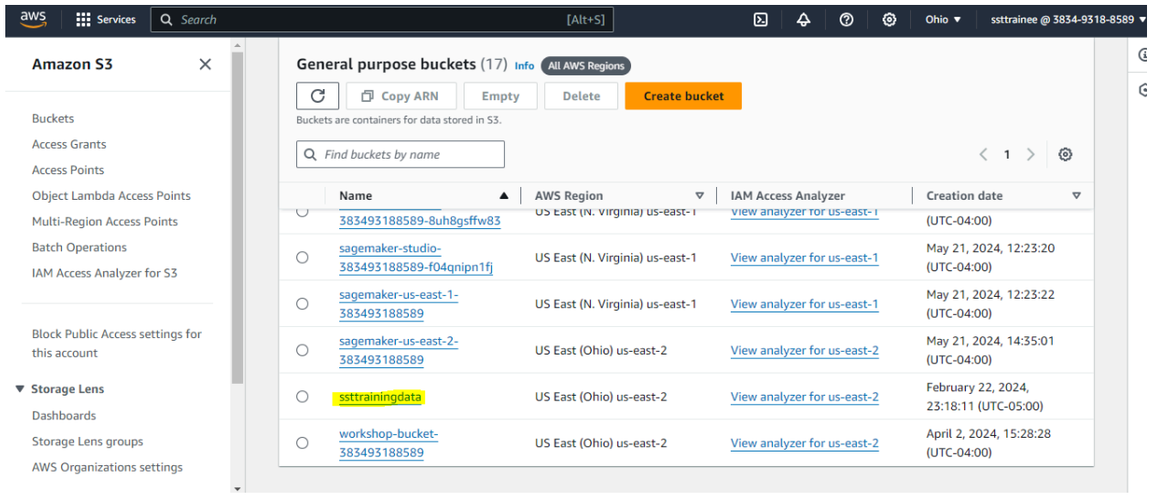
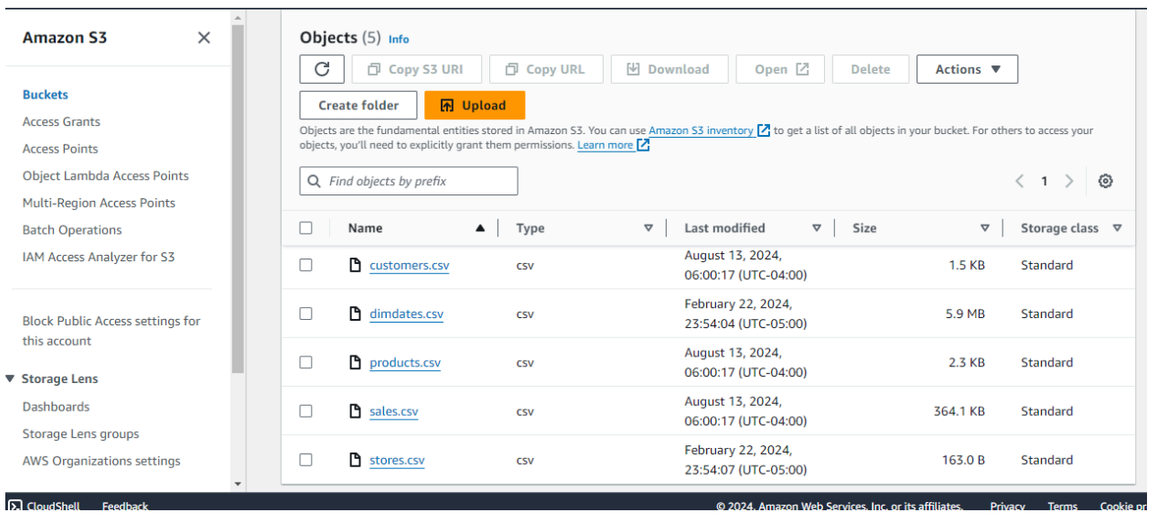
Automated ETL Pipeline for Data Ingestion and SCD2 Implementation Using AWS and Snowflake and PowerBI

Name: **HIMA SREE CHALASANI**
Role: Data Analyst Trainee

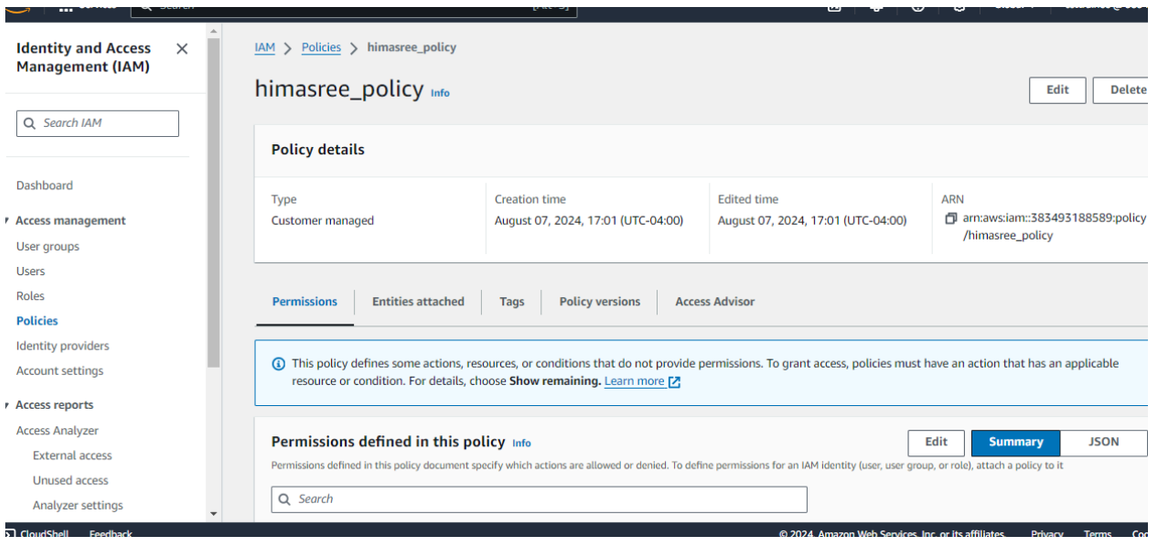
This is the S3 bucket named “ssttrainingdata” that I am accessing for my data needs. This bucket contains the data files that I use in my Snowflake environment.





I created a policy that allows Snowflake to connect to the bucket securely. This policy is associated with an AWS IAM role, which grants the necessary permissions for Snowflake to read the files in the bucket.

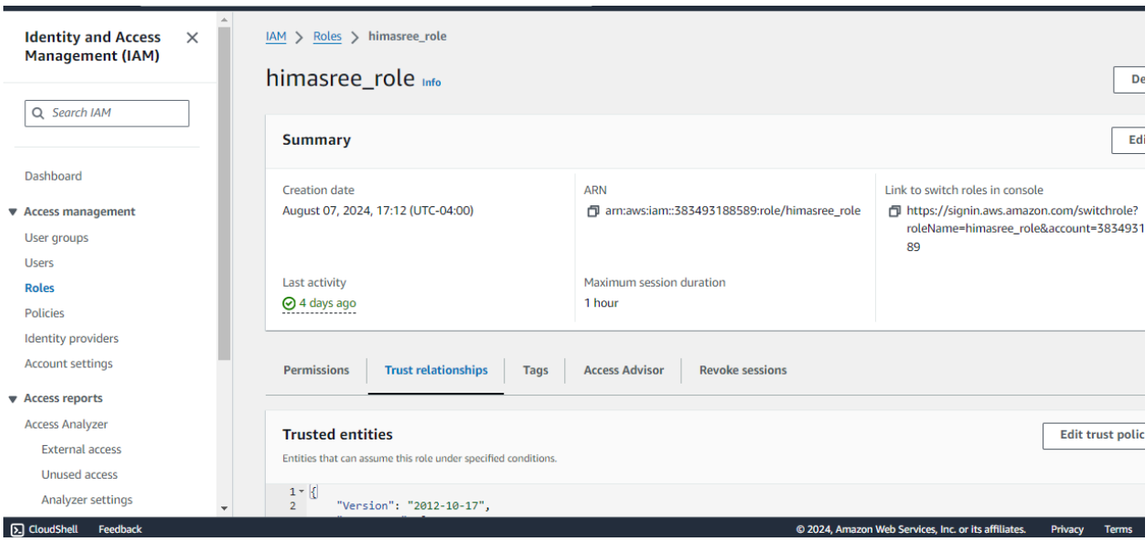
Policy name : himasree_policy

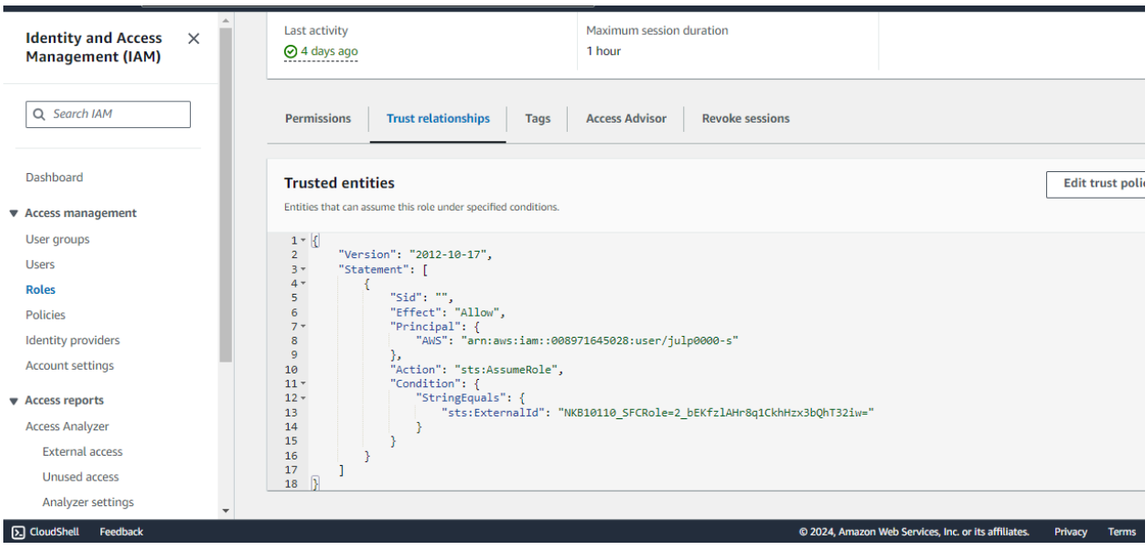


Create an IAM Role: I created an AWS Identity and Access Management (IAM) role specifically for Snowflake to access the S3 bucket. This role defines the permissions needed to interact with the bucket.

Attach Policy to the Role: I attached a policy to this IAM role that grants read access to the S3 bucket ssttrainingdata. The policy specifies which actions Snowflake can perform, such as listing and reading files.

Role: himasree_role





TASK : Creating Tables and Loading Data from Amazon AWS S3 Bucket to Snowflake Data Warehouse Staging Tables

- Created a Snowflake Warehouse, Database, Schema

```
create warehouse HIMA_WH;
create database HIMA_DB;
create schema HIMA_DB.STAGING;
```

- Created Storage Integration

Storage integration allows Snowflake to securely access your S3 bucket without directly exposing AWS credentials.

```
create or replace storage integration Hima_OBJ
 type = external_stage
 storage_provider = s3
  enabled = true
  storage_aws_role_arn = 'arn:aws:iam::383493188589:role/himasree_role'
 storage_allowed_locations = ('s3://ssttrainingdata/');

desc integration Hima_OBJ;
```

- Create File Format

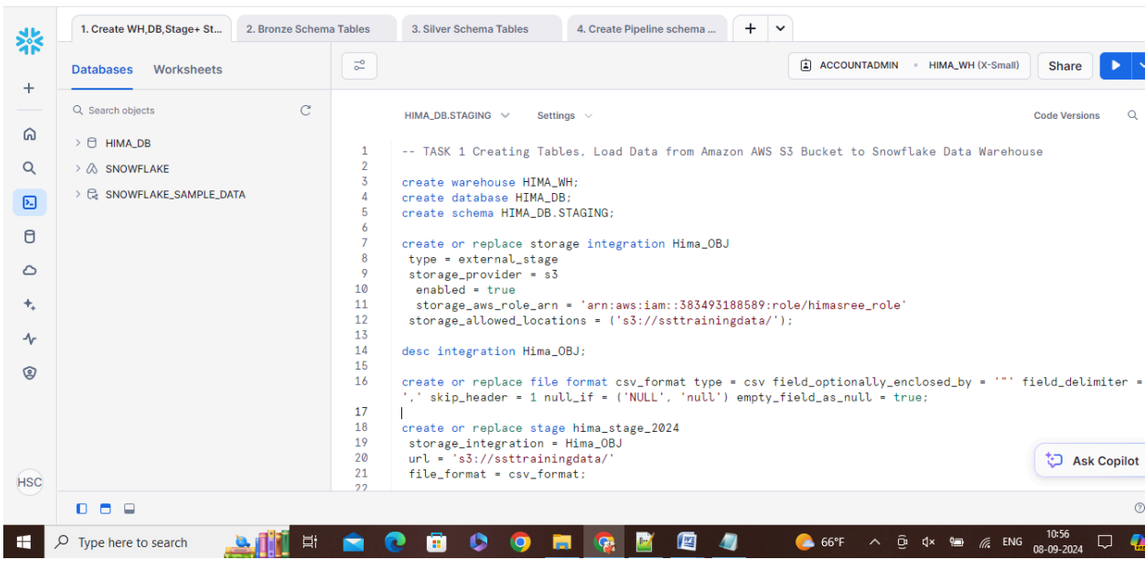
File formats define how Snowflake reads data files from S3. This setup will handle CSV files.

```
create or replace file format csv_format type = csv field_optionally_enclosed_by = ""
field_delimiter = ',' skip_header = 1 null_if = ('NULL', 'null') empty_field_as_null = true;
```

- Set Up the External Stage in Snowflake

Created an External Stage: An external stage points to your S3 bucket and tells Snowflake where to find the data. To do this, configure the external stage with the S3 bucket URL, file format, and the necessary credentials.

```
create or replace stage hima_stage_2024
 storage_integration = Hima_OBJ
 url = 's3://ssttrainingdata/'
 file_format = csv_format;
```

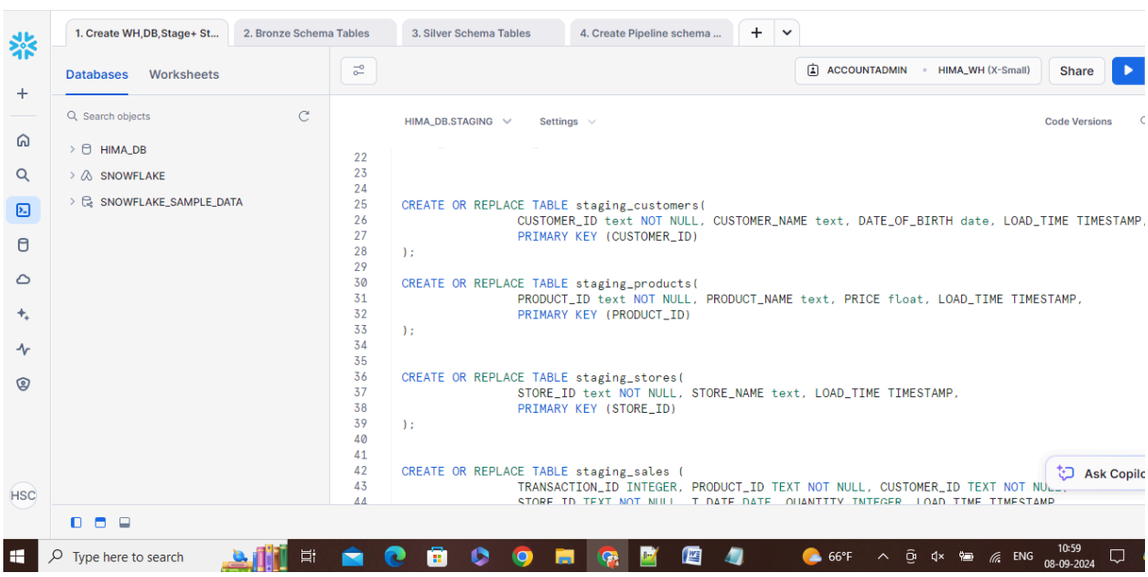


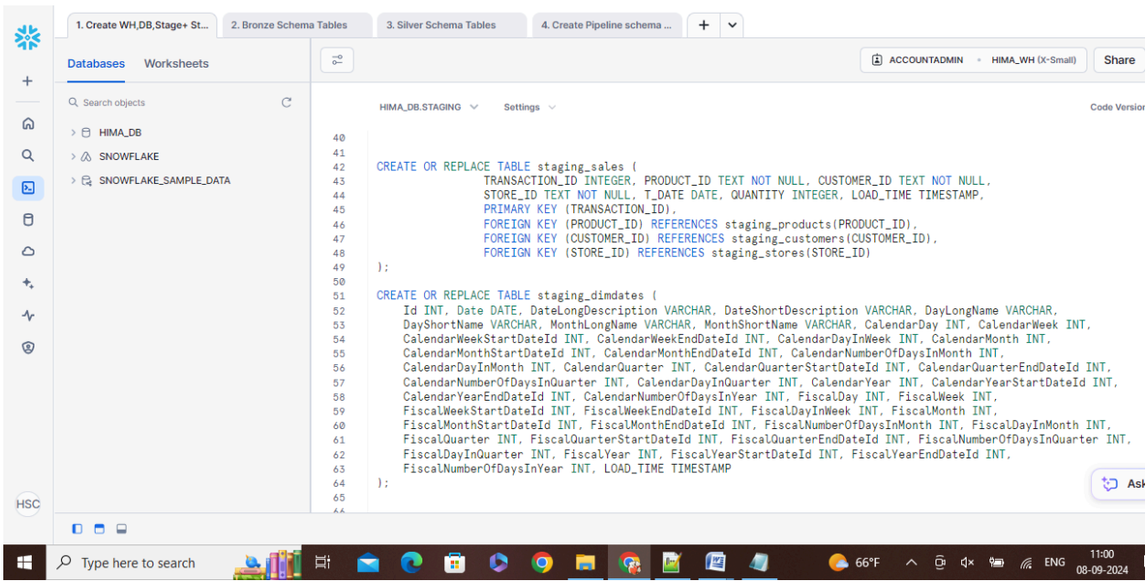
STAGING LAYER

- Created Staging Tables in Snowflake

```
CREATE OR REPLACE TABLE staging_customers(
        CUSTOMER_ID text NOT NULL, CUSTOMER_NAME text, DATE_OF_BIRTH
date, LOAD_TIME TIMESTAMP,
        PRIMARY KEY (CUSTOMER_ID)
);


CREATE OR REPLACE TABLE staging_products(
        PRODUCT_ID text NOT NULL, PRODUCT_NAME text, PRICE float,
LOAD_TIME TIMESTAMP,
        PRIMARY KEY (PRODUCT_ID)
);

CREATE OR REPLACE TABLE staging_stores(
        STORE_ID text NOT NULL, STORE_NAME text, LOAD_TIME TIMESTAMP,
        PRIMARY KEY (STORE_ID)
);


CREATE OR REPLACE TABLE staging_sales (
        TRANSACTION_ID INTEGER, PRODUCT_ID TEXT NOT NULL,
CUSTOMER_ID TEXT NOT NULL,
        STORE_ID TEXT NOT NULL, T_DATE DATE, QUANTITY INTEGER,
LOAD_TIME TIMESTAMP,
        PRIMARY KEY (TRANSACTION_ID),
        FOREIGN KEY (PRODUCT_ID) REFERENCES staging_products(PRODUCT_ID),
```

```
        FOREIGN KEY (CUSTOMER_ID) REFERENCES
staging_customers(CUSTOMER_ID),
        FOREIGN KEY (STORE_ID) REFERENCES staging_stores(STORE_ID)
);


CREATE OR REPLACE TABLE staging_dimdates (
    Id INT, Date DATE, DateLongDescription VARCHAR, DateShortDescription VARCHAR,
DayLongName VARCHAR,
    DayShortName VARCHAR, MonthLongName VARCHAR, MonthShortName VARCHAR,
CalendarDay INT, CalendarWeek INT,
    CalendarWeekStartDateId INT, CalendarWeekEndDateId INT, CalendarDayInWeek INT,
CalendarMonth INT,
    CalendarMonthStartDateId INT, CalendarMonthEndDateId INT,
CalendarNumberOfDaysInMonth INT,
    CalendarDayInMonth INT, CalendarQuarter INT, CalendarQuarterStartDateId INT,
CalendarQuarterEndDateId INT,
    CalendarNumberOfDaysInQuarter INT, CalendarDayInQuarter INT, CalendarYear INT,
CalendarYearStartDateId INT,
    CalendarYearEndDateId INT, CalendarNumberOfDaysInYear INT, FiscalDay INT,
FiscalWeek INT,
    FiscalWeekStartDateId INT, FiscalWeekEndDateId INT, FiscalDayInWeek INT, FiscalMonth
INT,
    FiscalMonthStartDateId INT, FiscalMonthEndDateId INT, FiscalNumberOfDaysInMonth
INT, FiscalDayInMonth INT,
    FiscalQuarter INT, FiscalQuarterStartDateId INT, FiscalQuarterEndDateId INT,
FiscalNumberOfDaysInQuarter INT,
    FiscalDayInQuarter INT, FiscalYear INT, FiscalYearStartDateId INT, FiscalYearEndDateId
INT,
    FiscalNumberOfDaysInYear INT, LOAD_TIME TIMESTAMP
);
```





- Created a view for Staging tables

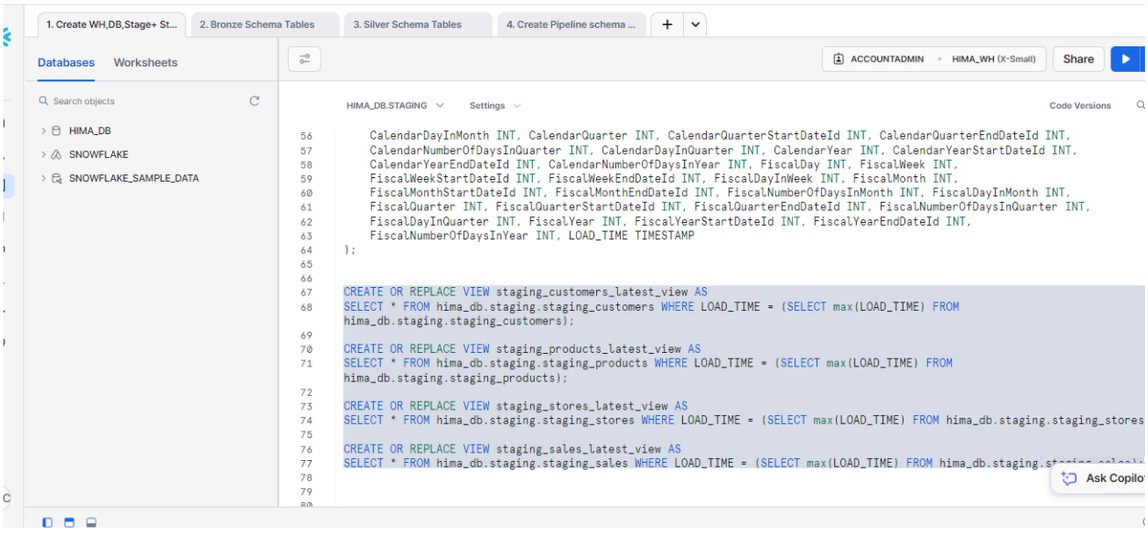
Create views for each of your staging tables to capture the latest data based on the LOAD_TIME column.

