend

Creating this report requires a well-crafted approach for developing complexed queries. The first thing to do here is to build the base data by selecting the data from the database. In this case, I will start with the fact table, then join the dimension tables and filter only on the columns needed. This base data will serve as the foundation for the remainder of the effort. I will build starting with a CTE since there are multiple steps in the overall effort (deriving new columns, executing some transformations).

Retrieving Core Columns from the Tables:

Here I am retrieving the core columns, conducting quick eval to determine whether or not any additional transformations are required to prepare them for aggregation.

Retrieve Core Columns

```
/*-----------------------------------------------------------------
1) Build the Base Query:  Retrieve core columns from tables
-----------------------------------------------------------------*/
SELECT
f.order_number,
f.product_key,
f.order_date,
f.sales_amount,
f.quantity,
c.customer_key,
c.customer_number,
c.first_name,
c.last_name,
c.birthdate
FROM gold.fact_sales f
LEFT JOIN gold.dim_customer c
ON c.customer_key = f.customer_key
WHERE order_date IS NOT NULL
```

99 % 15 0

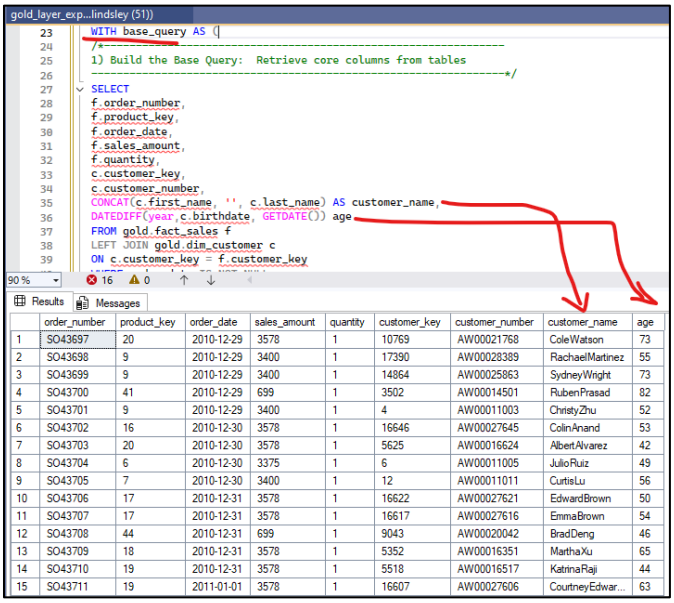
Results Messages

	order_number	product_key	order_date	sales_amount	quantity	customer_key	customer_number	first_name	last_name	birthdate
1	SO43697	20	2010-12-29	3578	1	10769	AW00021768	Cole	Watson	1952-02-19
2	SO43698	9	2010-12-29	3400	1	17390	AW00028389	Rachael	Martinez	1970-06-17
3	SO43699	9	2010-12-29	3400	1	14864	AW00025863	Sydney	Wright	1952-06-01
4	SO43700	41	2010-12-29	699	1	3502	AW00014501	Ruben	Prasad	1943-11-10
5	SO43701	9	2010-12-29	3400	1	4	AW00011003	Christy	Zhu	1973-08-14
6	SO43702	16	2010-12-30	3578	1	16646	AW00027645	Colin	Anand	1972-08-14
7	SO43703	20	2010-12-30	3578	1	5625	AW00016624	Albert	Alvarez	1983-07-24
8	SO43704	6	2010-12-30	3375	1	6	AW00011005	Julio	Ruiz	1976-08-01
9	SO43705	7	2010-12-30	3400	1	12	AW00011011	Curtis	Lu	1969-05-03
10	SO43706	17	2010-12-31	3578	1	16622	AW00027621	Edward	Brown	1975-01-23
11	SO43707	17	2010-12-31	3578	1	16617	AW00027616	Emma	Brown	1971-06-10

Upon evaluating the core columns, I determine that the first and last names should be concatenated so they are in a single column that reads ‘First Name Last Name’.

I also need to capture the age for each customer. I use DATEDIFF for this. I also add syntax to begin creating the CTE.