CREATE OR REPLACE VIEW staging_customers_latest_view AS
SELECT * FROM hima_db.staging.staging_customers WHERE LOAD_TIME = (SELECT max(LOAD_TIME) FROM hima_db.staging.staging_customers);

CREATE OR REPLACE VIEW staging_products_latest_view AS
SELECT * FROM hima_db.staging.staging_products WHERE LOAD_TIME = (SELECT max(LOAD_TIME) FROM hima_db.staging.staging_products);

CREATE OR REPLACE VIEW staging_stores_latest_view AS
SELECT * FROM hima_db.staging.staging_stores WHERE LOAD_TIME = (SELECT max(LOAD_TIME) FROM hima_db.staging.staging_stores);

CREATE OR REPLACE VIEW staging_sales_latest_view AS
SELECT * FROM hima_db.staging.staging_sales WHERE LOAD_TIME = (SELECT max(LOAD_TIME) FROM hima_db.staging.staging_sales);



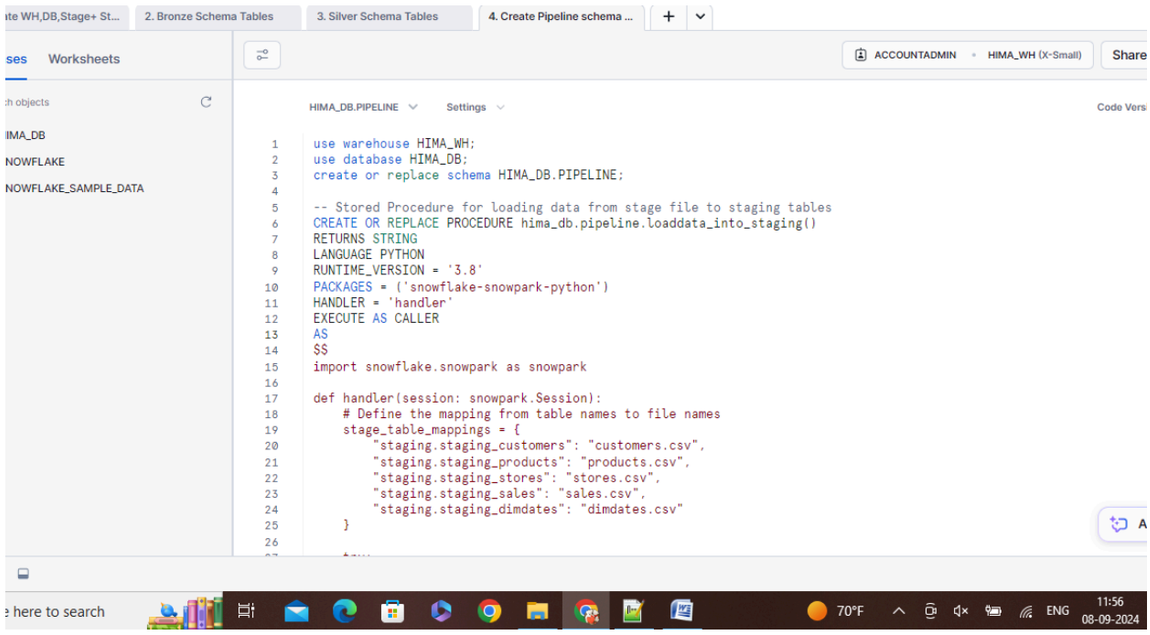
- Load Data from S3 Bucket into Staging Tables

Below is a Python-based Snowflake stored procedure that loads data from the S3 bucket into the specified staging tables:

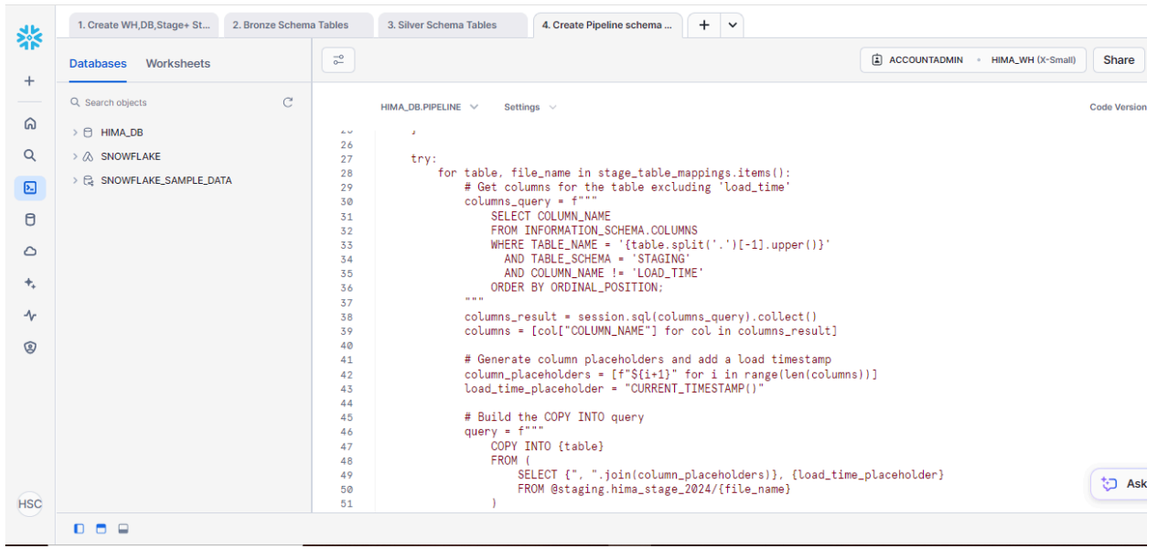
```
-- Stored Procedure for loading data from stage file to staging tables
CREATE OR REPLACE PROCEDURE hima_db.pipeline.loaddata_into_staging()
RETURNS STRING
LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('snowflake-snowpark-python')
HANDLER = 'handler'
EXECUTE AS CALLER
AS
$$
import snowflake.snowpark as snowpark

def handler(session: snowpark.Session):
    # Define the mapping from table names to file names
    stage_table_mappings = {
        "staging.staging_customers": "customers.csv",
        "staging.staging_products": "products.csv",
        "staging.staging_stores": "stores.csv",
        "staging.staging_sales": "sales.csv",
        "staging.staging_dimdates": "dimdates.csv"
    }

    try:
        for table, file_name in stage_table_mappings.items():
            # Get columns for the table excluding 'load_time'
```

```
        columns_query = f"""
            SELECT COLUMN_NAME
            FROM INFORMATION_SCHEMA.COLUMNS
            WHERE TABLE_NAME = '{table.split('.')[-1].upper()}'
              AND TABLE_SCHEMA = 'STAGING'
              AND COLUMN_NAME != 'LOAD_TIME'
            ORDER BY ORDINAL_POSITION;
        """
        columns_result = session.sql(columns_query).collect()
        columns = [col["COLUMN_NAME"] for col in columns_result]

        # Generate column placeholders and add a load timestamp
        column_placeholders = [f"${i+1}" for i in range(len(columns))]
        load_time_placeholder = "CURRENT_TIMESTAMP()"

        # Build the COPY INTO query
        query = f"""
            COPY INTO {table}
            FROM (
                SELECT {", ".join(column_placeholders)}, {load_time_placeholder}
                FROM @staging.hima_stage_2024/{file_name}
            )
            FILE_FORMAT = (TYPE = 'CSV' FIELD_OPTIONALLY_ENCLOSED_BY = "")
            ON_ERROR = 'CONTINUE'
            FORCE = TRUE;
        """
        # Execute the query
        session.sql(query).collect()

    return "Data loaded successfully "

  except Exception as e:
    return f"Failed: {str(e)}"
$$;
```

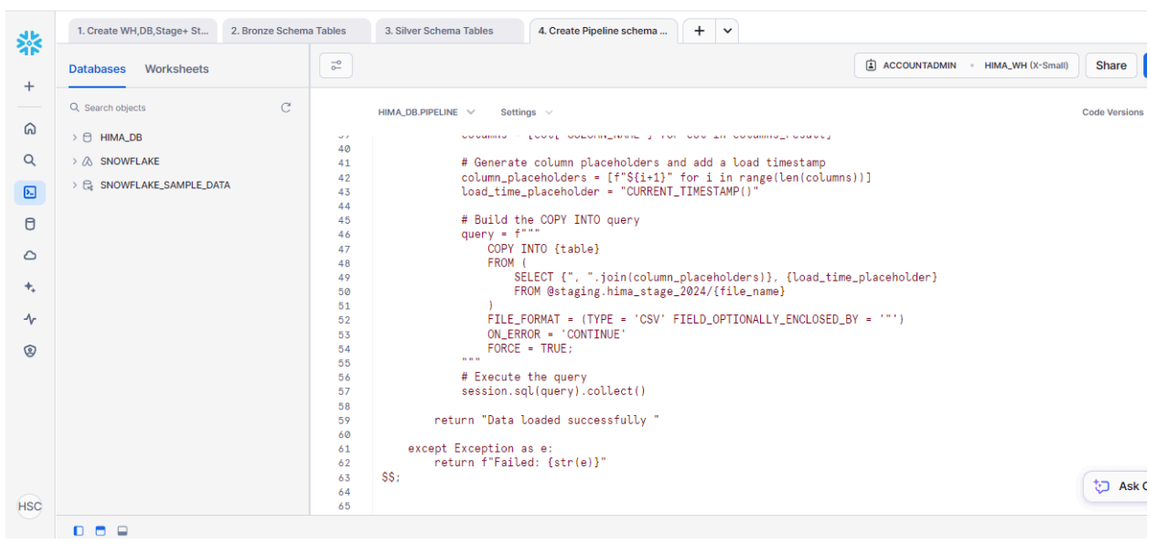
This stored procedure pipeline.loaddata_into_staging() created in the pipeline schema is designed to automate the loading of data from the S3 bucket into the Snowflake staging tables. It handles the file-to-table mappings, error logging, and execution summary, making it an efficient approach to manage your data loading tasks.



```
use warehouse HIMA_WH;
use database HIMA_DB;
create or replace schema HIMA_DB.PIPELINE;

-- Stored Procedure for loading data from stage file to staging tables
CREATE OR REPLACE PROCEDURE hima_db.pipeline.loaddata_into_staging()
RETURNS STRING
LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('snowflake-snowpark-python')
HANDLER = 'handler'
EXECUTE AS CALLER
AS
$$
import snowflake.snowpark as snowpark

def handler(session: snowpark.Session):
    # Define the mapping from table names to file names
    stage_table_mappings = {
        "staging.staging_customers": "customers.csv",
        "staging.staging_products": "products.csv",
        "staging.staging_stores": "stores.csv",
        "staging.staging_sales": "sales.sales.csv",
        "staging.staging_dimdates": "dimdates.csv"
    }
```



```
    try:
        for table, file_name in stage_table_mappings.items():
            # Get columns for the table excluding 'load_time'
            columns_query = f"""
                SELECT COLUMN_NAME
                FROM INFORMATION_SCHEMA.COLUMNS
                WHERE TABLE_NAME = '{table.split('.')[-1].upper()}'
                  AND TABLE_SCHEMA = 'STAGING'
                  AND COLUMN_NAME != 'LOAD_TIME'
                ORDER BY ORDINAL_POSITION;
            """
            columns_result = session.sql(columns_query).collect()
            columns = [col["COLUMN_NAME"] for col in columns_result]

            # Generate column placeholders and add a load timestamp
            column_placeholders = [f"${i+1}" for i in range(len(columns))]
            load_time_placeholder = "CURRENT_TIMESTAMP()"

            # Build the COPY INTO query
            query = f"""
                COPY INTO {table}
                FROM (
                    SELECT {", ".join(column_placeholders)}, {load_time_placeholder}
                    FROM @staging.hima_stage_2024/{file_name}
                )
```

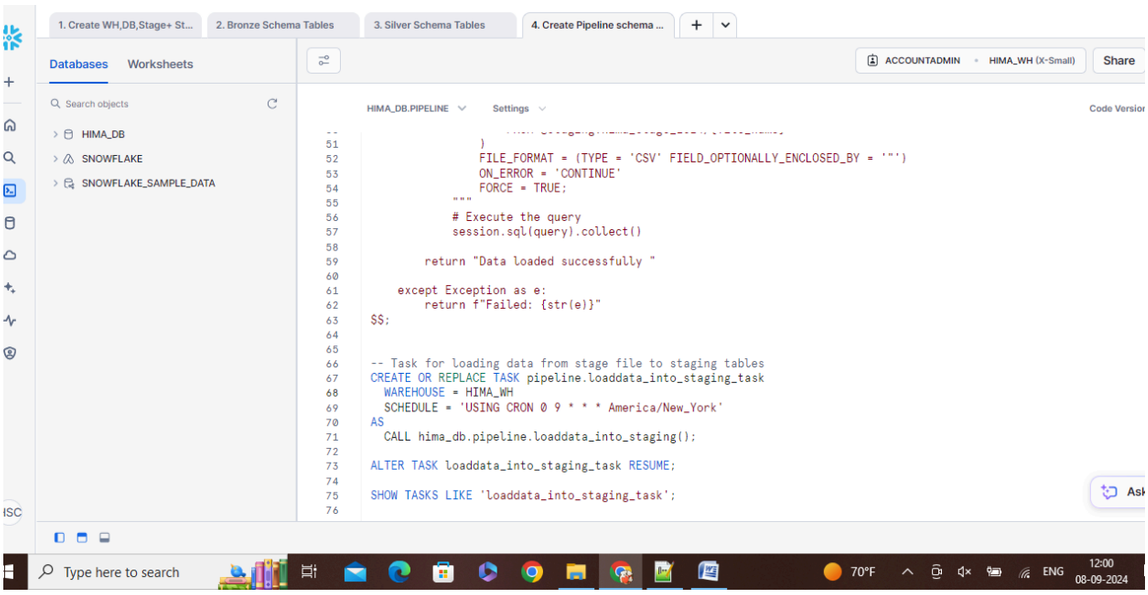


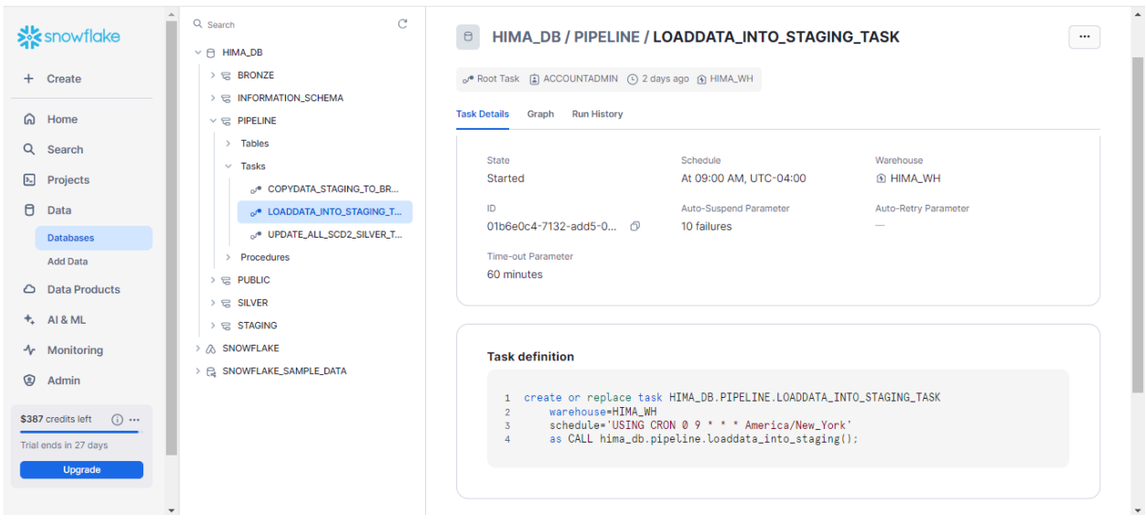
```
            # Generate column placeholders and add a load timestamp
            column_placeholders = [f"${i+1}" for i in range(len(columns))]
            load_time_placeholder = "CURRENT_TIMESTAMP()"

            # Build the COPY INTO query
            query = f"""
                COPY INTO {table}
                FROM (
                    SELECT {", ".join(column_placeholders)}, {load_time_placeholder}
                    FROM @staging.hima_stage_2024/{file_name}
                )
                FILE_FORMAT = (TYPE = 'CSV' FIELD_OPTIONALLY_ENCLOSED_BY = '"')
                ON_ERROR = 'CONTINUE'
                FORCE = TRUE;
            """
            # Execute the query
            session.sql(query).collect()

        return "Data loaded successfully "

    except Exception as e:
        return f"Failed: {str(e)}"
$$;
```

- Create a Scheduled Task

Task that runs the pipeline.loaddata_staging() stored procedure every day at 9 AM

```
-- Task for loading data from stage file to staging tables
CREATE OR REPLACE TASK pipeline.loaddata_into_staging_task
  WAREHOUSE = HIMA_WH
  SCHEDULE = 'USING CRON 0 9 * * * America/New_York'
AS
  CALL hima_db.pipeline.loaddata_into_staging();

ALTER TASK loaddata_into_staging_task RESUME;
```





Process to Maintain Only the Latest Data in Bronze Tables

TASK: Ensure that each bronze table in Snowflake contains only the most recent data by removing outdated records and inserting the latest data from the staging tables. Additionally, impose the following rules:

1. All primary keys of the tables should not be null or empty.

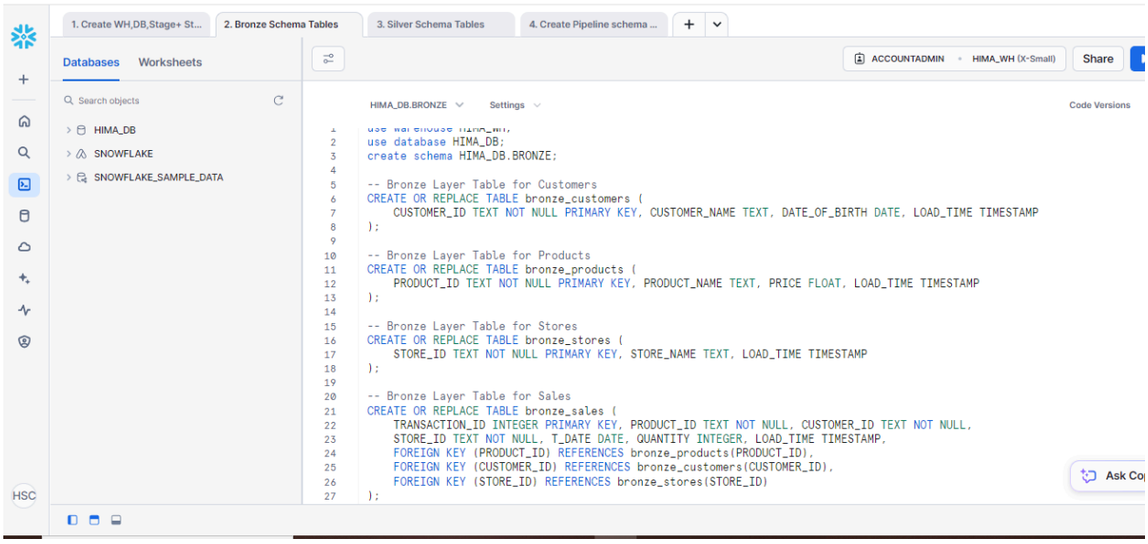
2. If a customer is deleted, their related sales should also be deleted.

3. If a store is deleted, related products should also be deleted.

4. Only data that qualifies these rules should be present in the bronze layer (today's data).

- Created Bronze Schema and Tables

```
use warehouse HIMA_WH;
use database HIMA_DB;
create schema HIMA_DB.BRONZE;

-- Bronze Layer Table for Customers
CREATE OR REPLACE TABLE bronze_customers (
    CUSTOMER_ID TEXT NOT NULL PRIMARY KEY, CUSTOMER_NAME TEXT,
DATE_OF_BIRTH DATE, LOAD_TIME TIMESTAMP
);

-- Bronze Layer Table for Products
CREATE OR REPLACE TABLE bronze_products (
    PRODUCT_ID TEXT NOT NULL PRIMARY KEY, PRODUCT_NAME TEXT, PRICE
FLOAT, LOAD_TIME TIMESTAMP
);

-- Bronze Layer Table for Stores
CREATE OR REPLACE TABLE bronze_stores (
    STORE_ID TEXT NOT NULL PRIMARY KEY, STORE_NAME TEXT, LOAD_TIME
TIMESTAMP
);

-- Bronze Layer Table for Sales
CREATE OR REPLACE TABLE bronze_sales (
    TRANSACTION_ID INTEGER PRIMARY KEY, PRODUCT_ID TEXT NOT NULL,
CUSTOMER_ID TEXT NOT NULL,
    STORE_ID TEXT NOT NULL, T_DATE DATE, QUANTITY INTEGER, LOAD_TIME
TIMESTAMP,
    FOREIGN KEY (PRODUCT_ID) REFERENCES bronze_products(PRODUCT_ID),
    FOREIGN KEY (CUSTOMER_ID) REFERENCES bronze_customers(CUSTOMER_ID),
    FOREIGN KEY (STORE_ID) REFERENCES bronze_stores(STORE_ID)
);
```



- Details of the DATA_QUALITY_RULES Table:

To ensure that data loaded into the bronze layer tables adheres to certain quality and consistency rules, you created a table named DATA_QUALITY_RULES in the pipeline schema. This table stores the rules that dictate how data should be validated and processed before it is inserted into the bronze tables.

-- Table for Logging Failures of data quality rules:

```
CREATE OR REPLACE TABLE HIMA_DB.PIPELINE.RULE_EXECUTION_LOGS (
    RULE_ID INT,
    RULE_TYPE VARCHAR(255),
    RULE_DESCRIPTION VARCHAR(255),
    STATUS VARCHAR(255),
    ERROR_MESSAGE VARCHAR(255),
    EXECUTION_TIME TIMESTAMP,
    FOREIGN KEY (RULE_ID) REFERENCES
HIMA_DB.PIPELINE.DATA_QUALITY_RULES(RULE_ID)
);
```

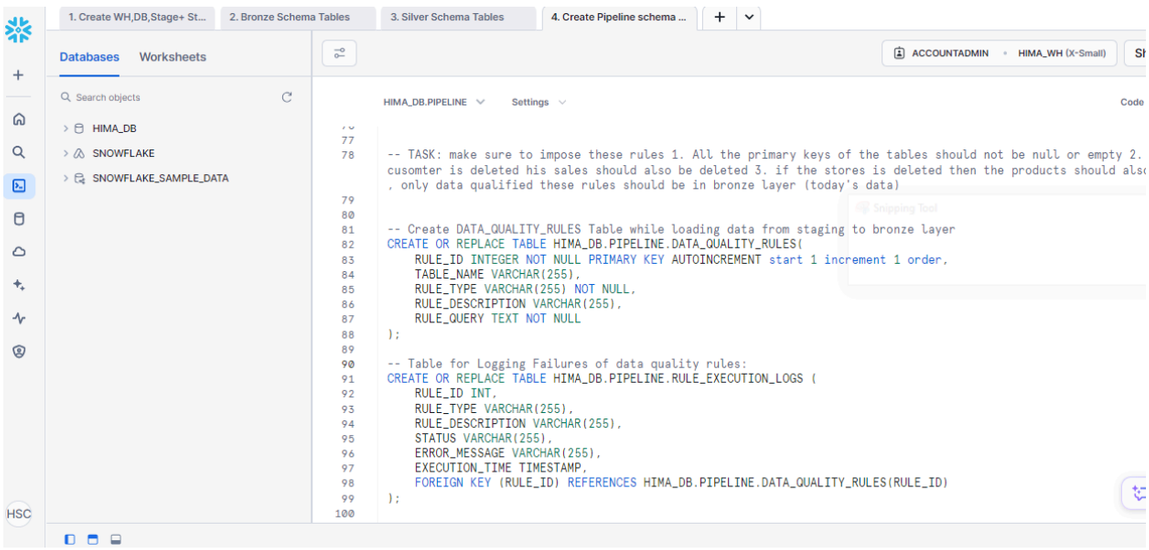
- Details of the RULE_EXECUTION_LOGS Table:

The logging table is used to capture and record any failures or issues that occur while applying data quality rules during the data loading process into the bronze layer. This helps in monitoring and troubleshooting data quality issues.