Name and Age Transformations (with CTE added but not executed)



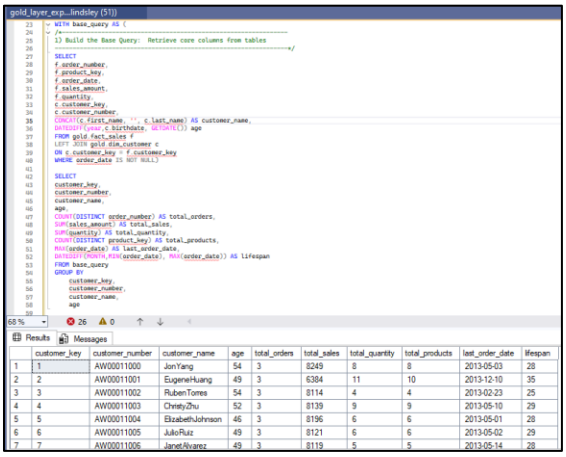
```
WITH base_query AS (
/*-----------------------------------------------------------------------------
1) Build the Base Query:  Retrieve core columns from tables
-----------------------------------------------------------------------------*/
SELECT
f_order_number,
f_product_key,
f_order_date,
f_sales_amount,
f_quantity,
c_customer_key,
c_customer_number,
CONCAT(c_first_name, ' ', c_last_name) AS customer_name,
DATEDIFF(year,c_birthdate, GETDATE()) age
FROM gold.fact_sales f
LEFT JOIN gold.dim_customer c
ON c_customer_key = f_customer_key
```

	order_number	product_key	order_date	sales_amount	quantity	customer_key	customer_number	customer_name	age
1	SO43697	20	2010-12-29	3578	1	10769	AW00021768	ColeWatson	73
2	SO43698	9	2010-12-29	3400	1	17390	AW00028389	RachaelMartinez	55
3	SO43699	9	2010-12-29	3400	1	14864	AW00025863	SydneyWright	73
4	SO43700	41	2010-12-29	699	1	3502	AW00014501	RubenPrasad	82
5	SO43701	9	2010-12-29	3400	1	4	AW00011003	ChristyZhu	52
6	SO43702	16	2010-12-30	3578	1	16646	AW00027645	ColinAnand	53
7	SO43703	20	2010-12-30	3578	1	5625	AW00016624	AlbertAlvarez	42
8	SO43704	6	2010-12-30	3375	1	6	AW00011005	JulioRuiz	49
9	SO43705	7	2010-12-30	3400	1	12	AW00011011	CurtisLu	56
10	SO43706	17	2010-12-31	3578	1	16622	AW00027621	EdwardBrown	50
11	SO43707	17	2010-12-31	3578	1	16617	AW00027616	EmmaBrown	54
12	SO43708	44	2010-12-31	699	1	9043	AW00020042	BradDeng	46
13	SO43709	18	2010-12-31	3578	1	5352	AW00016351	MarthaXu	65
14	SO43710	19	2010-12-31	3578	1	5518	AW00016517	KatrinaRaji	44
15	SO43711	19	2011-01-01	3578	1	16607	AW00027606	CourtneyEdwar...	63

With the intermediate results prepared in the CTE, I am now able to begin aggregating the results to needed for the required report.

The first aggregation identifies the total orders, the total sales, the total quantity of products ordered, and the different products ordered. It also sets the foundations for identifying customers by lifespan according to age by adding a DATEDIFF on MIN and MAX order_date.

Initial Aggregations



```
WITH base_query AS (
/*-----------------------------------------------------------------------------
1) Build the Base Query:  Retrieve core columns from tables
-----------------------------------------------------------------------------*/
SELECT
f_order_number,
f_product_key,
f_order_date,
f_sales_amount,
f_quantity,
c_customer_key,
c_customer_number,
CONCAT(c_first_name, ' ', c_last_name) AS customer_name,
DATEDIFF(year,c_birthdate, GETDATE()) age
FROM gold.fact_sales f
LEFT JOIN gold.dim_customer c
ON c_customer_key = f_customer_key
WHERE order_date IS NOT NULL)

SELECT
customer_key,
customer_number,
customer_name,
age,
COUNT(DISTINCT order_number) AS total_orders,
SUM(sales_amount) AS total_sales,
SUM(quantity) AS total_quantity,
COUNT(DISTINCT product_key) AS total_products,
MAX(order_date) AS last_order_date,
DATEDIFF(MONTH,MIN(order_date), MAX(order_date)) AS lifespan
FROM base_query
GROUP BY
    customer_key,
    customer_number,
    customer_name,
    age
```