-- Table for Logging Failures of data quality rules:

```
CREATE OR REPLACE TABLE HIMA_DB.PIPELINE.RULE_EXECUTION_LOGS (
    RULE_ID INT,
```

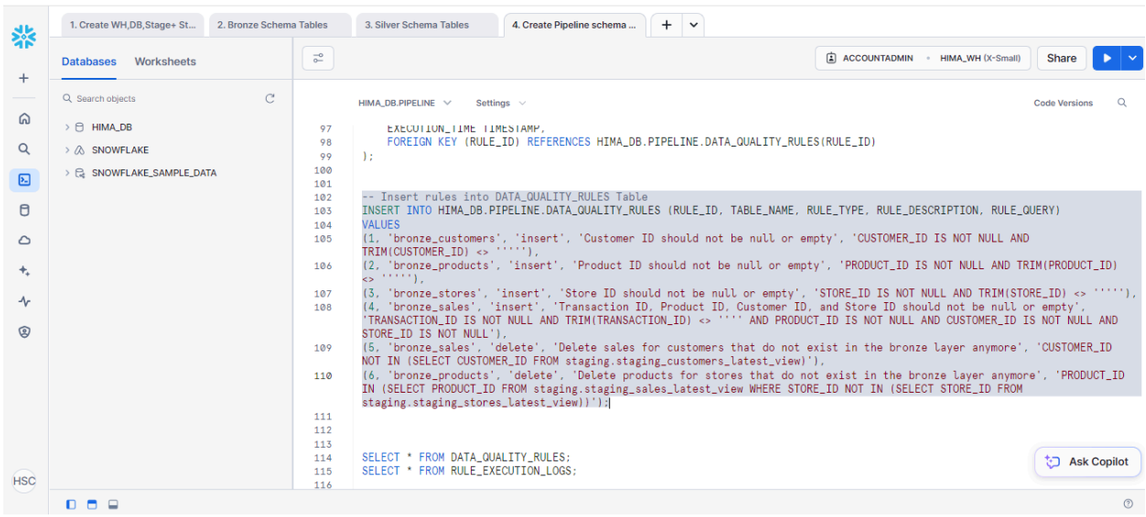
```
    RULE_TYPE VARCHAR(255),
    RULE_DESCRIPTION VARCHAR(255),
    STATUS VARCHAR(255),
    ERROR_MESSAGE VARCHAR(255),
    EXECUTION_TIME TIMESTAMP,
    FOREIGN KEY (RULE_ID) REFERENCES
HIMA_DB.PIPELINE.DATA_QUALITY_RULES(RULE_ID)
);
```



- To insert rules into the DATA_QUALITY_RULES table, you need to define the rules that will be applied to the data in the bronze tables.

```
-- Insert rules into DATA_QUALITY_RULES Table
INSERT INTO HIMA_DB.PIPELINE.DATA_QUALITY_RULES (RULE_ID, TABLE_NAME,
RULE_TYPE, RULE_DESCRIPTION, RULE_QUERY)
VALUES
(1, 'bronze_customers', 'insert', 'Customer ID should not be null or empty', 'CUSTOMER_ID IS
NOT NULL AND TRIM(CUSTOMER_ID) <> ''''),
(2, 'bronze_products', 'insert', 'Product ID should not be null or empty', 'PRODUCT_ID IS NOT
NULL AND TRIM(PRODUCT_ID) <> ''''),
(3, 'bronze_stores', 'insert', 'Store ID should not be null or empty', 'STORE_ID IS NOT NULL
AND TRIM(STORE_ID) <> ''''),
(4, 'bronze_sales', 'insert', 'Transaction ID, Product ID, Customer ID, and Store ID should not be
null or empty', 'TRANSACTION_ID IS NOT NULL AND TRIM(TRANSACTION_ID) <> ''''
AND PRODUCT_ID IS NOT NULL AND CUSTOMER_ID IS NOT NULL AND STORE_ID
IS NOT NULL'),
(5, 'bronze_sales', 'delete', 'Delete sales for customers that do not exist in the bronze layer
anymore', 'CUSTOMER_ID NOT IN (SELECT CUSTOMER_ID FROM
staging.staging_customers_latest_view)'),
```

(6, 'bronze_products', 'delete', 'Delete products for stores that do not exist in the bronze layer anymore', 'PRODUCT_ID IN (SELECT PRODUCT_ID FROM staging.staging_sales_latest_view WHERE STORE_ID NOT IN (SELECT STORE_ID FROM staging.staging_stores_latest_view))');



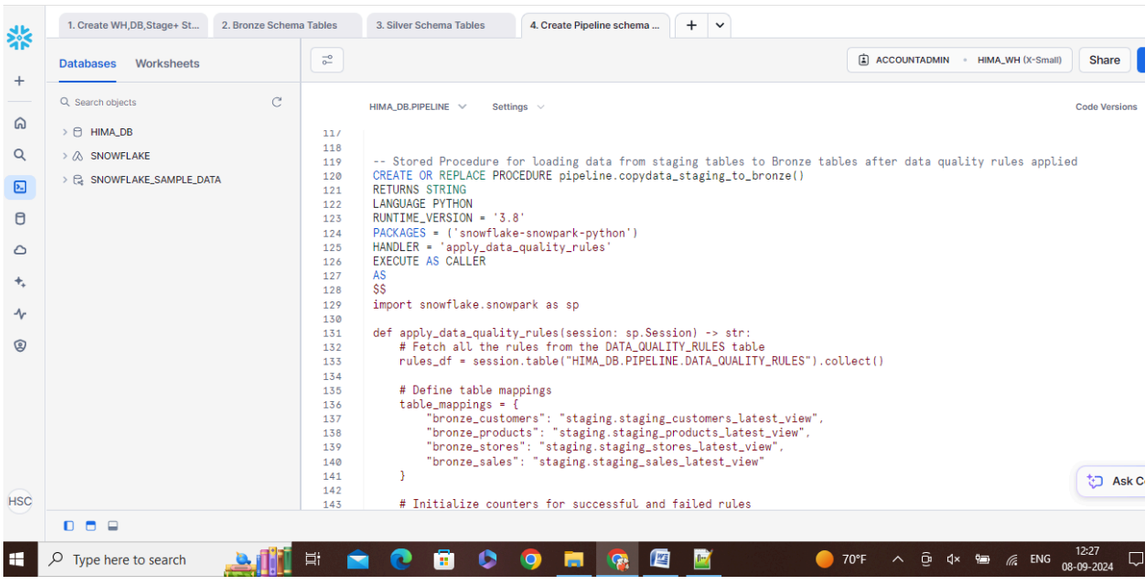
- Stored Procedure for Loading Data from Staging to Bronze Tables

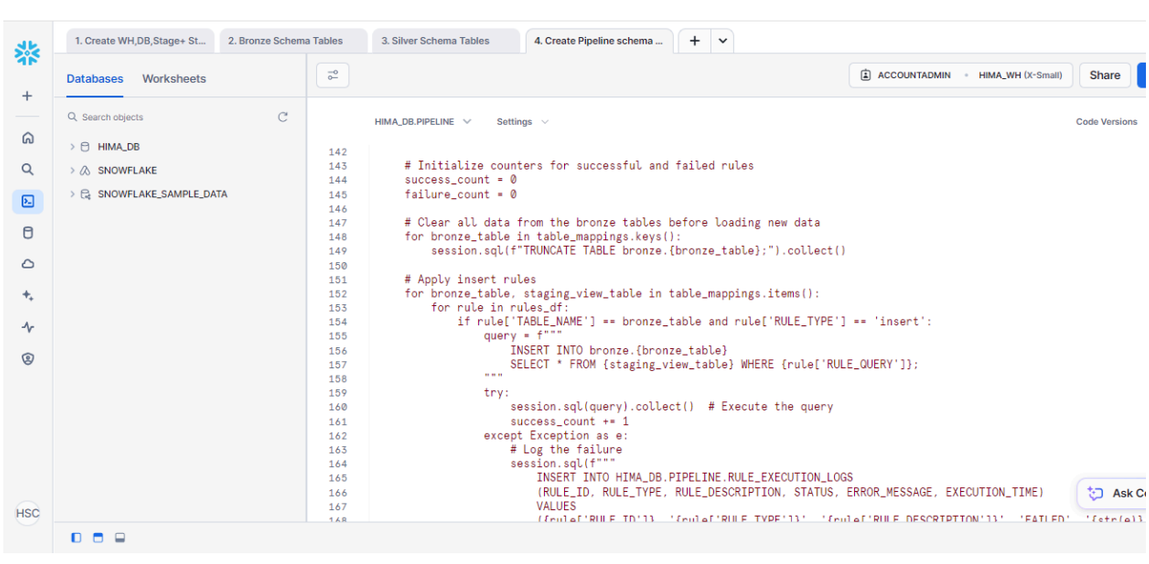
Below is a Python-based Snowflake stored procedure created in the pipeline schema to load data from staging tables to bronze tables after applying data quality rules. The procedure ensures that only data meeting these rules is transferred.

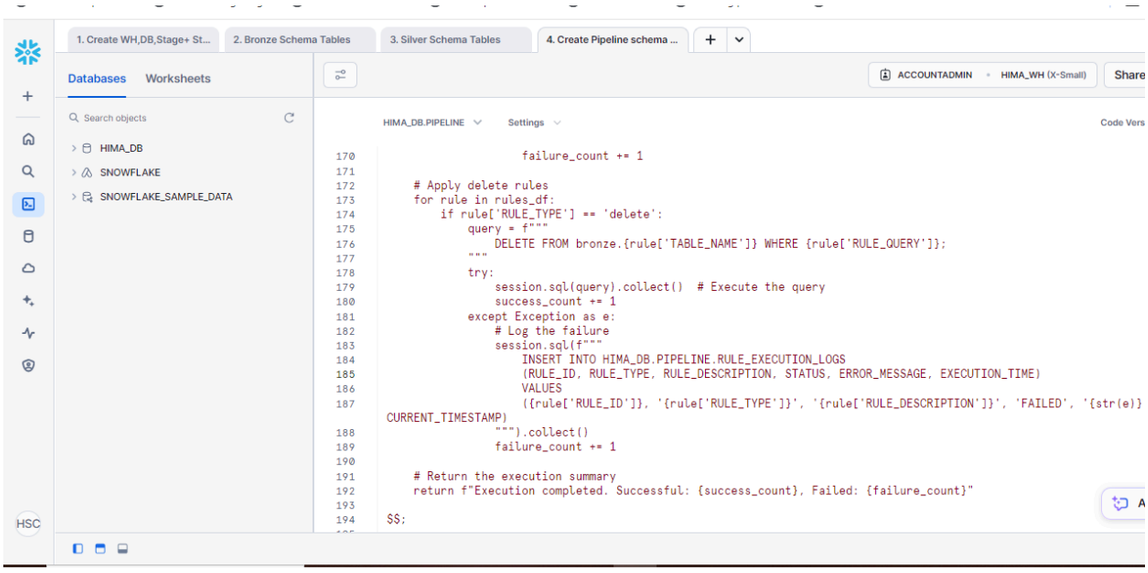
```
CREATE OR REPLACE PROCEDURE pipeline.copydata_staging_to_bronze()
RETURNS STRING
LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('snowflake-snowpark-python')
HANDLER = 'apply_data_quality_rules'
EXECUTE AS CALLER
AS
$$
import snowflake.snowpark as sp

def apply_data_quality_rules(session: sp.Session) -> str:
    # Fetch all the rules from the DATA_QUALITY_RULES table
    rules_df = session.table("HIMA_DB.PIPELINE.DATA_QUALITY_RULES").collect()

    # Define table mappings
    table_mappings = {
        "bronze_customers": "staging.staging_customers_latest_view",
        "bronze_products": "staging.staging_products_latest_view",
        "bronze_stores": "staging.staging_stores_latest_view",
```

```
    "bronze_sales": "staging.staging_sales_latest_view"
}

# Initialize counters for successful and failed rules
success_count = 0
failure_count = 0

# Clear all data from the bronze tables before loading new data
for bronze_table in table_mappings.keys():
    session.sql(f"TRUNCATE TABLE bronze.{bronze_table};").collect()

# Apply insert rules
for bronze_table, staging_view_table in table_mappings.items():
    for rule in rules_df:
        if rule['TABLE_NAME'] == bronze_table and rule['RULE_TYPE'] == 'insert':
            query = f"""
                INSERT INTO bronze.{bronze_table}
                SELECT * FROM {staging_view_table} WHERE {rule['RULE_QUERY']};
            """
            try:
                session.sql(query).collect()  # Execute the query
                success_count += 1
            except Exception as e:
                # Log the failure
                session.sql(f"""
                    INSERT INTO HIMA_DB.PIPELINE.RULE_EXECUTION_LOGS
                    (RULE_ID, RULE_TYPE, RULE_DESCRIPTION, STATUS,
ERROR_MESSAGE, EXECUTION_TIME)
                    VALUES
                    ({rule['RULE_ID']}, '{rule['RULE_TYPE']}', '{rule['RULE_DESCRIPTION']}',
'FAILED', '{str(e)}', CURRENT_TIMESTAMP)
                """).collect()
                failure_count += 1

# Apply delete rules
for rule in rules_df:
    if rule['RULE_TYPE'] == 'delete':
        query = f"""
            DELETE FROM bronze.{rule['TABLE_NAME']} WHERE {rule['RULE_QUERY']};
        """
        try:
            session.sql(query).collect()  # Execute the query
```

```
            success_count += 1
        except Exception as e:
            # Log the failure
            session.sql(f"""
                INSERT INTO HIMA_DB.PIPELINE.RULE_EXECUTION_LOGS
                (RULE_ID, RULE_TYPE, RULE_DESCRIPTION, STATUS,
ERROR_MESSAGE, EXECUTION_TIME)
                VALUES
                ({rule['RULE_ID']}, '{rule['RULE_TYPE']}', '{rule['RULE_DESCRIPTION']}',
'FAILED', '{str(e)}', CURRENT_TIMESTAMP)
            """).collect()
            failure_count += 1

    # Return the execution summary
    return f"Execution completed. Successful: {success_count}, Failed: {failure_count}"

$$;
```



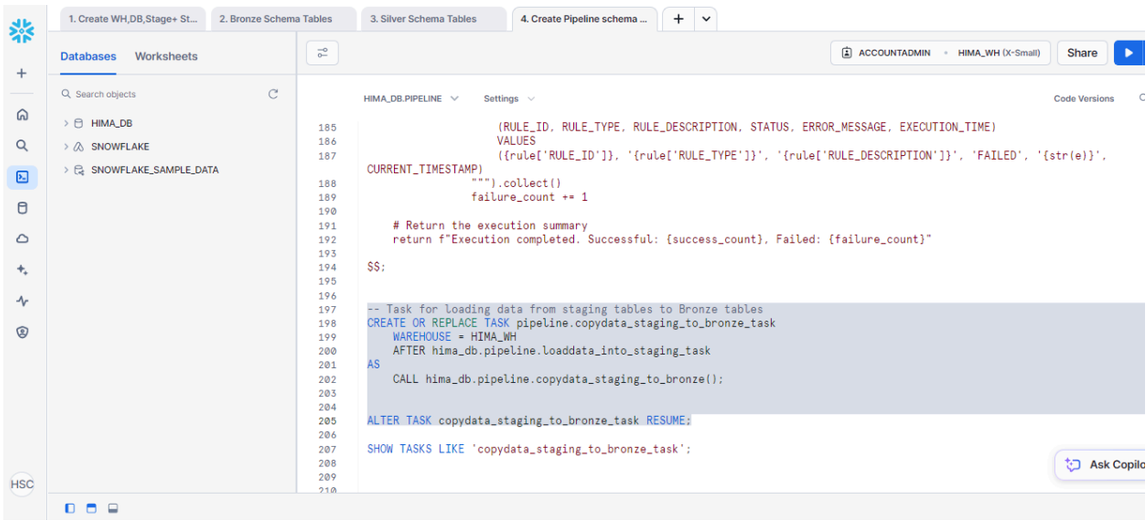


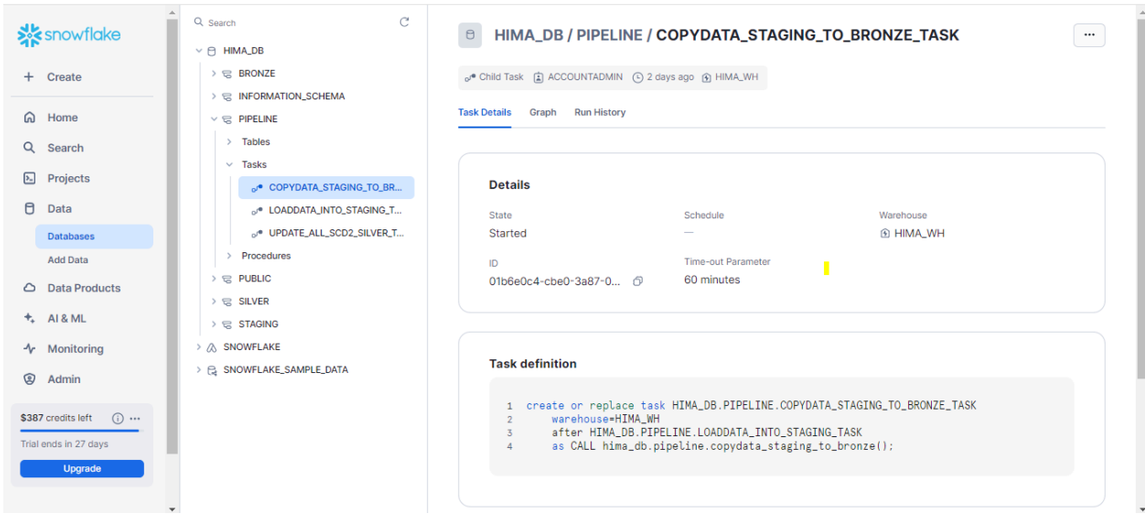


- Created a task for bronze stored procedure

This task runs a bronze stored procedure daily after the staging task, ensures that the bronze data loading procedure only runs after the staging tables have been updated, keeping the data pipeline streamlined and synchronized.

```
-- Task for loading data from staging tables to Bronze tables
CREATE OR REPLACE TASK pipeline.copydata_staging_to_bronze_task
    WAREHOUSE = HIMA_WH
    AFTER hima_db.pipeline.loaddata_into_staging_task
AS
    CALL hima_db.pipeline.copydata_staging_to_bronze();

ALTER TASK copydata_staging_to_bronze_task RESUME;
```





Process for Implementing Silver Layer Tables with SCD Type 2

TASK: Created silver layer tables that implement Slowly Changing Dimension (SCD) Type 2. The objective is to develop a proof of concept (POC) for deploying an SCD Type 2 table in Snowflake. The SCD Type 2 table should reflect changes made in the base table and incorporate the following features:

1. Record Updates: When a record is updated in the base table, the same record should be added as a new entry in the SCD Type 2 table. This new record should be marked as active.

2. Inactive Records: The previous version of the record should be marked as inactive in the SCD Type 2 table.

3. Record Deletions: If a record is deleted from the base table, the corresponding record in the SCD Type 2 table should be marked as inactive.

4. Date Tracking: Add START_DATE and END_DATE columns to the SCD Type 2 table to track the validity period of each record version. The START_DATE should be set when the record is inserted or updated, and the END_DATE should be updated when the record is deactivated or replaced.

- Created Silver Schema and Tables

```
use warehouse HIMA_WH;
use database HIMA_DB;
create or replace schema SILVER;

CREATE OR REPLACE TABLE silver.scd2_silver_customers (
    SURROGATE_KEY INTEGER AUTOINCREMENT PRIMARY KEY,
    CUSTOMER_ID TEXT,
```

```
    CUSTOMER_NAME TEXT,
    DATE_OF_BIRTH DATE,
    START_DATE DATE DEFAULT CURRENT_DATE,
    END_DATE DATE,
    IS_ACTIVE BOOLEAN DEFAULT TRUE
);

CREATE OR REPLACE TABLE silver.scd2_silver_products (
    SURROGATE_KEY INTEGER AUTOINCREMENT PRIMARY KEY,
    PRODUCT_ID TEXT,
    PRODUCT_NAME TEXT,
    PRICE FLOAT,
    START_DATE DATE DEFAULT CURRENT_DATE,
    END_DATE DATE,
    IS_ACTIVE BOOLEAN DEFAULT TRUE
);

CREATE OR REPLACE TABLE silver.scd2_silver_stores (
    SURROGATE_KEY INTEGER AUTOINCREMENT PRIMARY KEY,
    STORE_ID TEXT,
    STORE_NAME TEXT,
    START_DATE DATE DEFAULT CURRENT_DATE,
    END_DATE DATE,
    IS_ACTIVE BOOLEAN DEFAULT TRUE
);

CREATE OR REPLACE TABLE silver.scd2_silver_sales (
    SURROGATE_KEY INTEGER AUTOINCREMENT PRIMARY KEY,
    TRANSACTION_ID INTEGER,
    PRODUCT_ID TEXT,
    CUSTOMER_ID TEXT,
    STORE_ID TEXT,
    T_DATE DATE,
    QUANTITY INTEGER,
    START_DATE DATE DEFAULT CURRENT_DATE,
    END_DATE DATE,
    IS_ACTIVE BOOLEAN DEFAULT TRUE
);
```

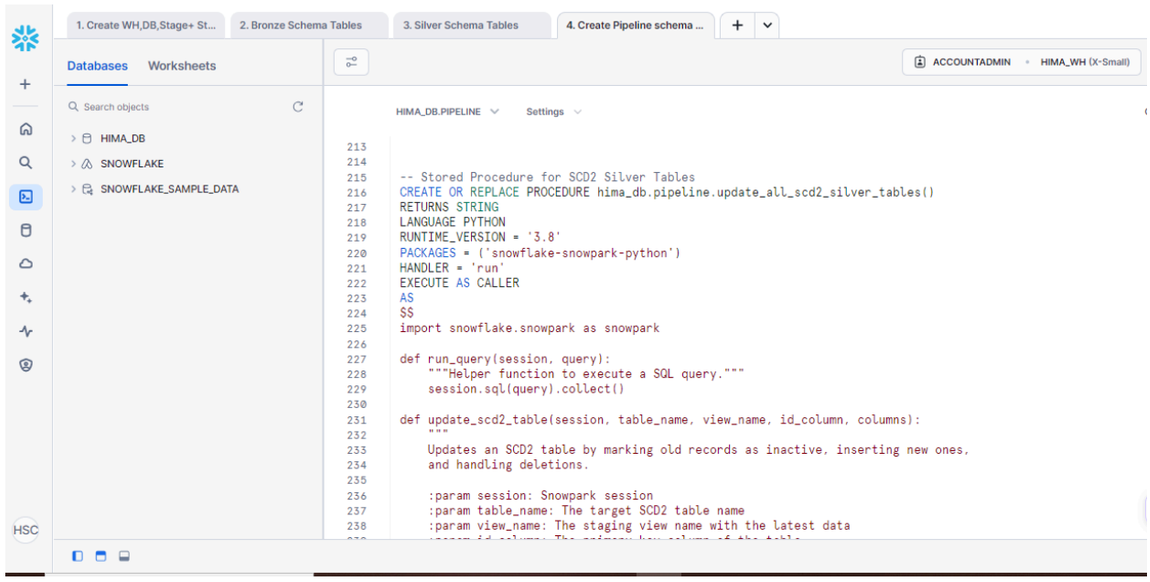
- Stored Procedure for SCD Type 2 in Silver Tables

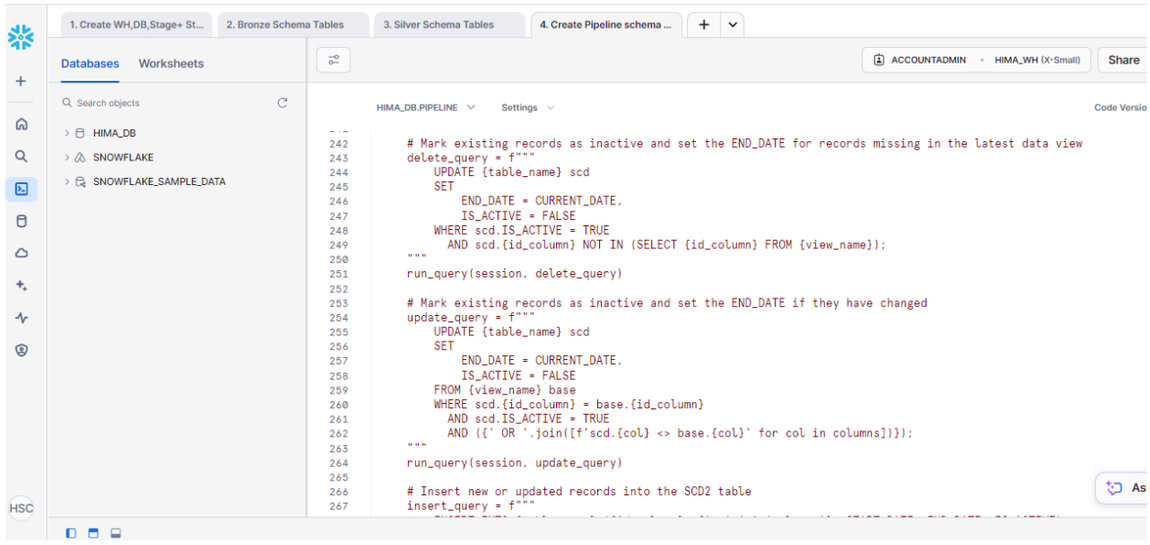
Below is a Python-based Snowflake stored procedure for implementing Slowly Changing Dimension (SCD) Type 2 in silver layer tables. This procedure is located in the pipeline schema. It manages the SCD Type 2 functionality by performing the above mentioned task:

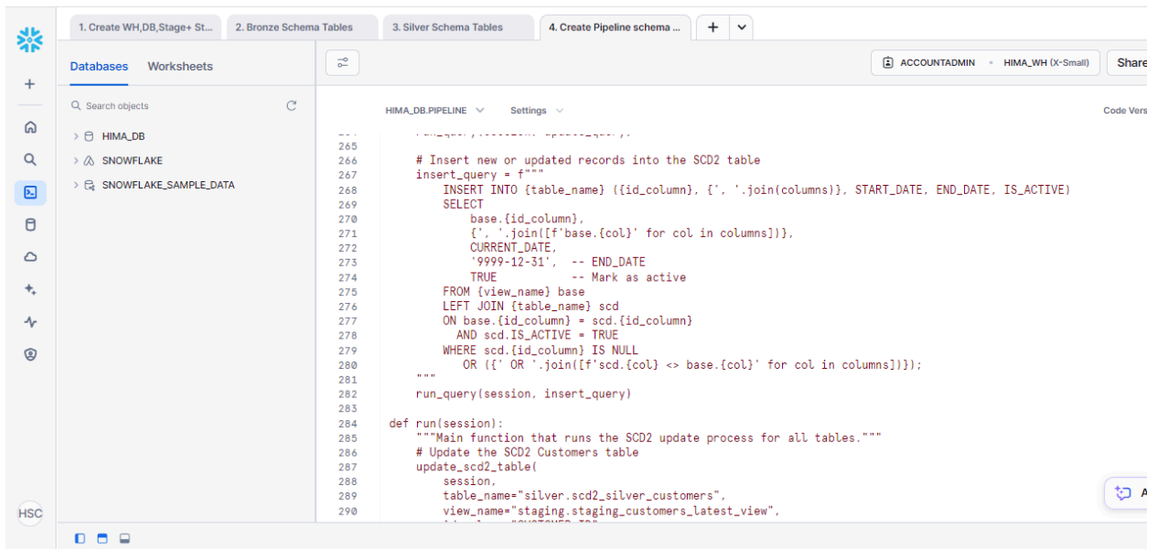
```
-- Stored Procedure for SCD2 Silver Tables
CREATE OR REPLACE PROCEDURE hima_db.pipeline.update_all_scd2_silver_tables()
RETURNS STRING
LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('snowflake-snowpark-python')
HANDLER = 'run'
EXECUTE AS CALLER
AS
$$
import snowflake.snowpark as snowpark

def run_query(session, query):
    """Helper function to execute a SQL query."""
    session.sql(query).collect()

def update_scd2_table(session, table_name, view_name, id_column, columns):
    """
    Updates an SCD2 table by marking old records as inactive, inserting new ones,
    and handling deletions.

    :param session: Snowpark session
    :param table_name: The target SCD2 table name
    :param view_name: The staging view name with the latest data
    :param id_column: The primary key column of the table
    :param columns: List of column names to compare for changes
    """
    # Mark existing records as inactive and set the END_DATE for records missing in the latest data view
    delete_query = f"""
        UPDATE {table_name} scd
        SET
            END_DATE = CURRENT_DATE,
            IS_ACTIVE = FALSE
        WHERE scd.IS_ACTIVE = TRUE
          AND scd.{id_column} NOT IN (SELECT {id_column} FROM {view_name});
    """
    run_query(session, delete_query)
```

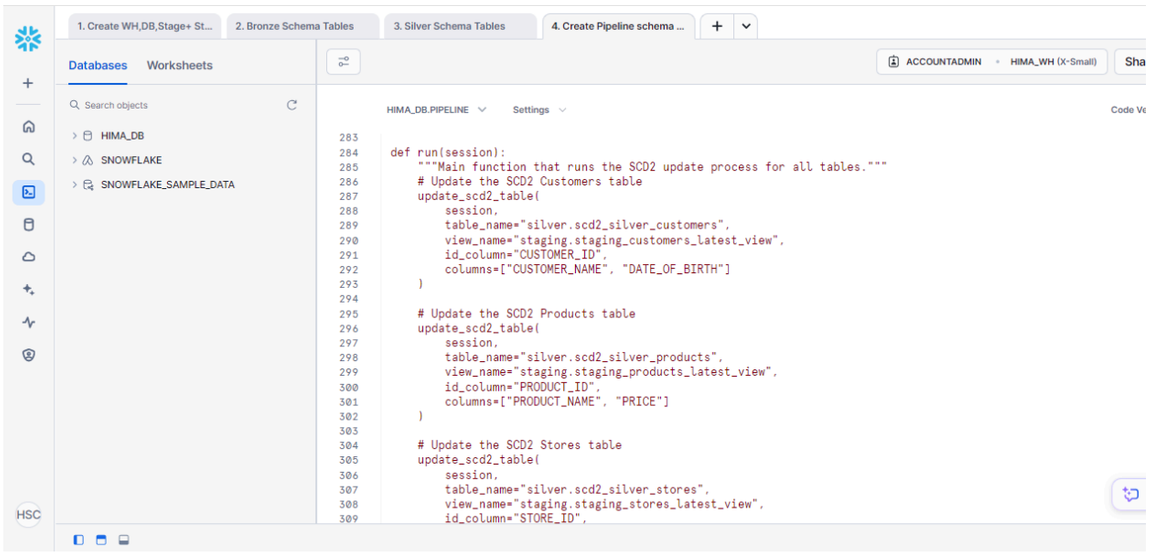
```
    # Mark existing records as inactive and set the END_DATE if they have changed
    update_query = f"""
        UPDATE {table_name} scd
        SET
            END_DATE = CURRENT_DATE,
            IS_ACTIVE = FALSE
        FROM {view_name} base
        WHERE scd.{id_column} = base.{id_column}
          AND scd.IS_ACTIVE = TRUE
          AND ({' OR '.join([f'scd.{col} <> base.{col}' for col in columns])});
    """
    run_query(session, update_query)

    # Insert new or updated records into the SCD2 table
    insert_query = f"""
        INSERT INTO {table_name} ({id_column}, {', '.join(columns)}, START_DATE,
END_DATE, IS_ACTIVE)
        SELECT
            base.{id_column},
            {', '.join([f'base.{col}' for col in columns])},
            CURRENT_DATE,
            '9999-12-31',  -- END_DATE
            TRUE           -- Mark as active
        FROM {view_name} base
        LEFT JOIN {table_name} scd
        ON base.{id_column} = scd.{id_column}
          AND scd.IS_ACTIVE = TRUE
        WHERE scd.{id_column} IS NULL
           OR ({' OR '.join([f'scd.{col} <> base.{col}' for col in columns])});
    """
    run_query(session, insert_query)

def run(session):
    """Main function that runs the SCD2 update process for all tables."""
    # Update the SCD2 Customers table
    update_scd2_table(
        session,
        table_name="silver.scd2_silver_customers",
        view_name="staging.staging_customers_latest_view",
        id_column="CUSTOMER_ID",
        columns=["CUSTOMER_NAME", "DATE_OF_BIRTH"]
```

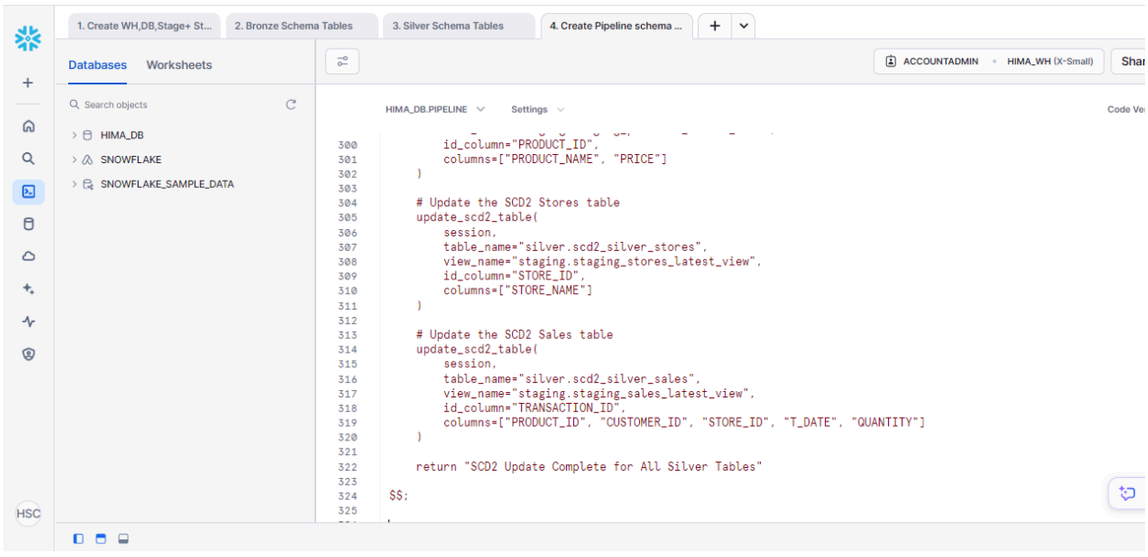
```
    )

    # Update the SCD2 Products table
    update_scd2_table(
        session,
        table_name="silver.scd2_silver_products",
        view_name="staging.staging_products_latest_view",
        id_column="PRODUCT_ID",
        columns=["PRODUCT_NAME", "PRICE"]
    )

    # Update the SCD2 Stores table
    update_scd2_table(
        session,
        table_name="silver.scd2_silver_stores",
        view_name="staging.staging_stores_latest_view",
        id_column="STORE_ID",
        columns=["STORE_NAME"]
    )

    # Update the SCD2 Sales table
    update_scd2_table(
        session,
        table_name="silver.scd2_silver_sales",
        view_name="bronze.bronze_sales",
        id_column="TRANSACTION_ID",
        columns=["PRODUCT_ID", "CUSTOMER_ID", "STORE_ID", "T_DATE",
"QUANTITY"]
    )

    return "SCD2 Update Complete for All Silver Tables"

$$;
```


1. Create WH,DB,Stage+ St...
2. Bronze Schema Tables
3. Silver Schema Tables
4. Create Pipeline schema ...
Databases
Worksheets
ACCOUNTADMIN
HIMA_WH (X-Small)
HIMA_DB.PIPELINE
Settings
HIMA_DB
SNOWFLAKE
SNOWFLAKE_SAMPLE_DATA
-- Stored Procedure for SCD2 Silver Tables
CREATE OR REPLACE PROCEDURE hima_db.pipeline.update_all_scd2_silver_tables()
RETURNS STRING
LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('snowflake-snowpark-python')
HANDLER = 'run'
EXECUTE AS CALLER
AS
$$
import snowflake.snowpark as snowpark
def run_query(session, query):
"""Helper function to execute a SQL query."""
session.sql(query).collect()
def update_scd2_table(session, table_name, view_name, id_column, columns):
"""
Updates an SCD2 table by marking old records as inactive, inserting new ones,
and handling deletions.
:param session: Snowpark session
:param table_name: The target SCD2 table name
:param view_name: The staging view name with the latest data


1. Create WH,DB,Stage+ St...
2. Bronze Schema Tables
3. Silver Schema Tables
4. Create Pipeline schema ...
Databases
Worksheets
ACCOUNTADMIN
HIMA_WH (X-Small)
Share
HIMA_DB.PIPELINE
Settings
HIMA_DB
SNOWFLAKE
SNOWFLAKE_SAMPLE_DATA
# Mark existing records as inactive and set the END_DATE for records missing in the latest data view
delete_query = f"""
UPDATE {table_name} scd
SET
END_DATE = CURRENT_DATE,
IS_ACTIVE = FALSE
WHERE scd.IS_ACTIVE = TRUE
AND scd.{id_column} NOT IN (SELECT {id_column} FROM {view_name});
"""
run_query(session, delete_query)
# Mark existing records as inactive and set the END_DATE if they have changed
update_query = f"""
UPDATE {table_name} scd
SET
END_DATE = CURRENT_DATE,
IS_ACTIVE = FALSE
FROM {view_name} base
WHERE scd.{id_column} = base.{id_column}
AND scd.IS_ACTIVE = TRUE
AND ({' OR '.join([f'scd.{col} <> base.{col}' for col in columns])});
"""
run_query(session, update_query)
# Insert new or updated records into the SCD2 table
insert_query = f"""

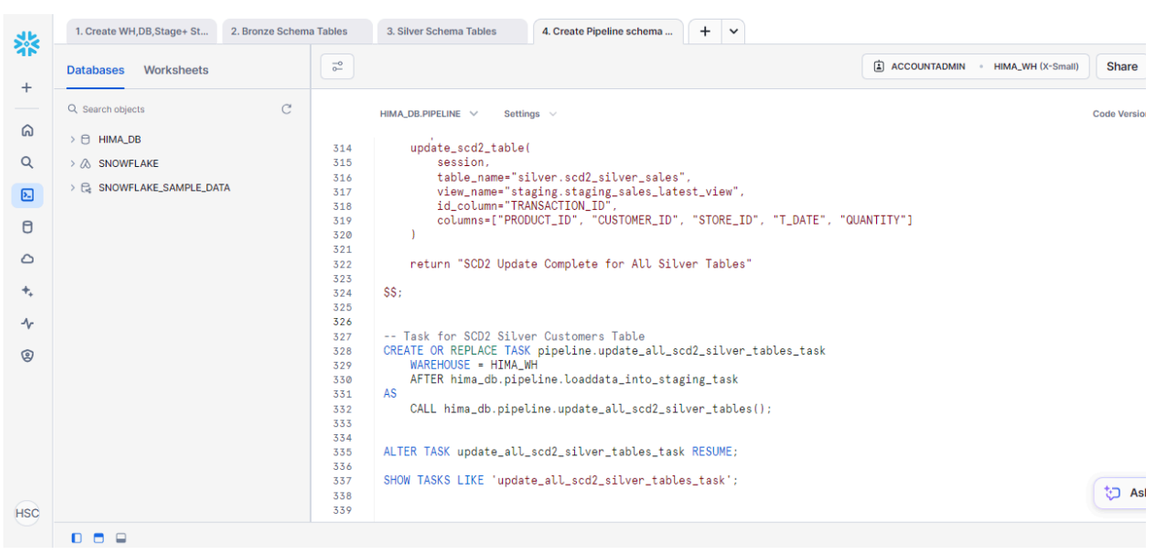

1. Create WH,DB,Stage+ St...
2. Bronze Schema Tables
3. Silver Schema Tables
4. Create Pipeline schema ...
Databases
Worksheets
ACCOUNTADMIN
HIMA_WH (X-Small)
Share
HIMA_DB.PIPELINE
Settings
HIMA_DB
SNOWFLAKE
SNOWFLAKE_SAMPLE_DATA
# Insert new or updated records into the SCD2 table
insert_query = f"""
INSERT INTO {table_name} ({id_column}, {', '.join(columns)}, START_DATE, END_DATE, IS_ACTIVE)
SELECT
base.{id_column},
{', '.join([f'base.{col}' for col in columns])},
CURRENT_DATE,
'9999-12-31', -- END_DATE
TRUE -- Mark as active
FROM {view_name} base
LEFT JOIN {table_name} scd
ON base.{id_column} = scd.{id_column}
AND scd.IS_ACTIVE = TRUE
WHERE scd.{id_column} IS NULL
OR ({' OR '.join([f'scd.{col} <> base.{col}' for col in columns])});
"""
run_query(session, insert_query)
def run(session):
"""Main function that runs the SCD2 update process for all tables."""
Update the SCD2 Customers table
update_scd2_table(
session,
table_name="silver.scd2_silver_customers",
view_name="staging.staging_customers_latest_view",

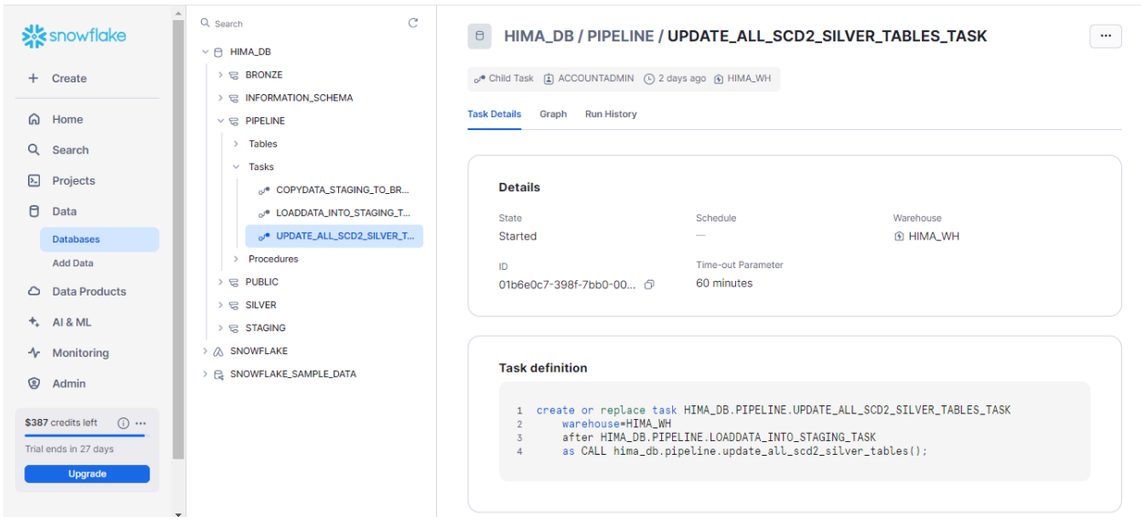




- Created a task for silver stored procedure

This task has been created in the pipeline schema that runs every day after the staging task. This task executes the stored procedure for the SCD Type 2 tables, ensuring that the data in the silver layer is updated according to the SCD Type 2 logic.





OUTPUT

Monitoring all layer tables—staging, bronze, and silver—for a period of three days. This monitoring involves checking for:

Data Consistency: Ensuring that data is correctly processed and reflects the expected changes in each layer.

Data Accuracy: Verifying that data transformations and rules are applied correctly.

Task Execution: Confirming that scheduled tasks, including data loading and SCD Type 2 updates, are running as planned.

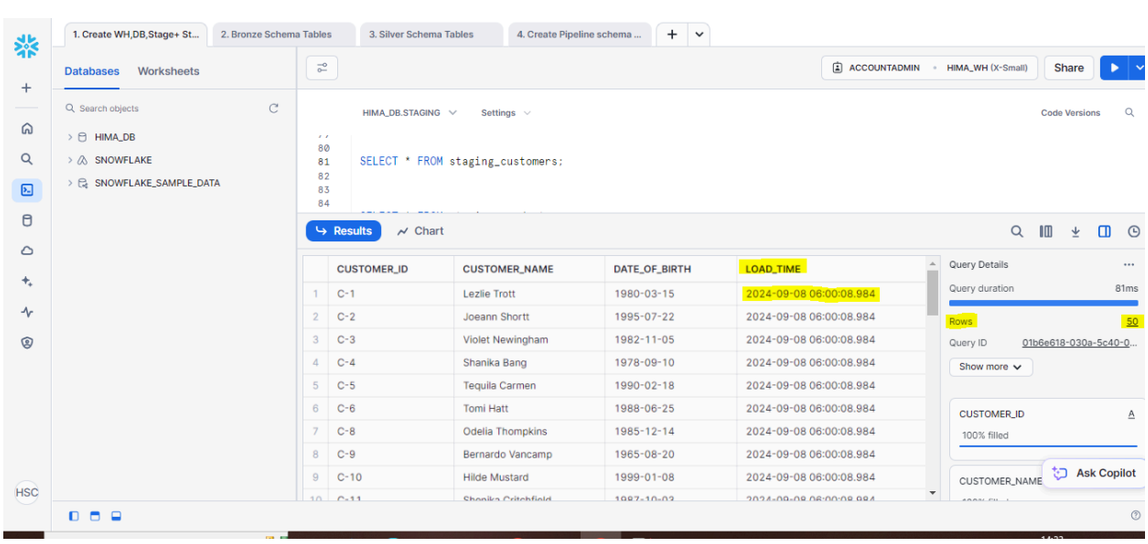
Error Logs: Reviewing logs for any errors or issues that may arise during data processing.

DAY 1

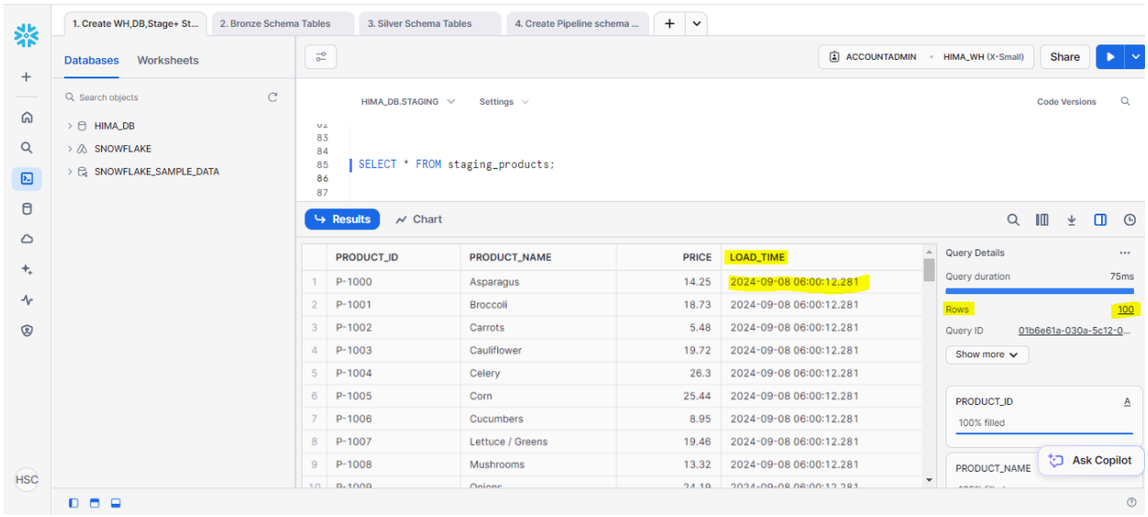
Check Loaded Data with SELECT queries for layers tables

STAGING LAYER TABLES

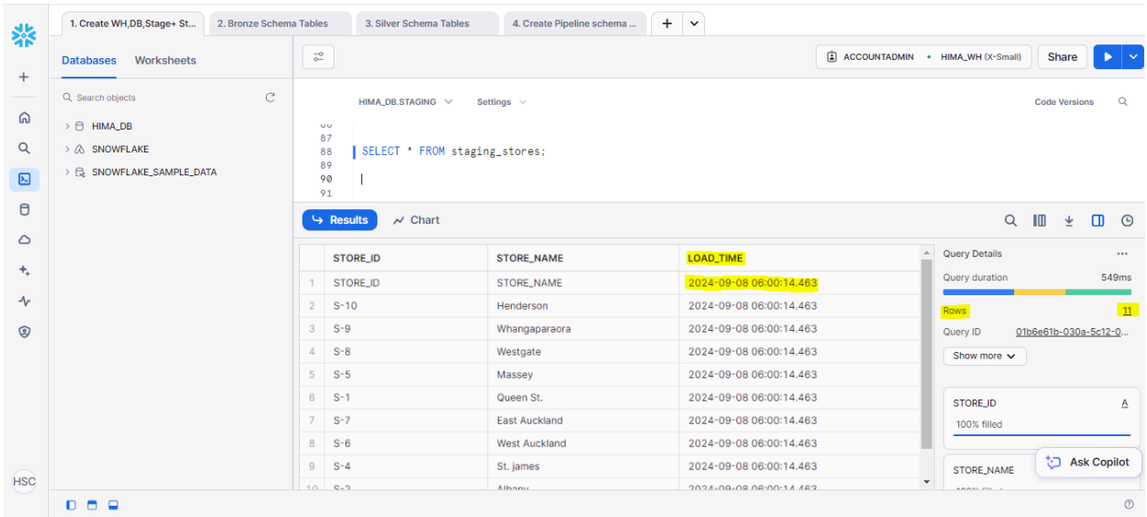
1. STAGING_CUSTOMERS table



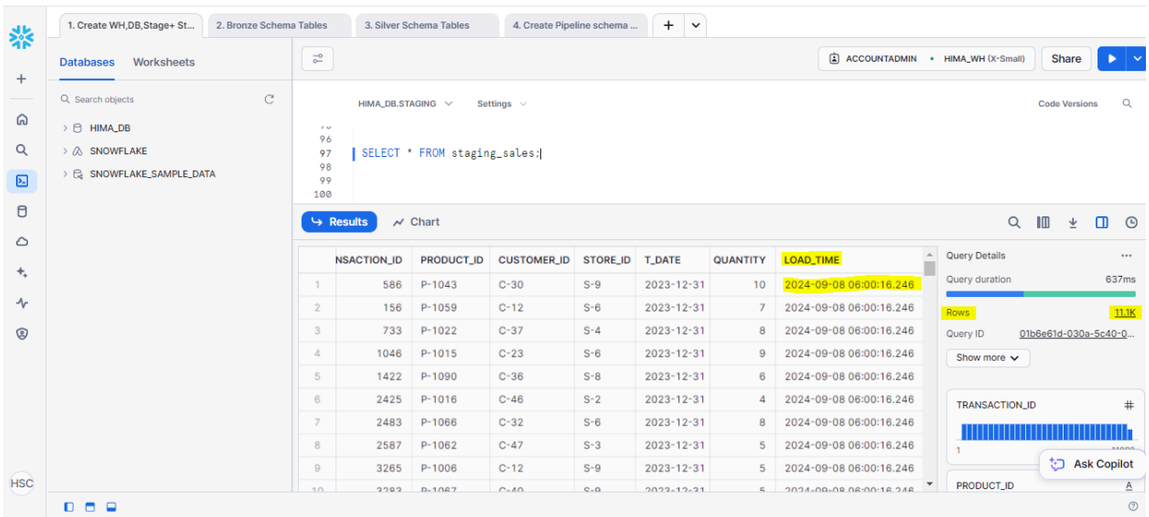
2. STAGING_PRODUCTS table



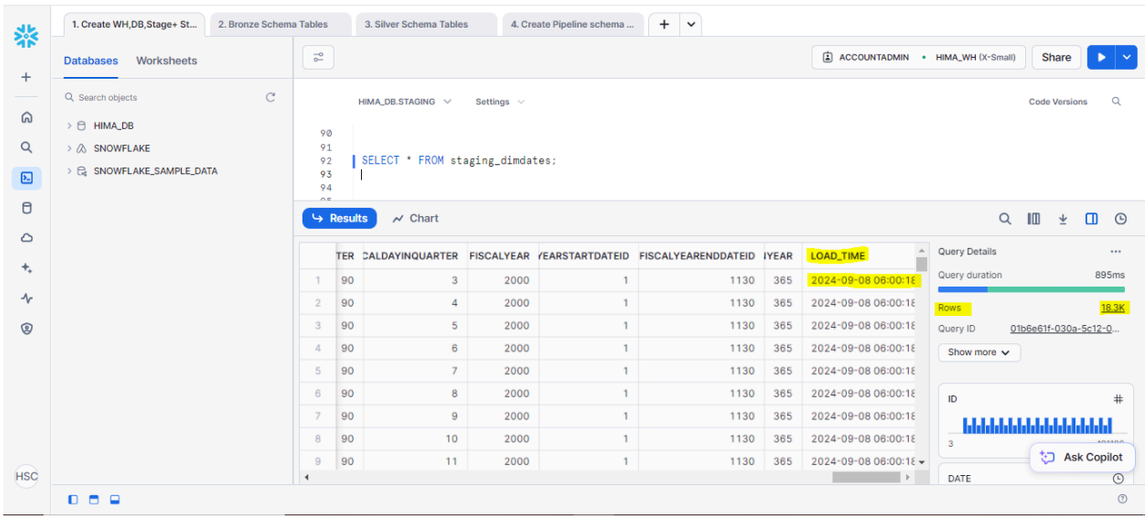
3. STAGING_STORES table



4. STAGING_SALES table

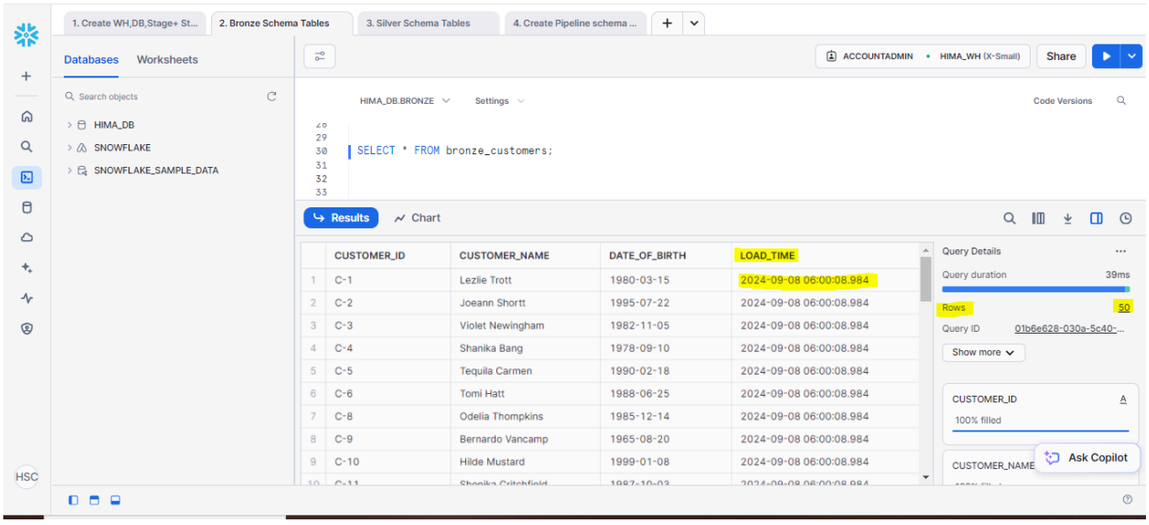


5. STAGING_DIMDATES table

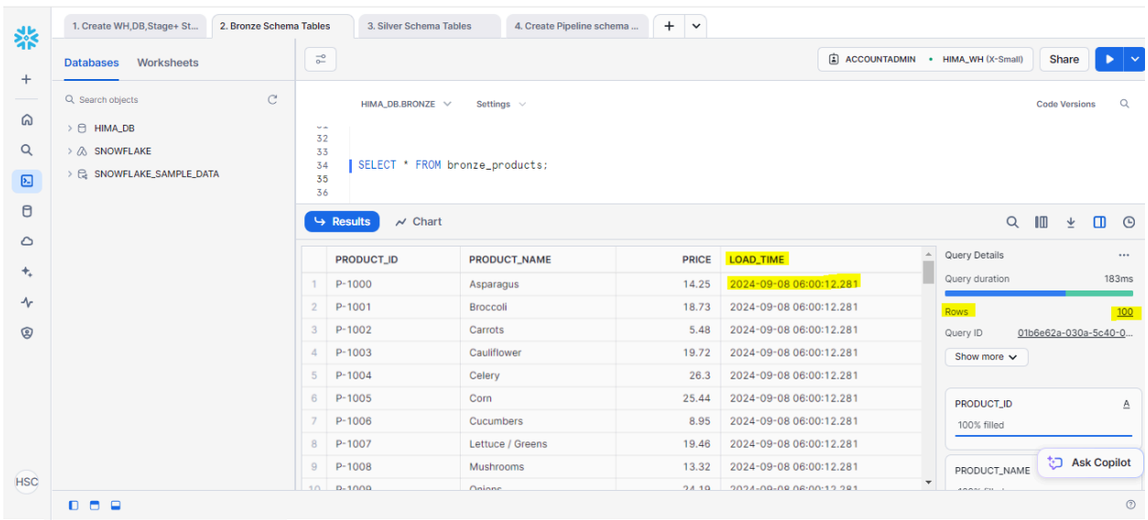


BRONZE LAYER TABLES

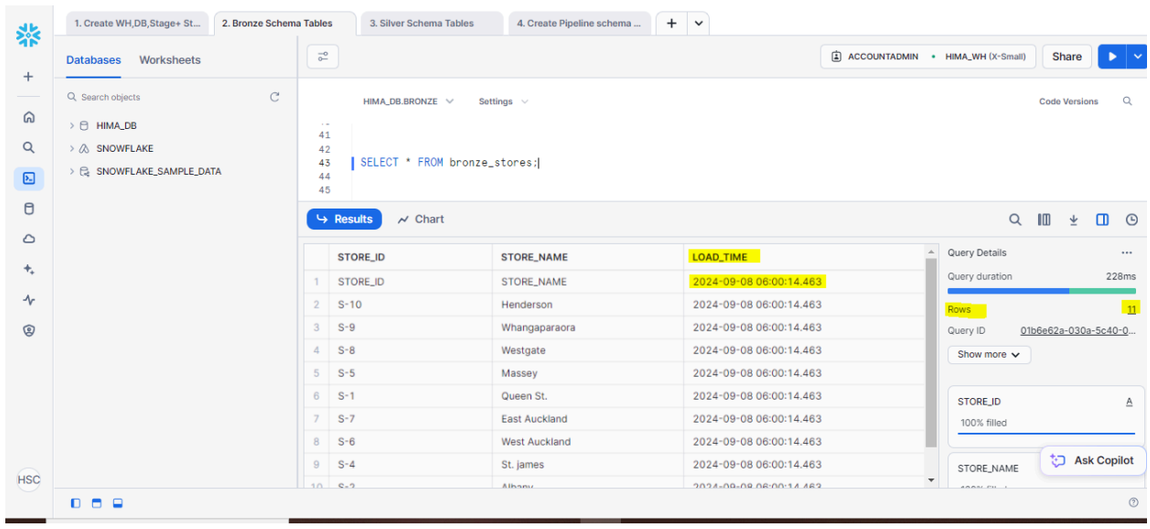
1. BRONZE_CUSTOMER table



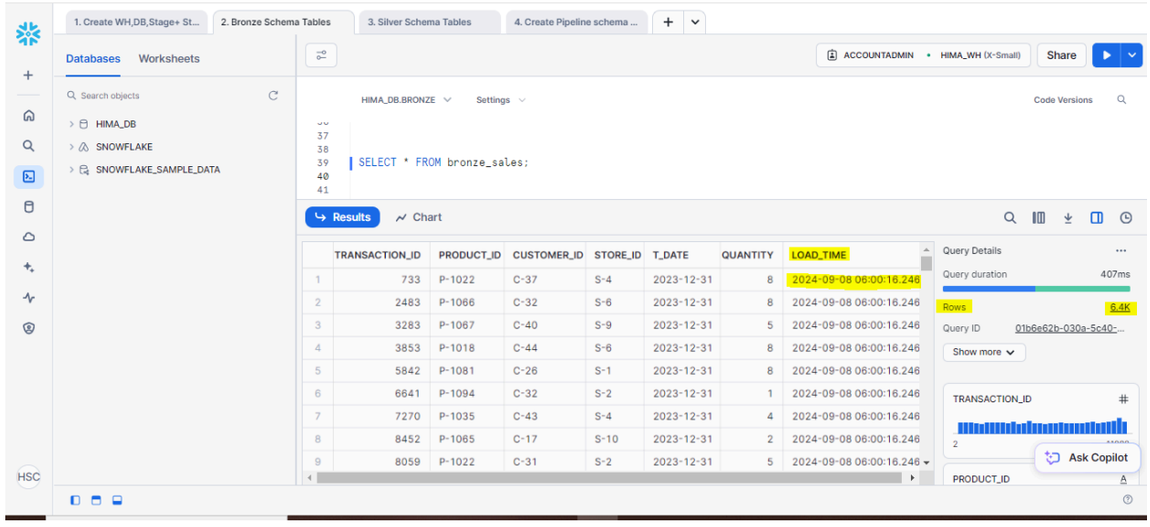
2. BRONZE_PRODUCTS table



3. BRONZE_STORES table

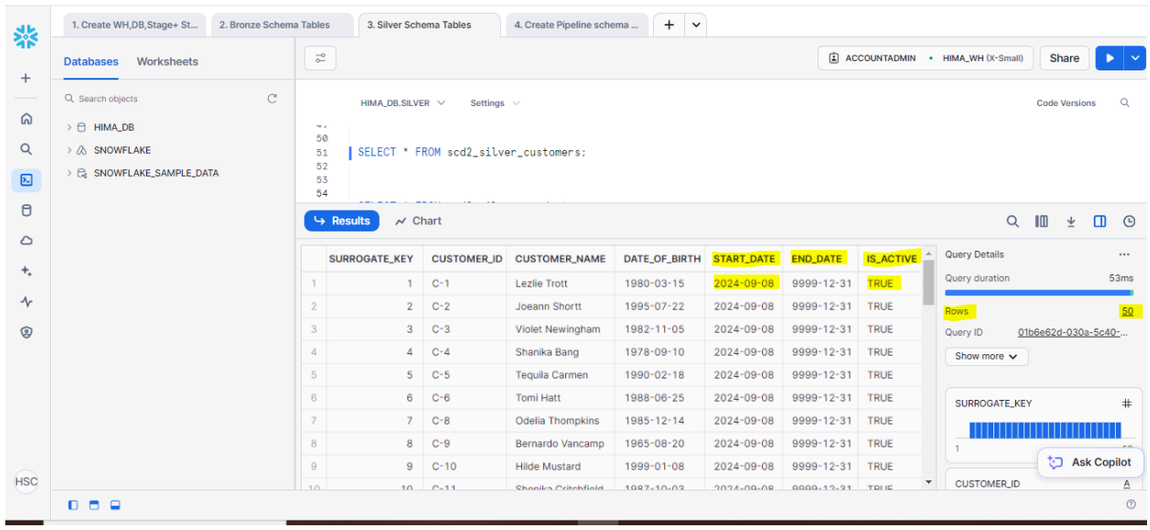


4. BRONZE_SALES table

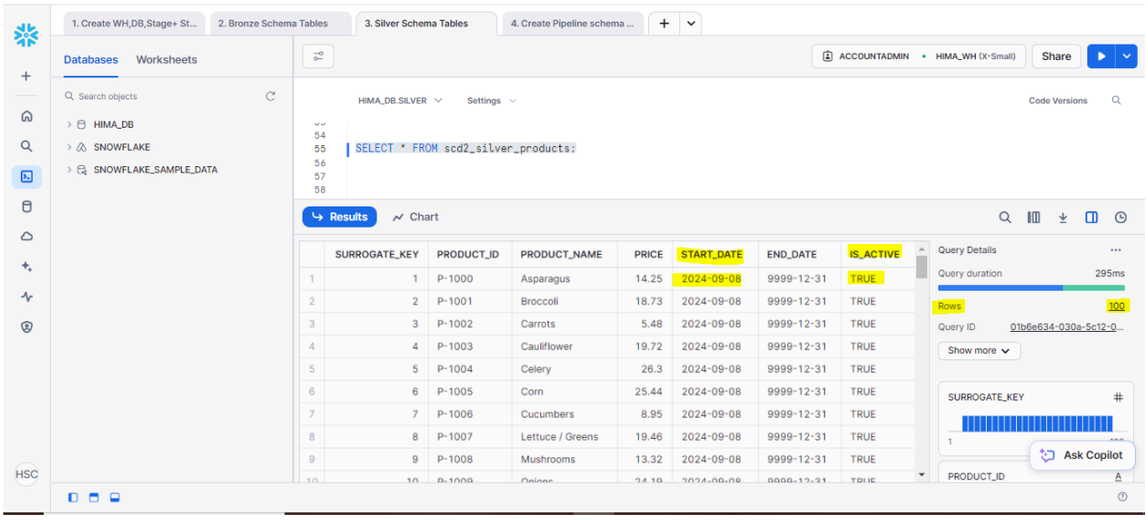


SILVER LAYER TABLES

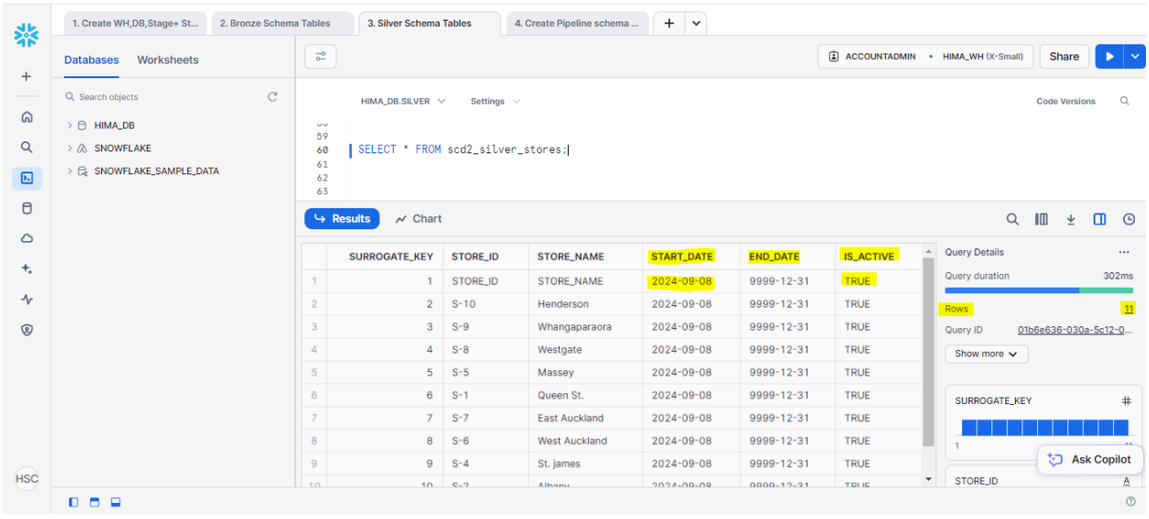
1. SCD2_SILVER_CUSTOMERS table



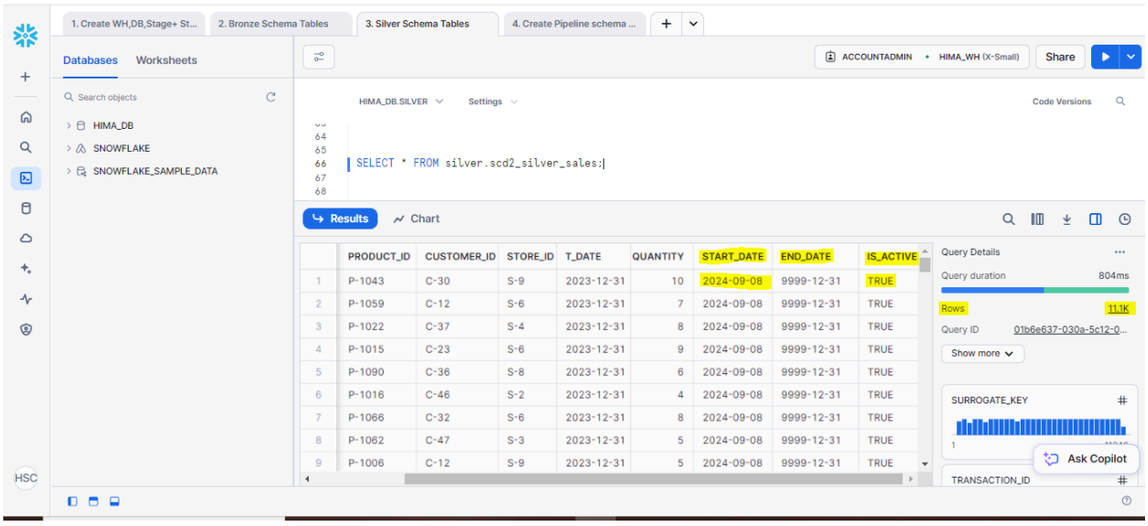
2. SCD2_SILVER_PRODUCTS



3. SCD2_SILVER_STORES table



4. SCD2_SILVER_SALES table



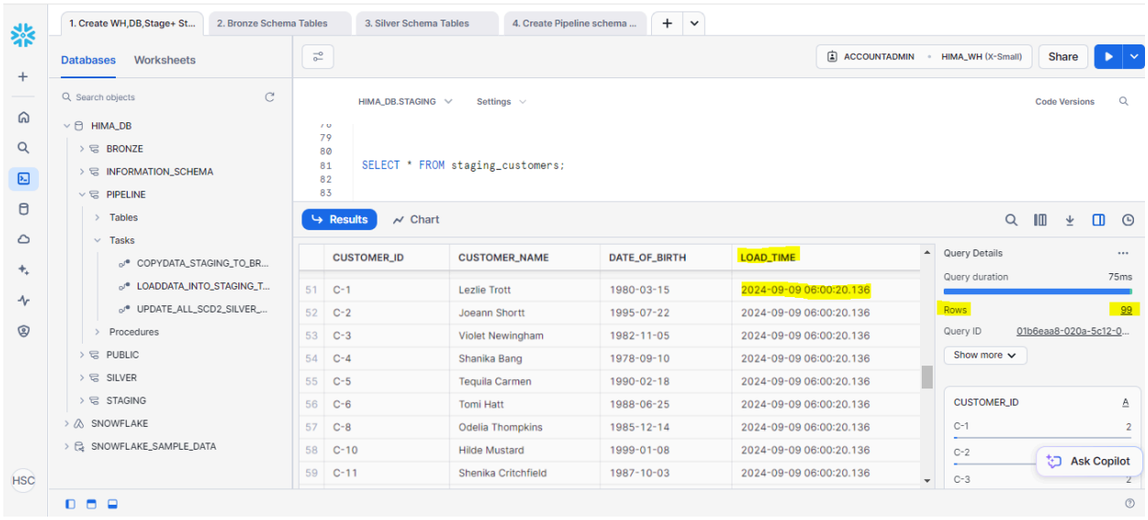
DAY 2

STAGING LAYER TABLES

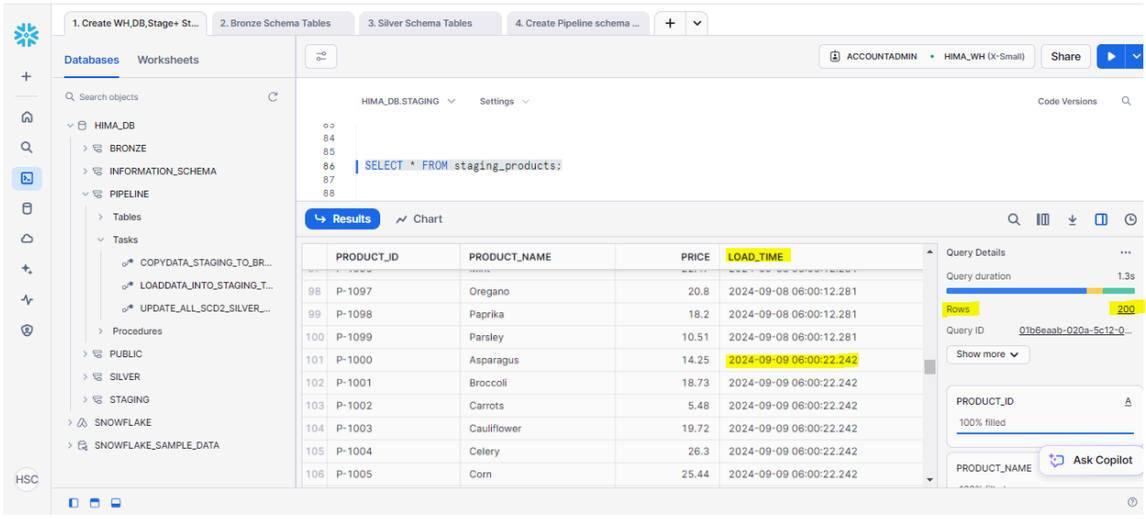
1. STAGING_CUSTOMERS table

For example:

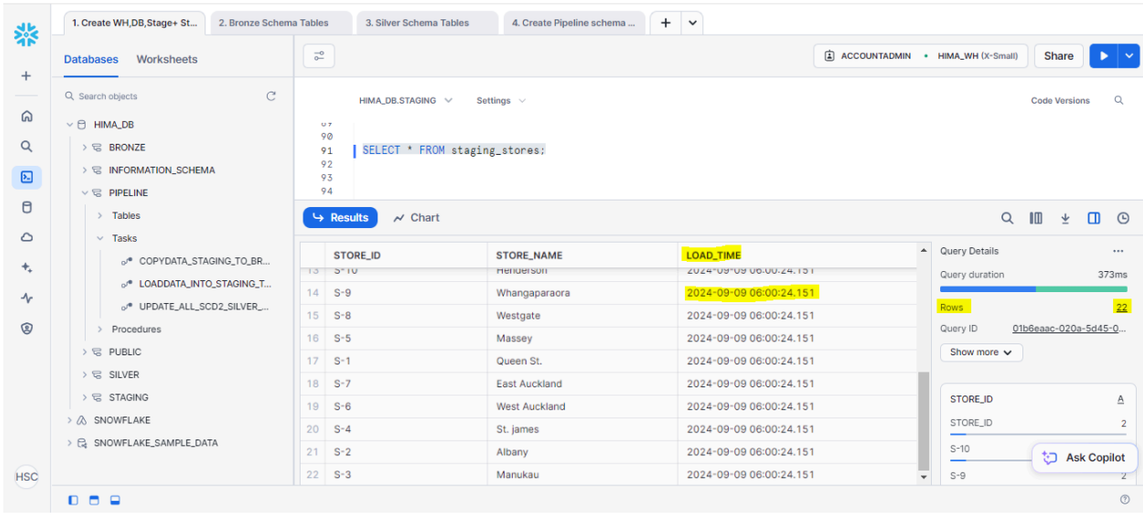
On Day 1, the staging_customers table includes a customer with the Customer ID C-9. However, on Day 2, this Customer ID C-9 is no longer present in the table.