	customer_key	customer_number	customer_name	age	total_orders	total_sales	total_quantity	total_products	last_order_date	lifespan
1	1	AW00011000	JonYang	54	3	8249	8	8	2013-05-03	28
2	2	AW00011001	EugeneHuang	49	3	6384	11	10	2013-12-10	35
3	3	AW00011002	RubenTorres	54	3	8114	4	4	2013-02-23	25
4	4	AW00011003	ChristyZhu	52	3	8139	9	9	2013-05-10	29
5	5	AW00011004	ElizabethJohnson	46	3	8196	6	6	2013-05-01	28
6	6	AW00011005	JulioRuiz	49	3	8121	6	6	2013-05-02	29
7	7	AW00011006	JanetAlvarez	49	3	8119	5	5	2013-05-14	28

At this point, the query is getting fairly large. So, descriptions of it’s evolution will follow. The final script will be committed to Git for reference.

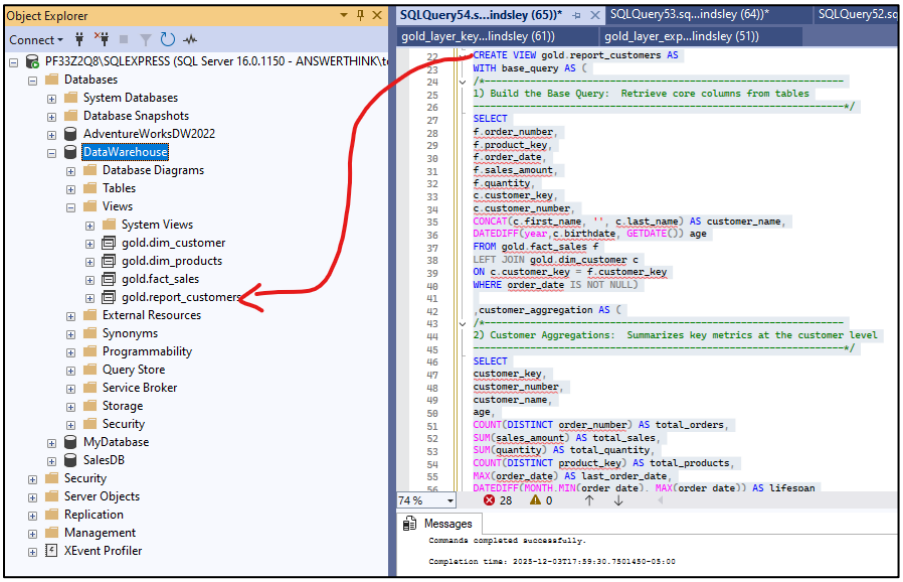
The aggregations are now ready to prepare the final result with the customer segmentation by age and customer category (VIP, Regular, and New) as well as required KPIs.

At this point I built in the following by creating an additional CTE:

- Customer Category (VIP, Regular, and New)
- Customer Age Category
- Recency (months since last order)
- Average order value
- Average monthly spend

With the query complete, I then created a view in the Gold layer to enable (fictional) data analysts to access the data for analysis and reporting.

Creation of Customer View for Analysis & Reporting



h. Build Products report

The purpose of this report is to consolidate key product metrics and behaviors.