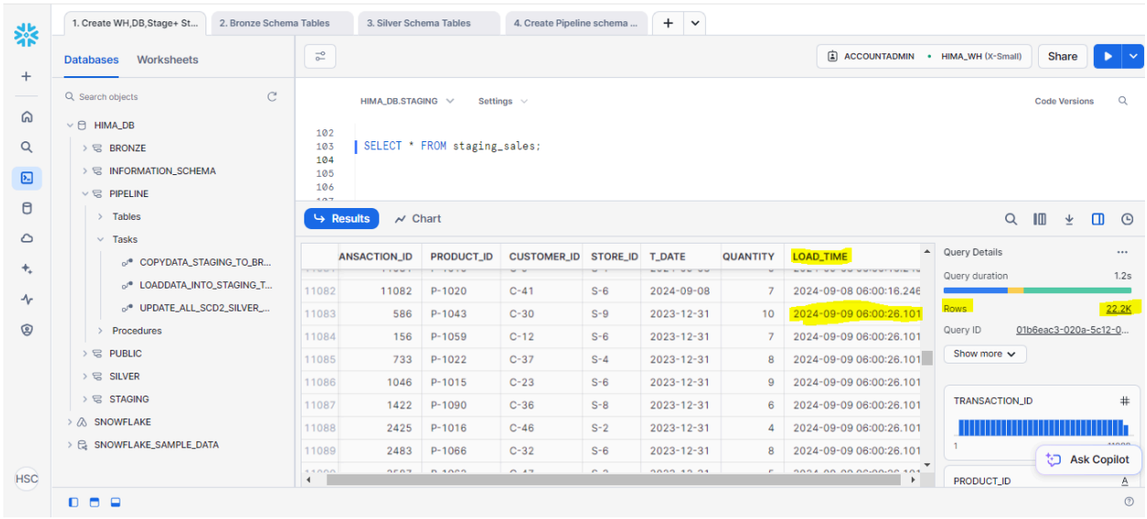
2. STAGING_PRODUCTS table



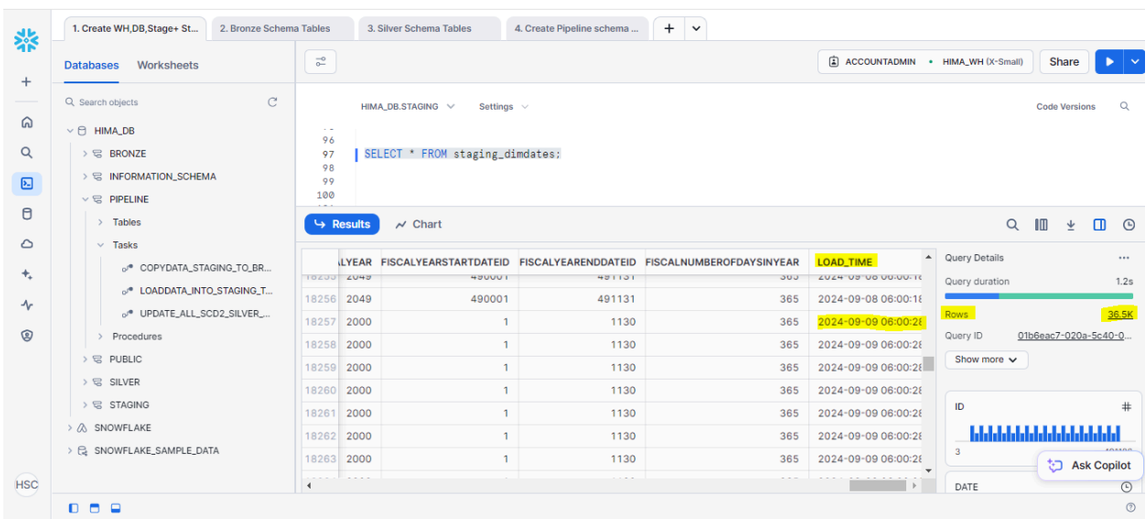
3. STAGING_STORES table



4. STAGING_SALES table



5. STAGING_DIMDATES table



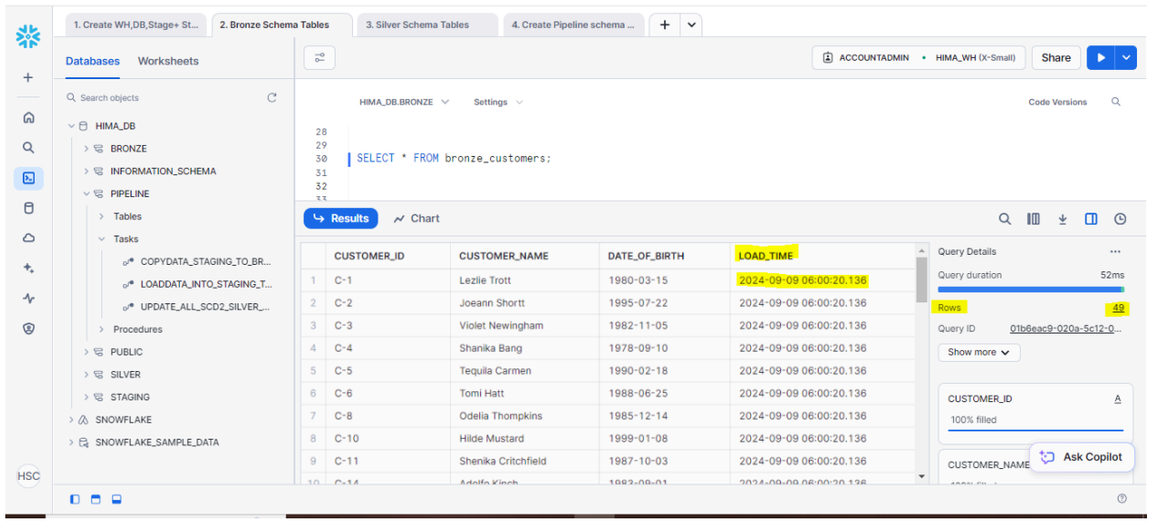
BRONZE LAYER TABLES

1. BRONZE_CUSTOMERS table

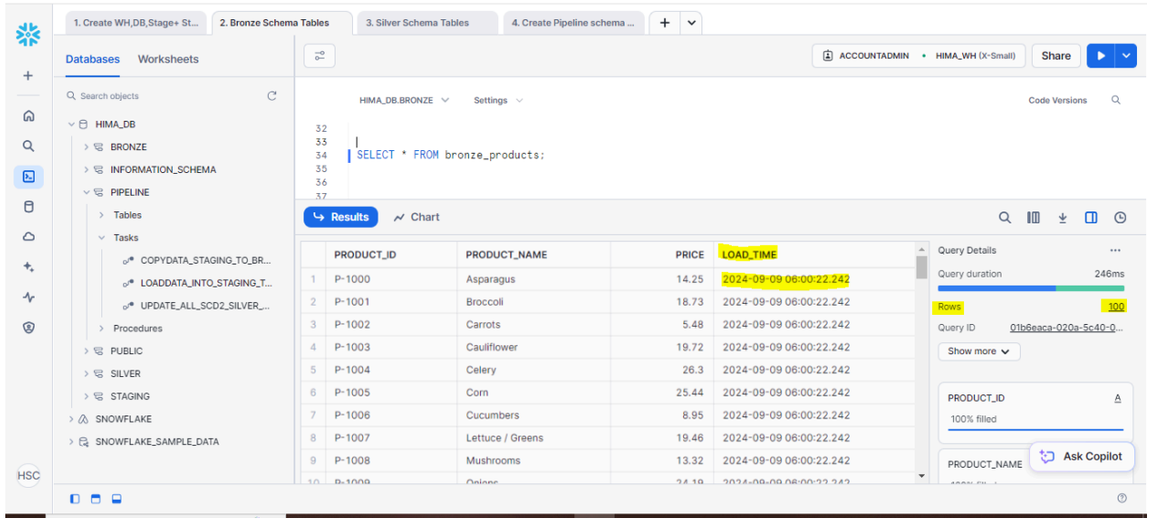
For example:

On Day 1, the `staging_customers` table contains a customer with Customer ID `C-9`. However, on Day 2, this Customer ID `C-9` is no longer present in the staging table.

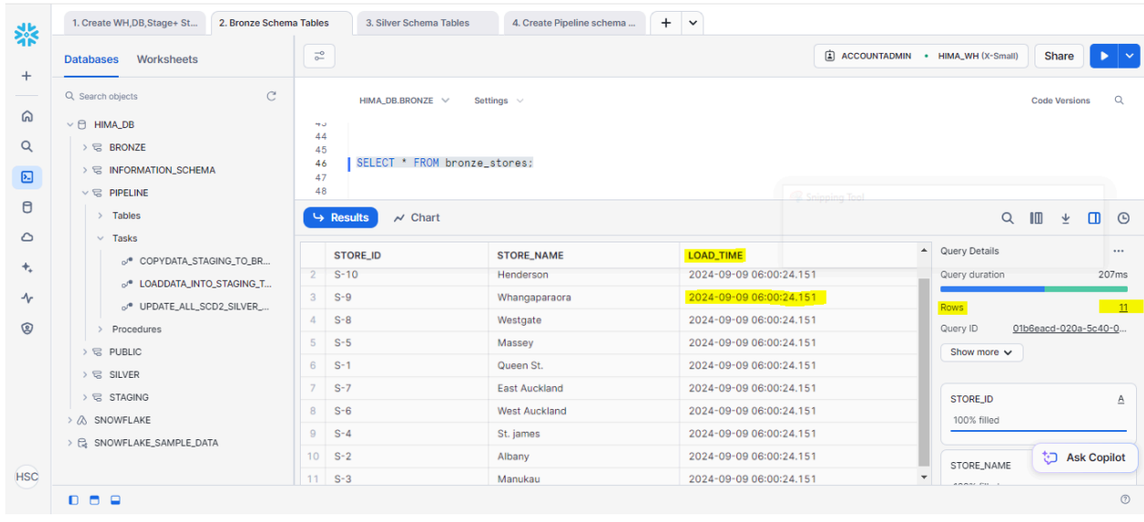
Since the bronze layer tables are designed to hold only the latest data, the Customer ID `C-9` is not present in the `bronze_customers` table. Consequently, in the `bronze_sales` table, any sales records related to Customer ID `C-9` are also not present, reflecting the enforcement of data quality rules where related sales data is removed when a customer no longer exists.



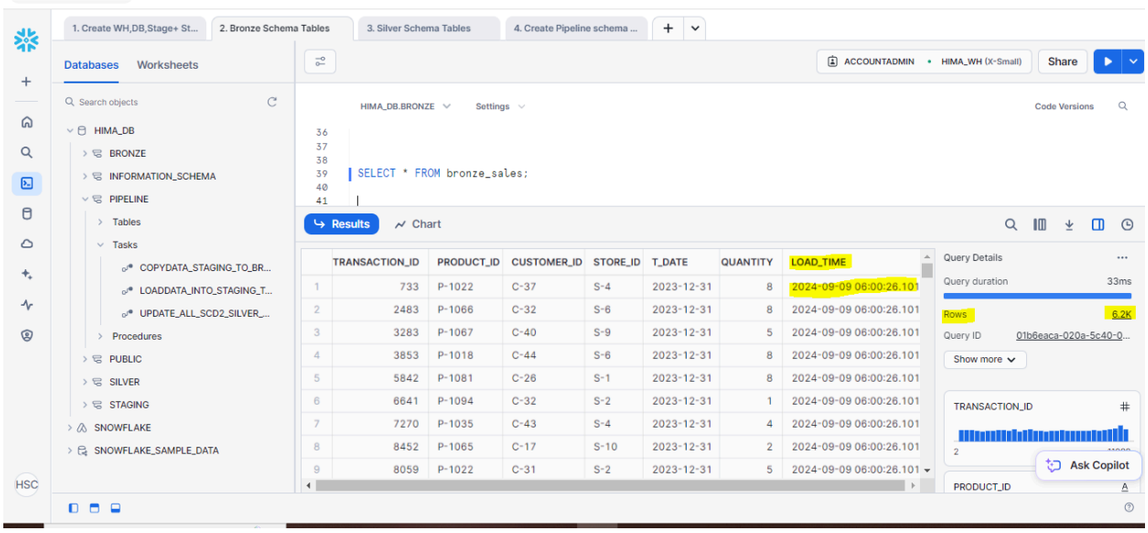
2. BRONZE_PRODUCTS table



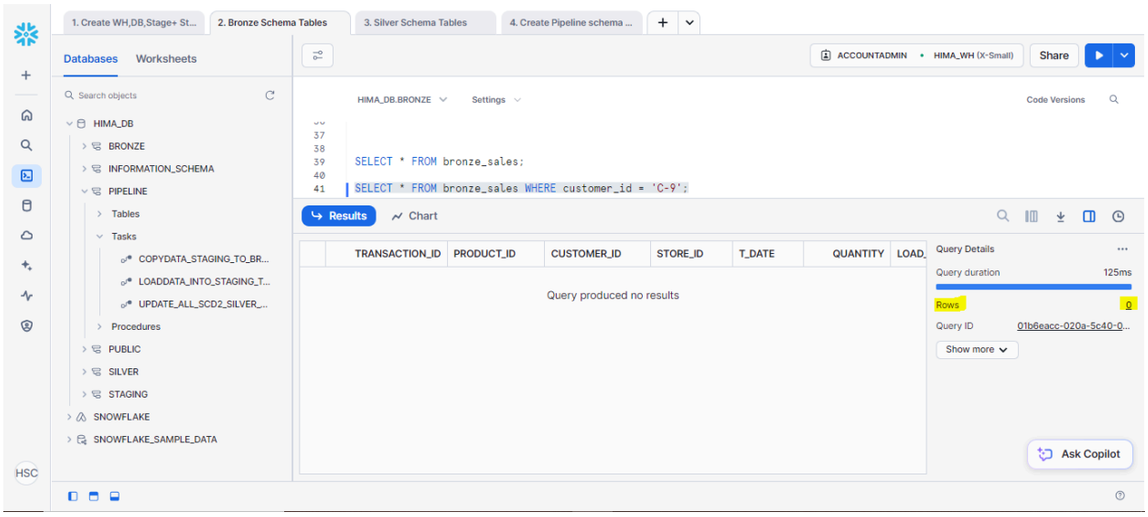
3. BRONZE_STORES table



4. BRONZE_SALES table



SELECT * FROM bronze_sales WHERE customer_id = 'C-9';



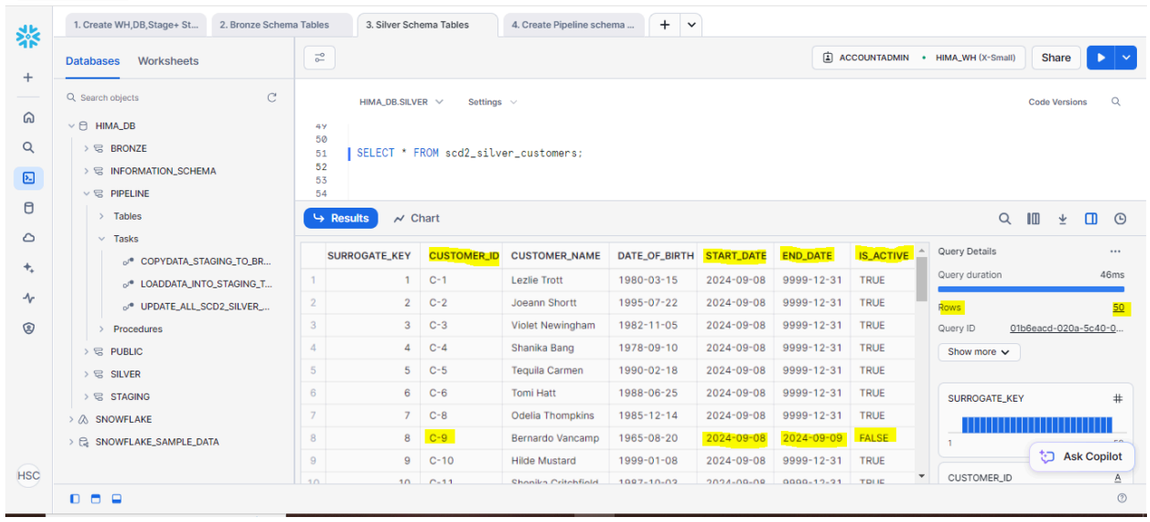
SILVER LAYER TABLES

1. SCD2_SILVER_CUSTOMERS table

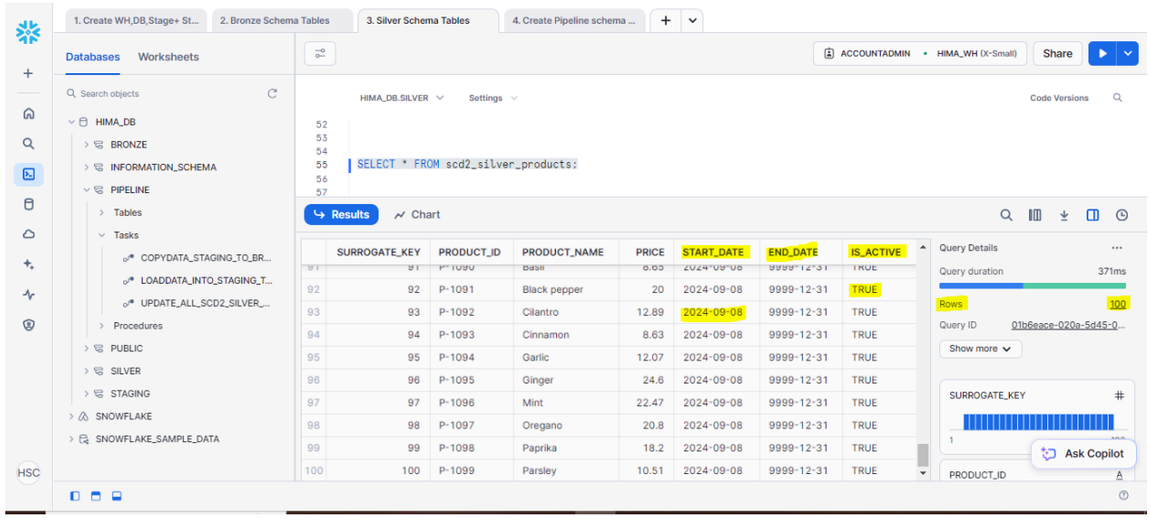
For example:

On Day 1, the `staging_customers` table contains a customer with Customer ID `C-9`. However, on Day 2, this Customer ID `C-9` is no longer present in the staging table.

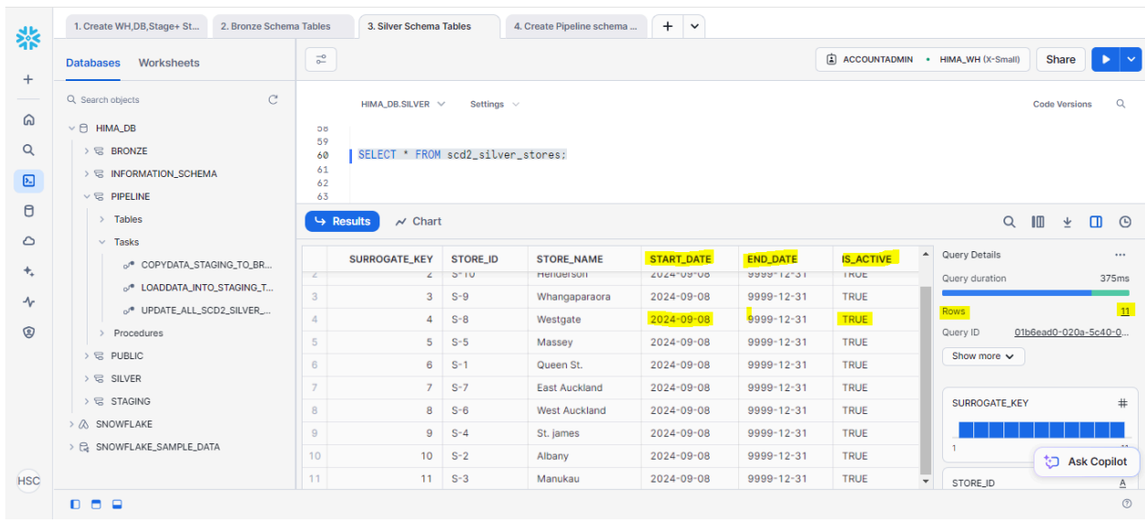
In the silver layer tables, which implement Slowly Changing Dimension Type 2 (SCD2), any data that is deleted, updated, or inserted in the base table is managed by marking the old record as inactive and the new record as active. Since Customer ID `C-9` is no longer present in the `staging_customers` table, the corresponding record in the `silver_customers` table is marked as inactive, reflecting the change in the staging layer.



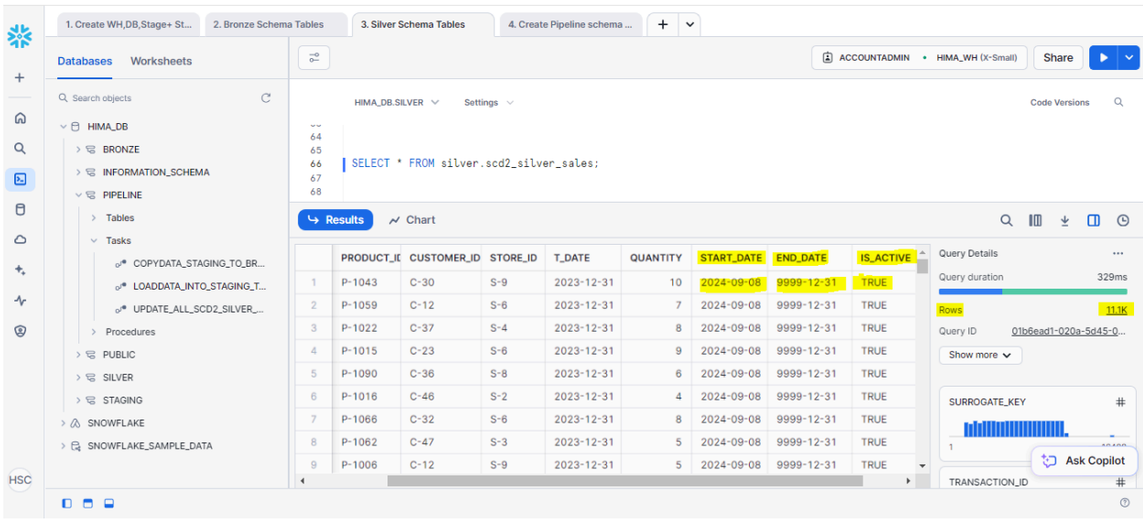
2. SCD2_SILVER_PRODUCTS table



3. SCD2_SILVER_STORES table



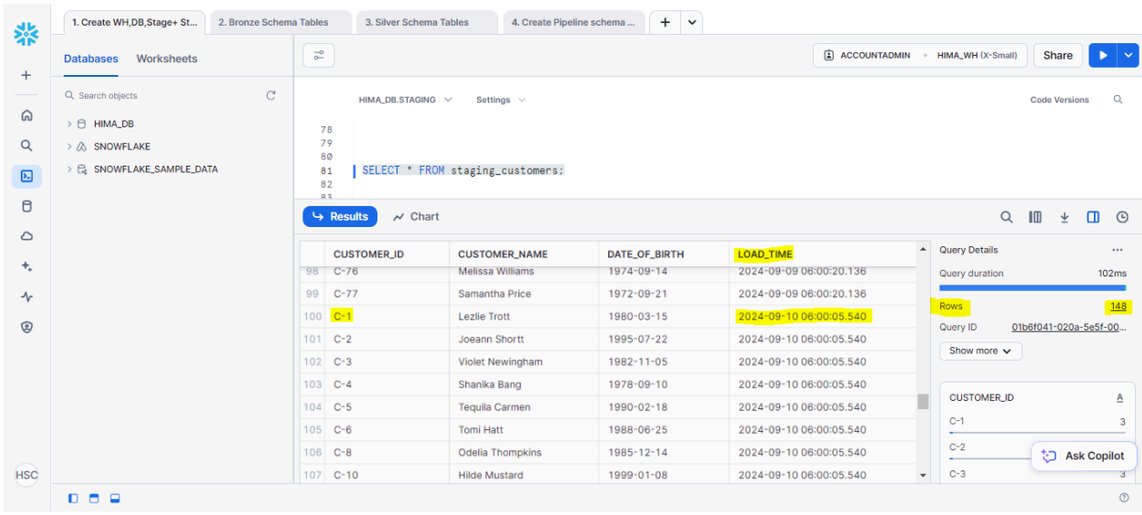
4. SCD2_SILVER_SALES table



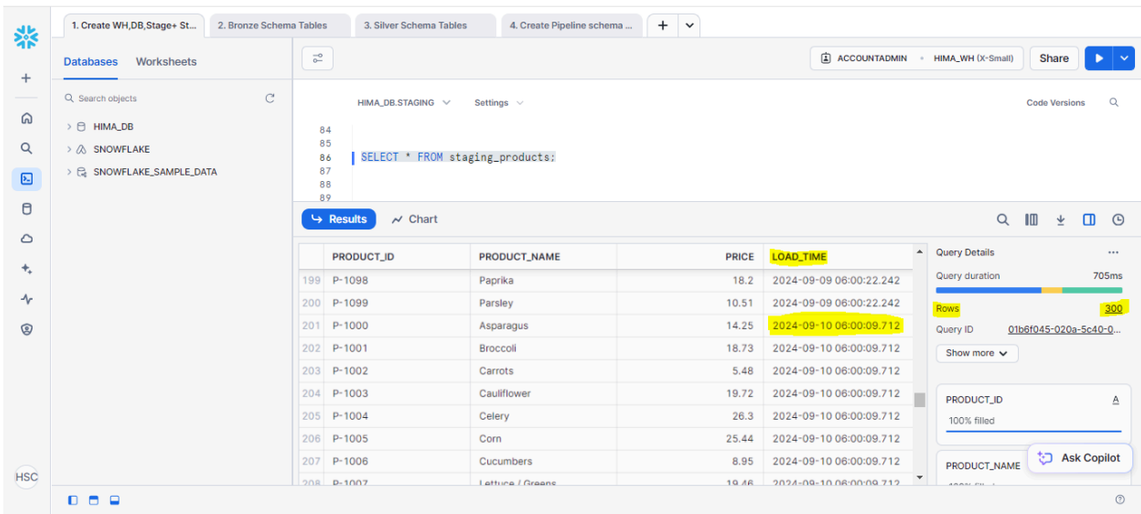
DAY 3

STAGING LAYER TABLES

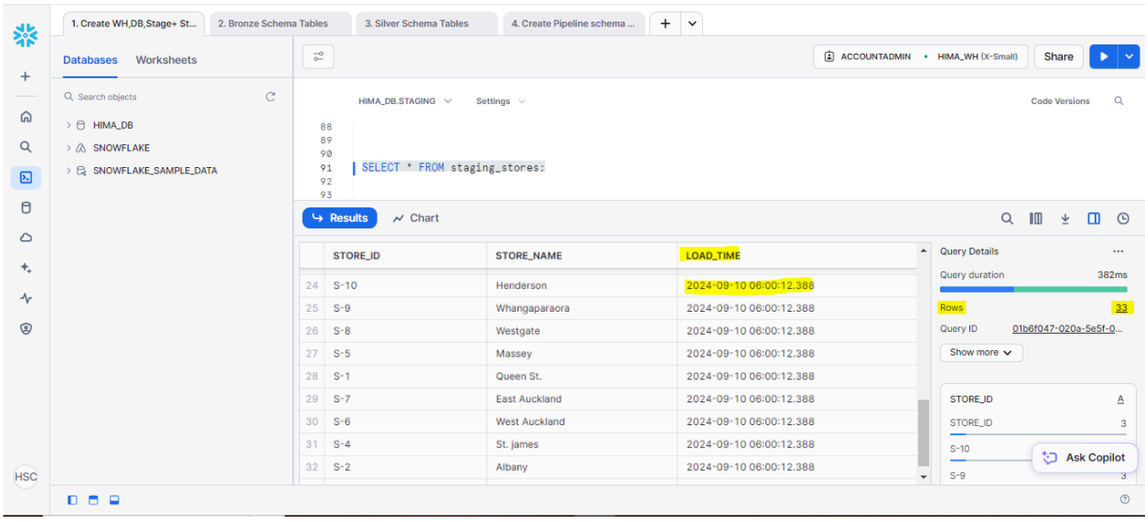
1. STAGING_CUSTOMERS table



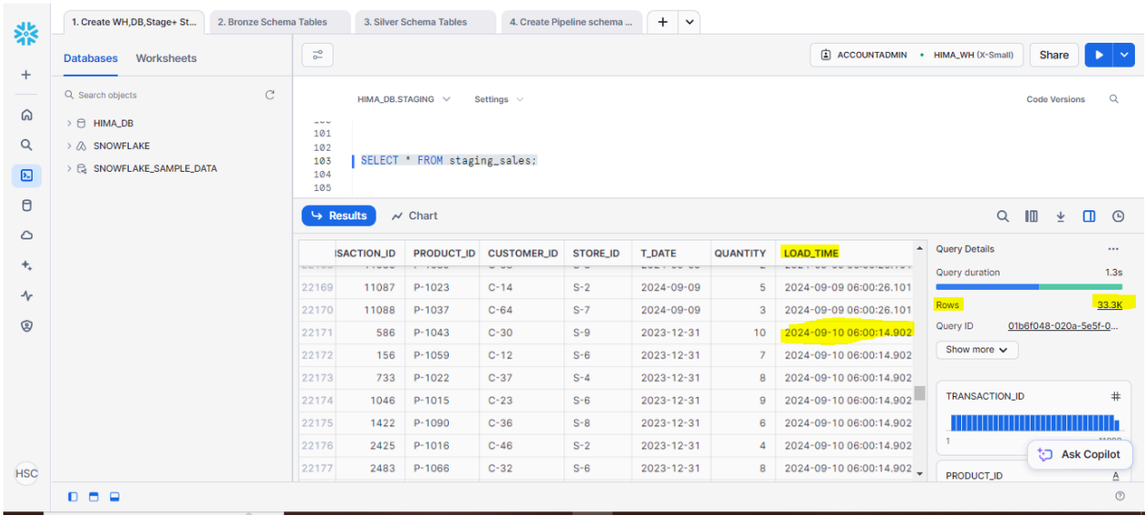
2. STAGING_PRODUCTS table



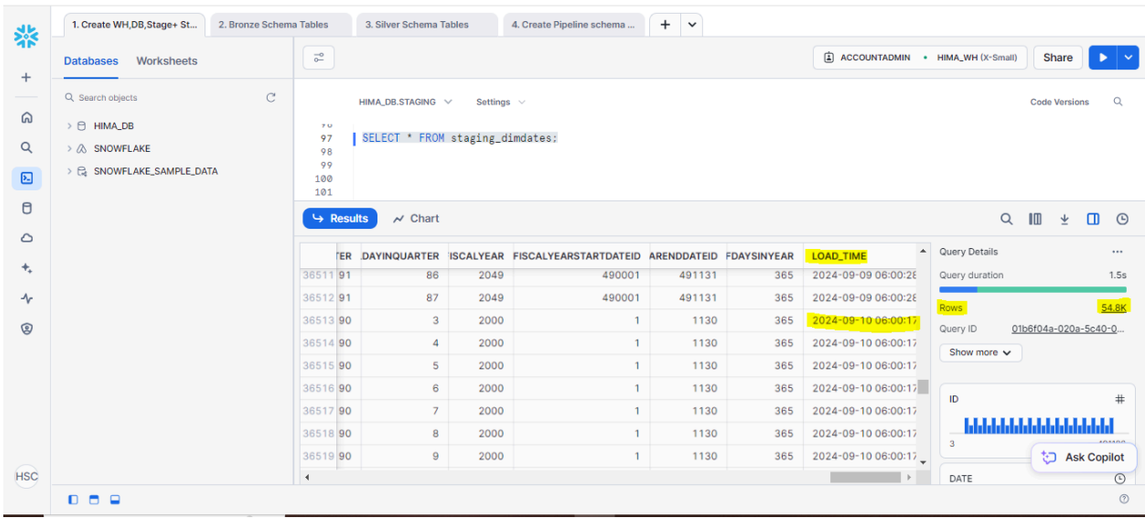
3. STAGING_STORES table



4. STAGING_SALES table

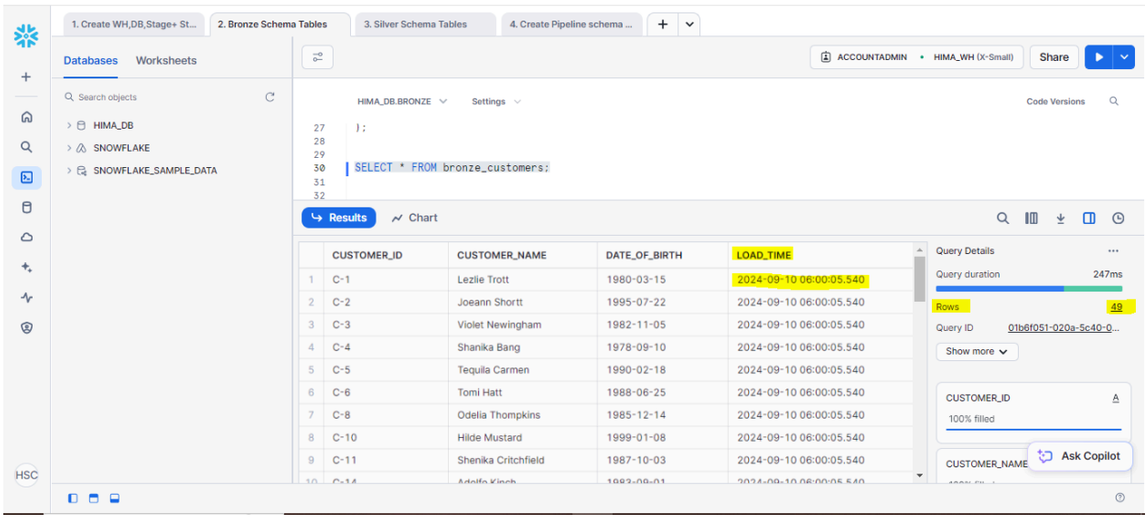


5. STAGING_DIMDATES table

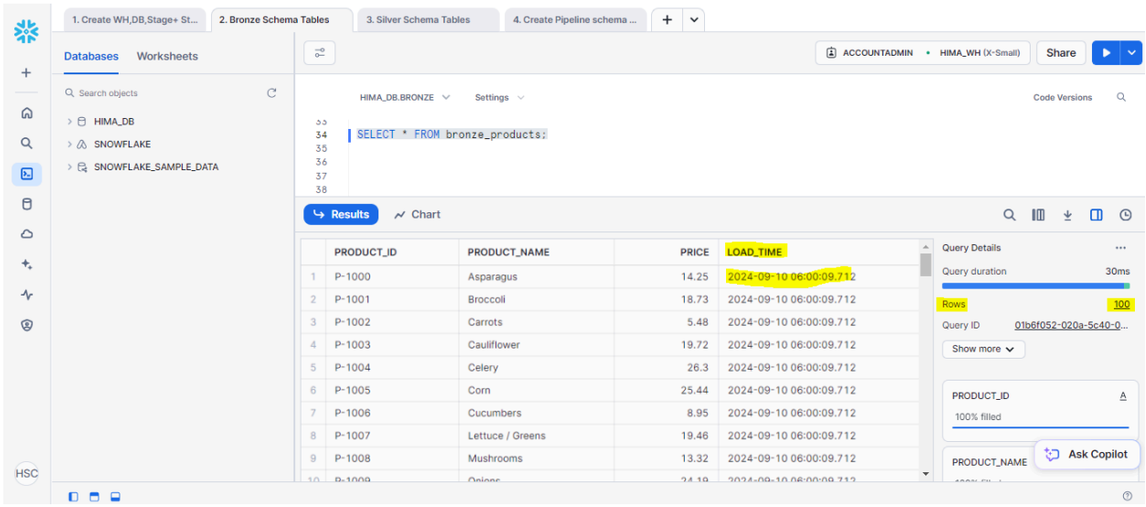


BRONZE LAYER TABLES

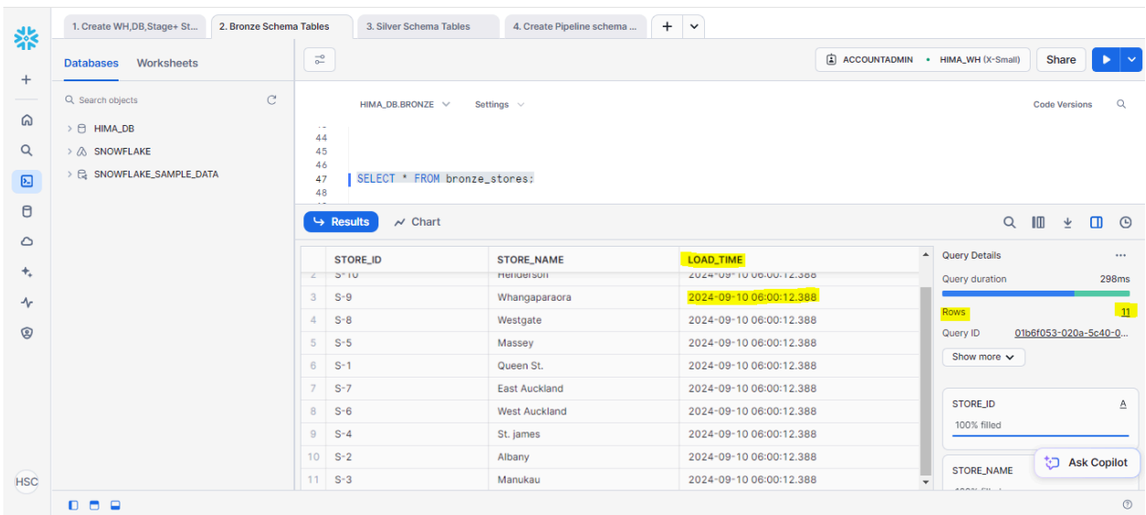
1. BRONZE_CUSTOMERS table



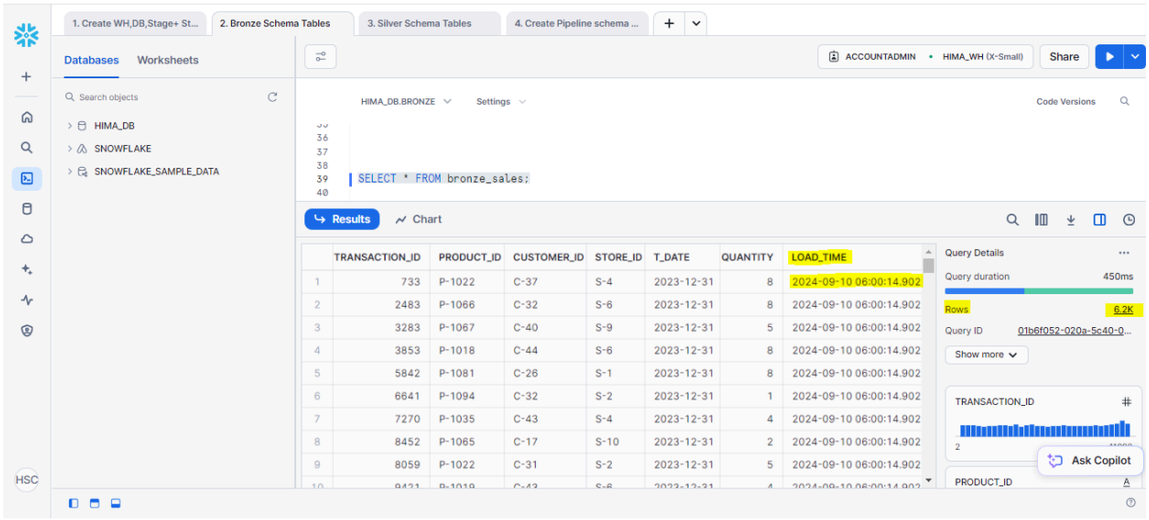
2. BRONZE_PRODUCTS table



3. BRONZE_STORES table

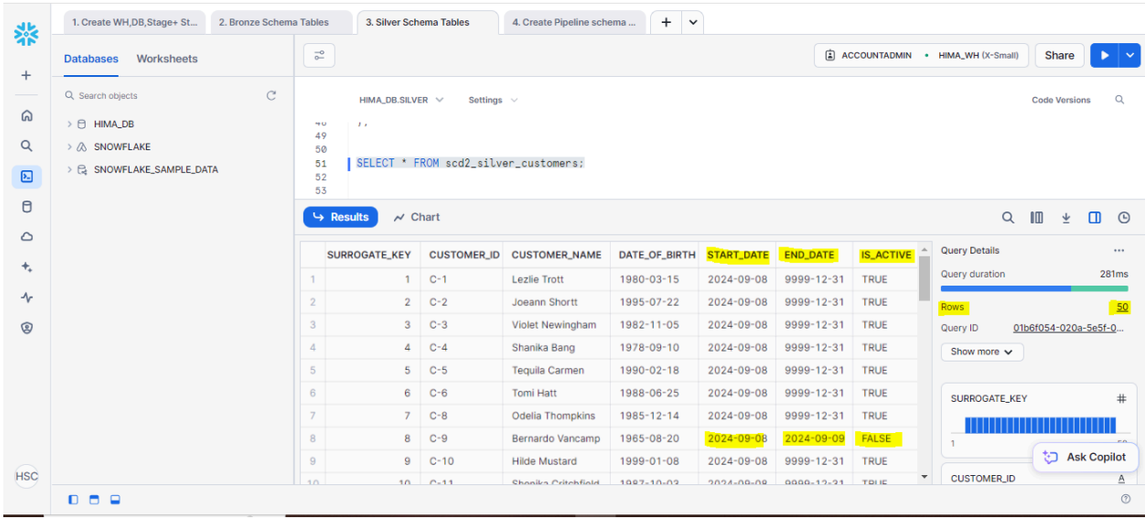


4. BRONZE_SALES table



SILVER LAYER TABLES

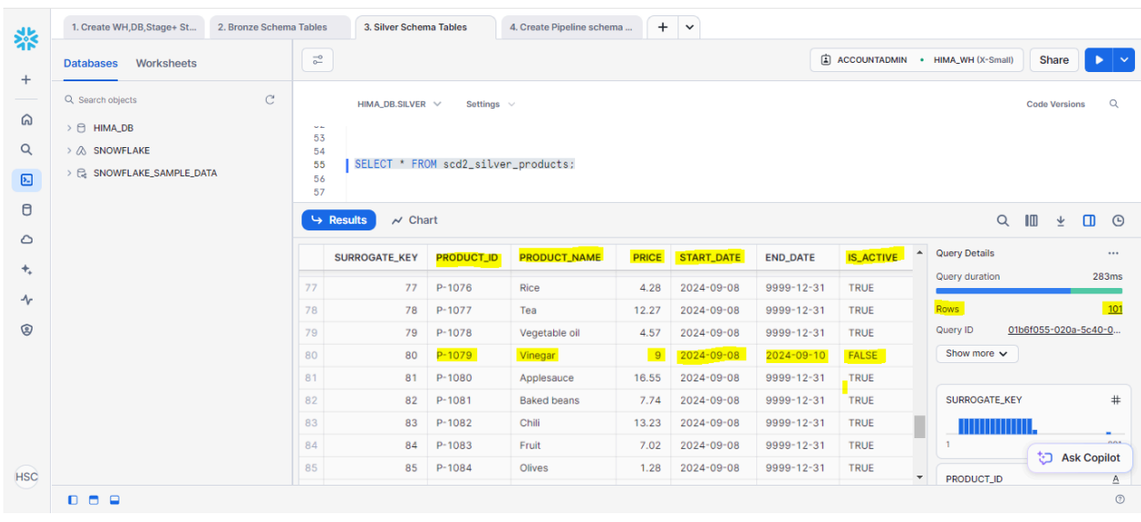
1. SCD2_SILVER_CUSTOMERS table

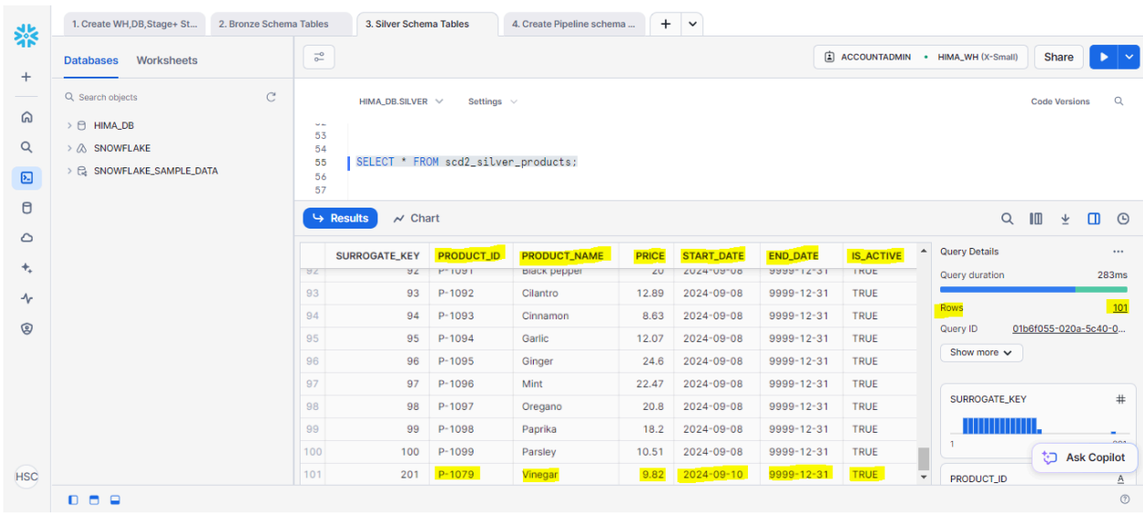


2. SCD2_SILVER_PRODUCTS table

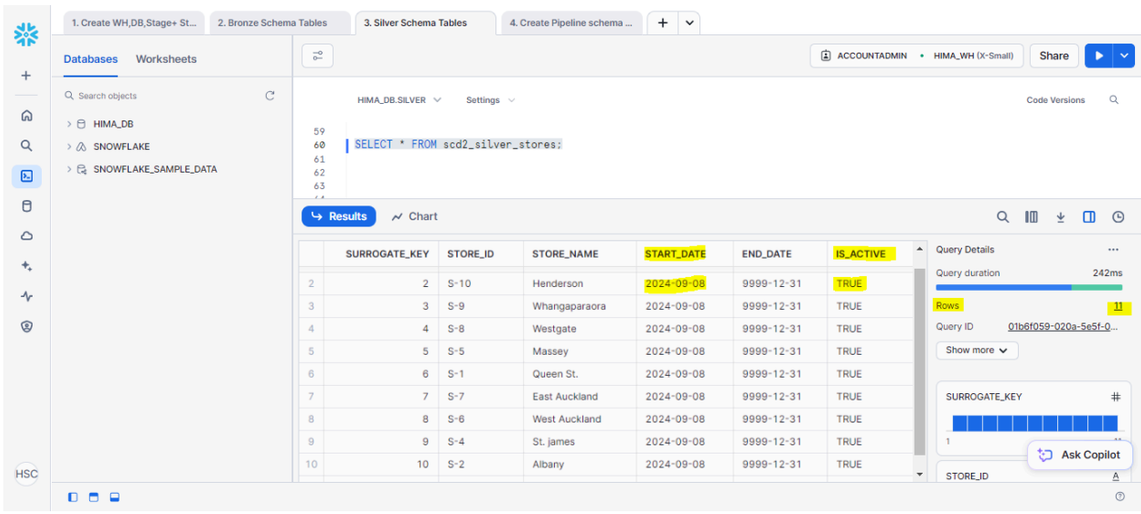
On Day 3, in the staging_products table, the price value for Product ID P-1079 was updated.