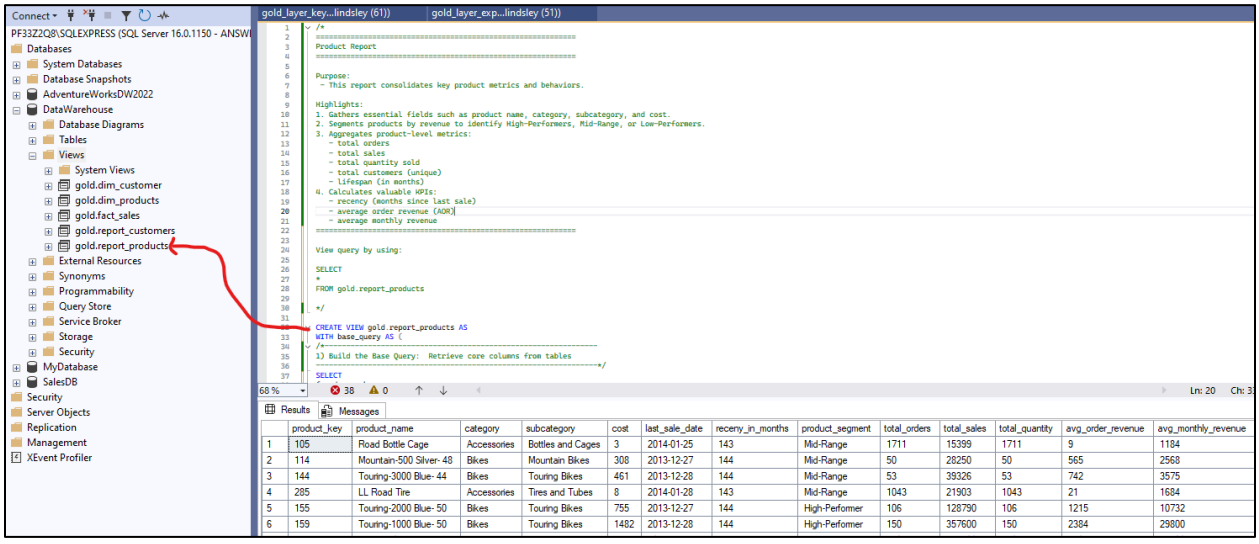
It should:
1. Gather essential fields such as product name, category, subcategory, and cost.
2. Segment products by revenue to identify High-Performers, Mid-Range, or Low-Performers.
3. Aggregate product-level metrics:
   - total orders
   - total sales
   - total quantity sold
   - total customers (unique)
   - lifespan (in months)
4. Calculate valuable KPIs:
   - recency (months since last sale)
   - average order revenue (AOR)
   - average monthly revenue

The SQL query was prepared using a CTE and CREATE VIEW. It was added to the Gold layer views.

This was a tough one as I had to build it on my own rather than following along step-by-step. It worked for the most part, but I continued to run into errors that I had trouble resolving. The instructor did provide an example of how it should look. So, after several attempts…and getting it 80% correct, I referred to his example to finalize my query. It turns out that portions of my query were calling columns that had not been declared in preceding CTE queries. Was able to correct for those and get the desired output.

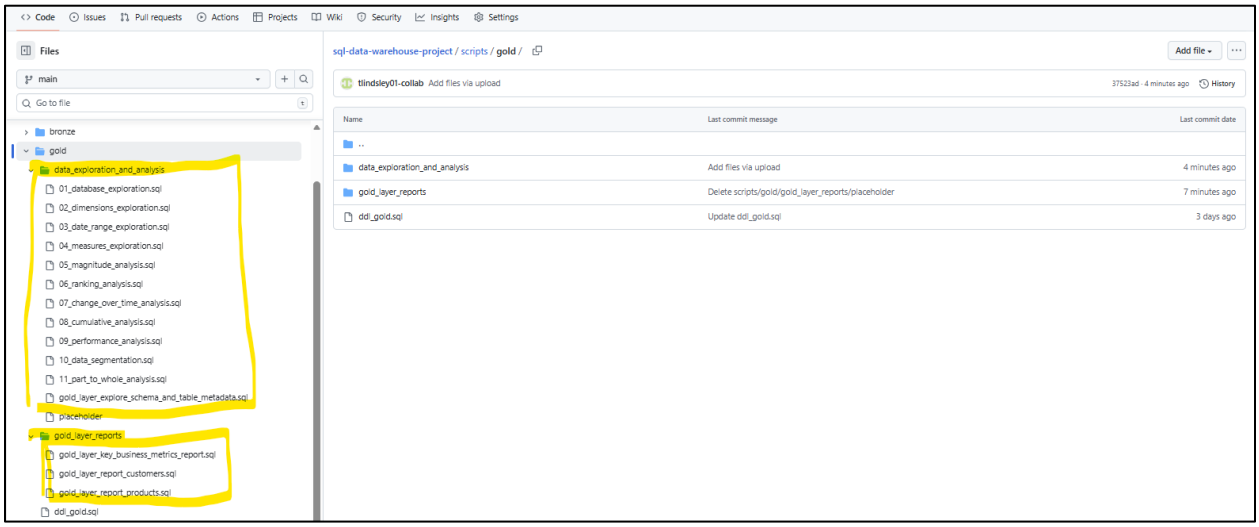
The below screenshot depicts the report location, the query (only a portion is visible) and the output (a portion of the output only.

'CREATE VIEW' Query for Products Report in Gold Layer



i. Document Work in Git

All sql code developed in the Exploratory and Advanced SQL analytics was loaded to my Git repo. It includes all scripts for exploratory analysis, advanced analysis, and the reports / views created for Customers and Products in the Gold Layer.



END