This change was reflected in the SCD2 silver table, where the old record was marked as inactive, and the updated record was inserted as active with appropriate start and end dates.

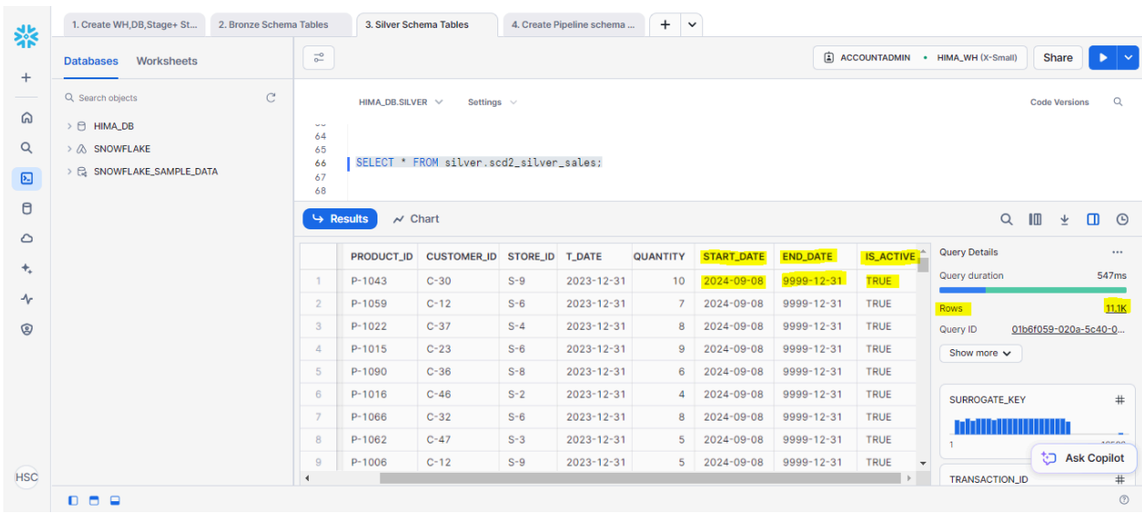




3. SCD2_SILVER_STORES table

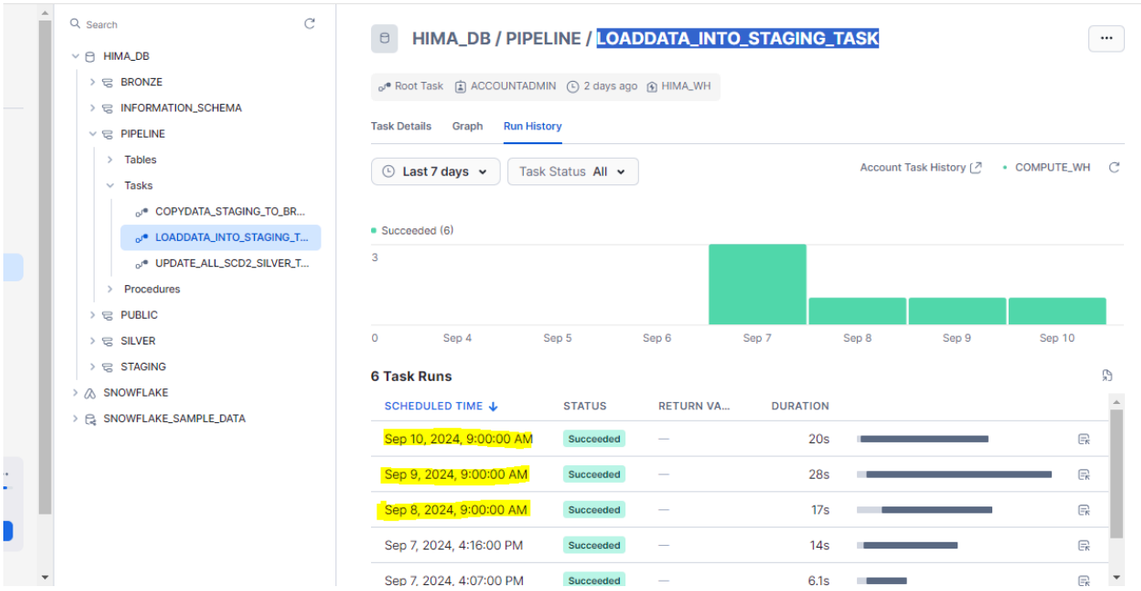


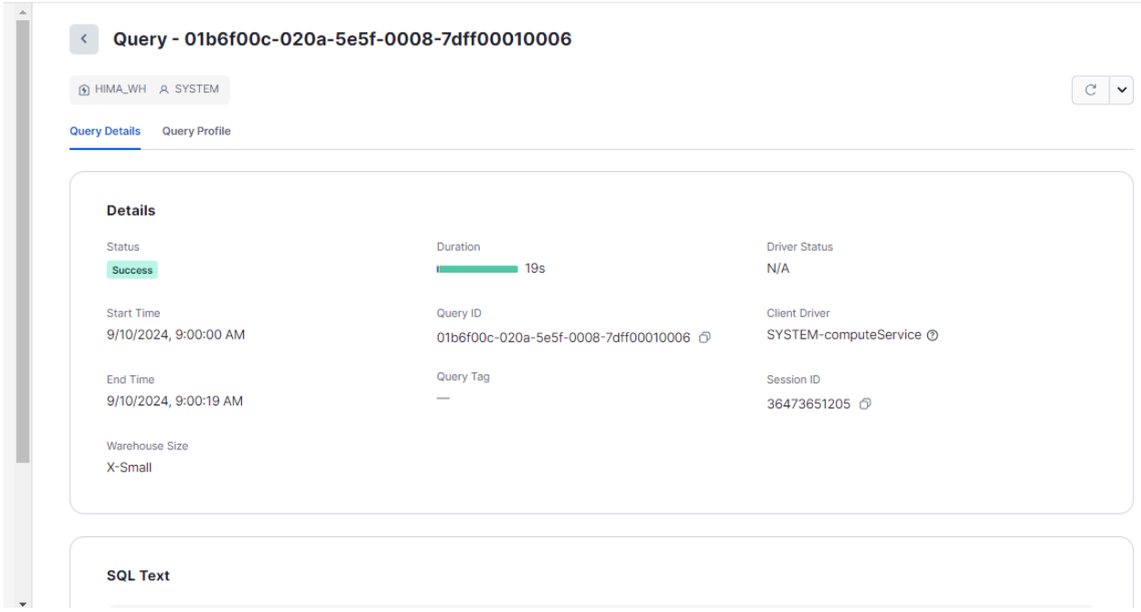
4. SCD2_SILVER_SALES table

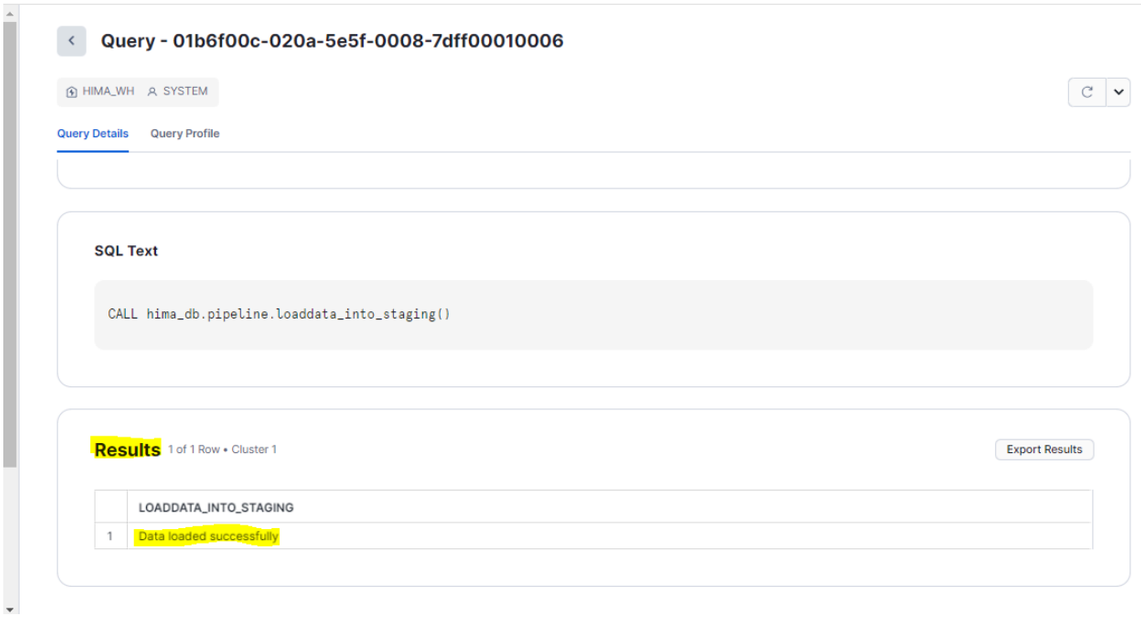


TASK Status for LOADDATA_INTO_STAGING_TASK over 3 days

This task is responsible for loading data into the staging tables. The status will indicate whether the task is currently running, completed, scheduled, or if there are any errors or issues during execution.

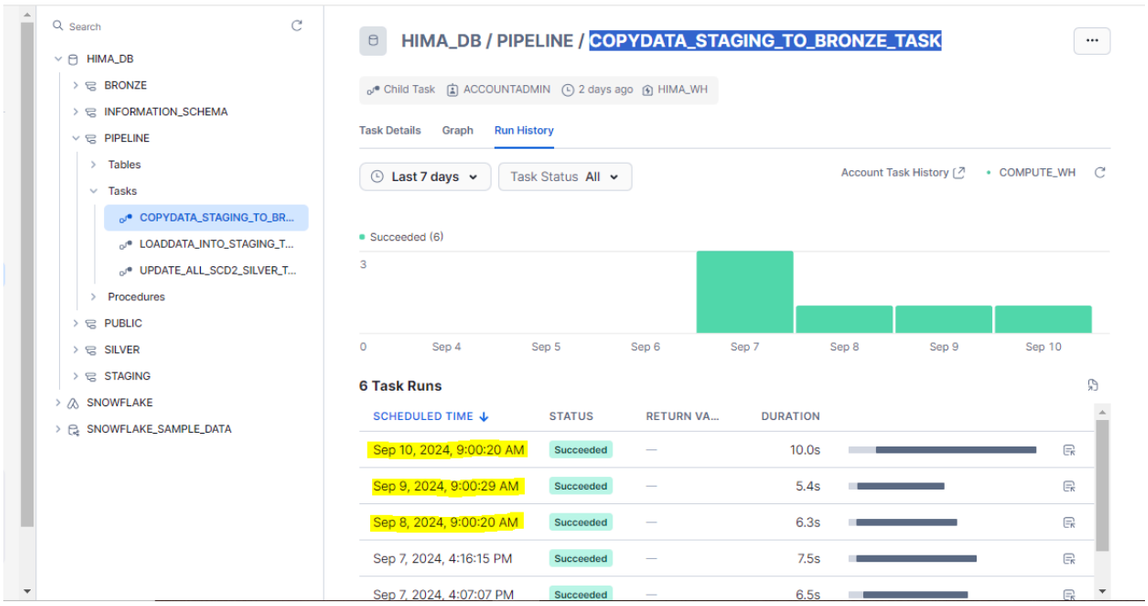


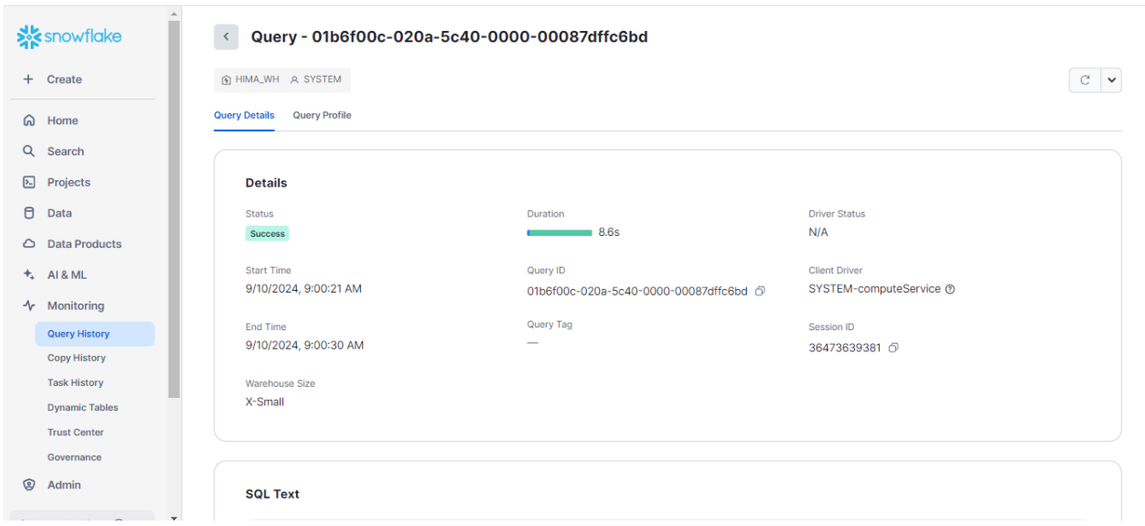


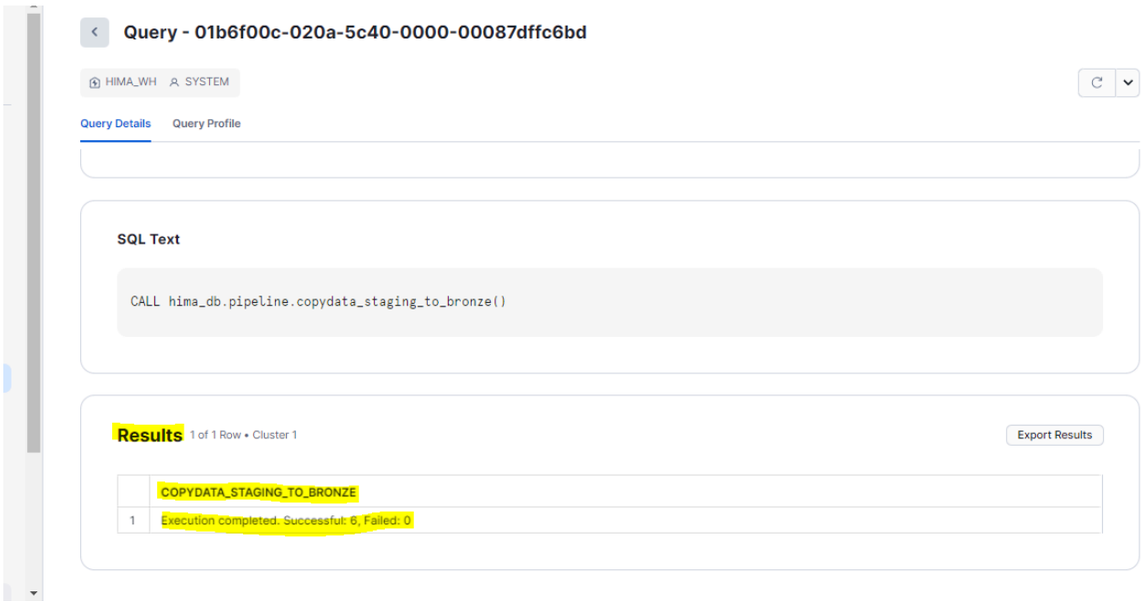


TASK Status for COPYDATA_STAGING_TO_BRONZE_TASK over 3 Days After LOADDATA_INTO_STAGING_TASK Execution. This task runs daily after the

LOADDATA_INTO_STAGING_TASK completes. The purpose of this task is to copy data from the staging tables to the bronze tables while enforcing the defined data quality rules.

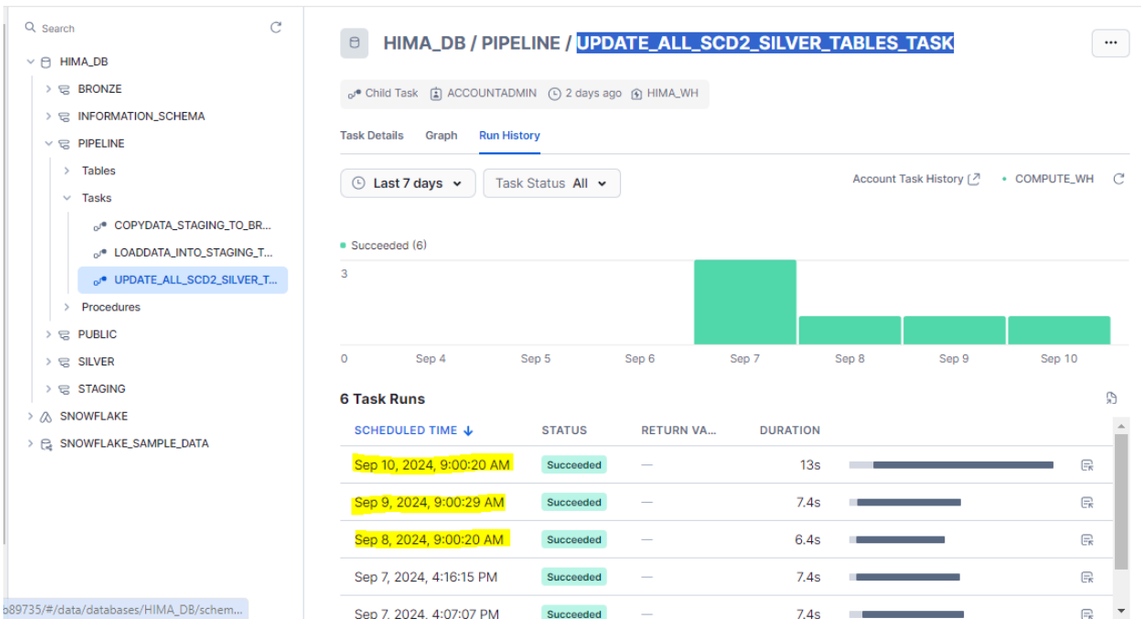


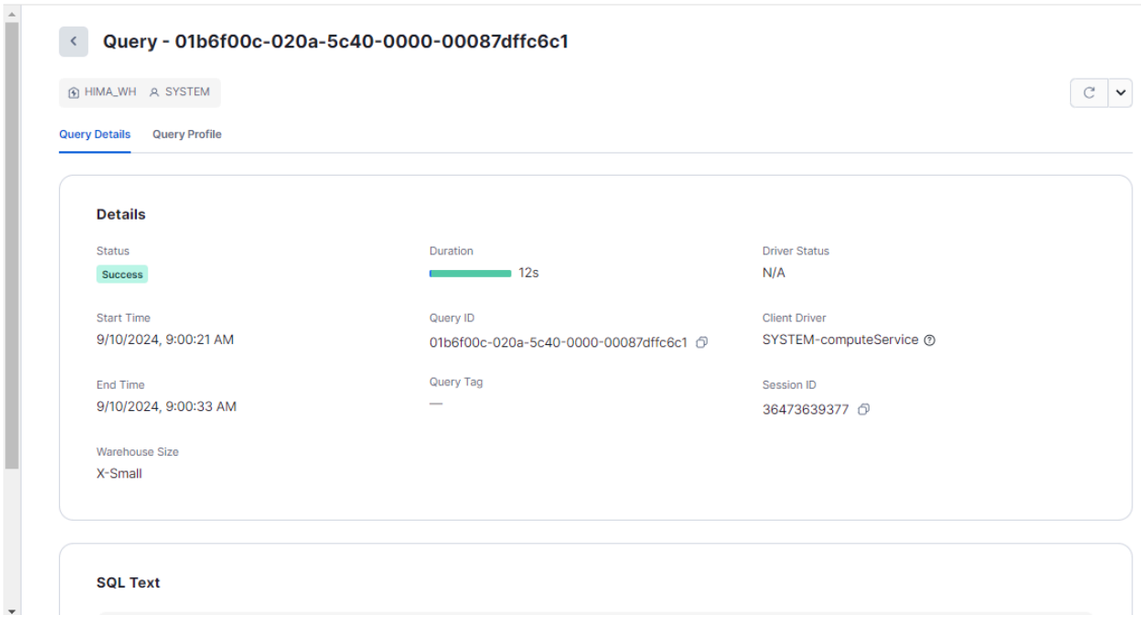


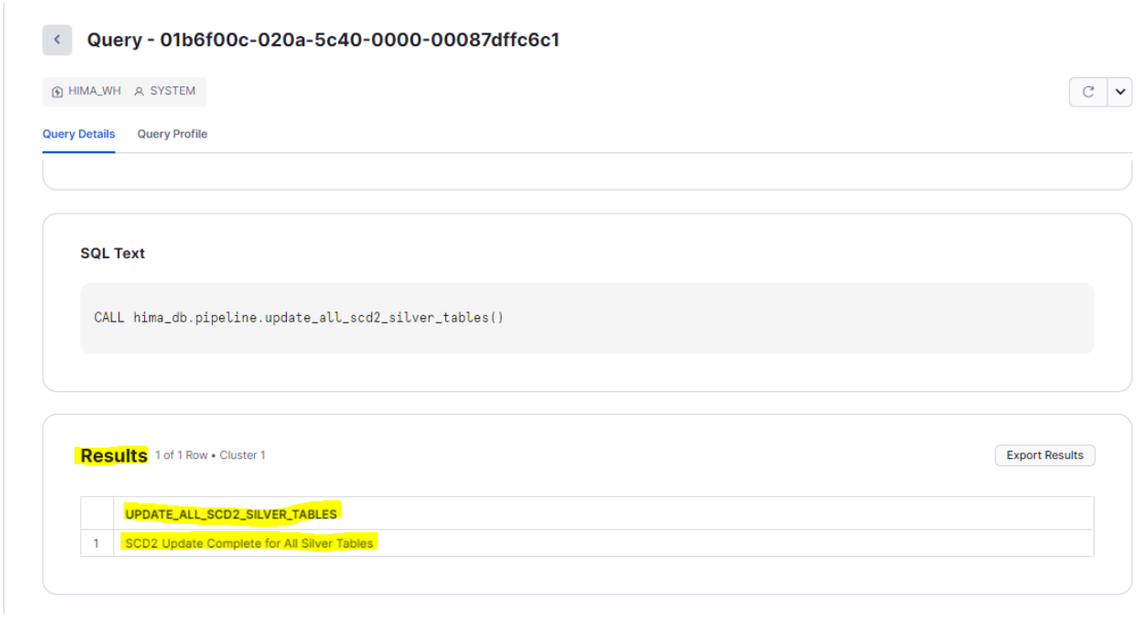


TASK Status for UPDATE_ALL_SCD2_SILVER_TABLES_TASK Over 3 Days. This task runs automatically every day, immediately following the completion of the

LOADDATA_INTO_STAGING_TASK. The task is designed to update the SCD Type 2 silver tables with the latest data from the base staging tables.







POWER BI

In Power BI, I developed an interactive sales dashboard utilizing data from the Silver schema (SCD Type 2 tables) to visualize key business metrics and trends. The data model integrated sales transactions with customer, product, and store details, allowing for historical analysis and current state tracking.

1. Data Merging and Enrichment: I enriched the sales data by merging the scd2_silver_sales table with the scd2_silver_customers, scd2_silver_products, and scd2_silver_stores tables, using foreign key relationships (such as CUSTOMER_ID, PRODUCT_ID, and STORE_ID) to bring in relevant details like CUSTOMER_NAME, PRODUCT_NAME, and STORE_NAME.

2. Slicers for Dynamic Filtering: Implemented a Price Range slicer using the PRICE field to filter products based on their prices.

Added a Time Range slicer using the T_DATE (transaction date) field, allowing dynamic filtering of sales transactions by date range.

3. KPI Cards: Created Total Users (Customers) KPI by counting distinct CUSTOMER_IDs, showing the total number of active users in the dataset.

Added a Total Products KPI by counting distinct PRODUCT_IDs, reflecting the number of unique products available.

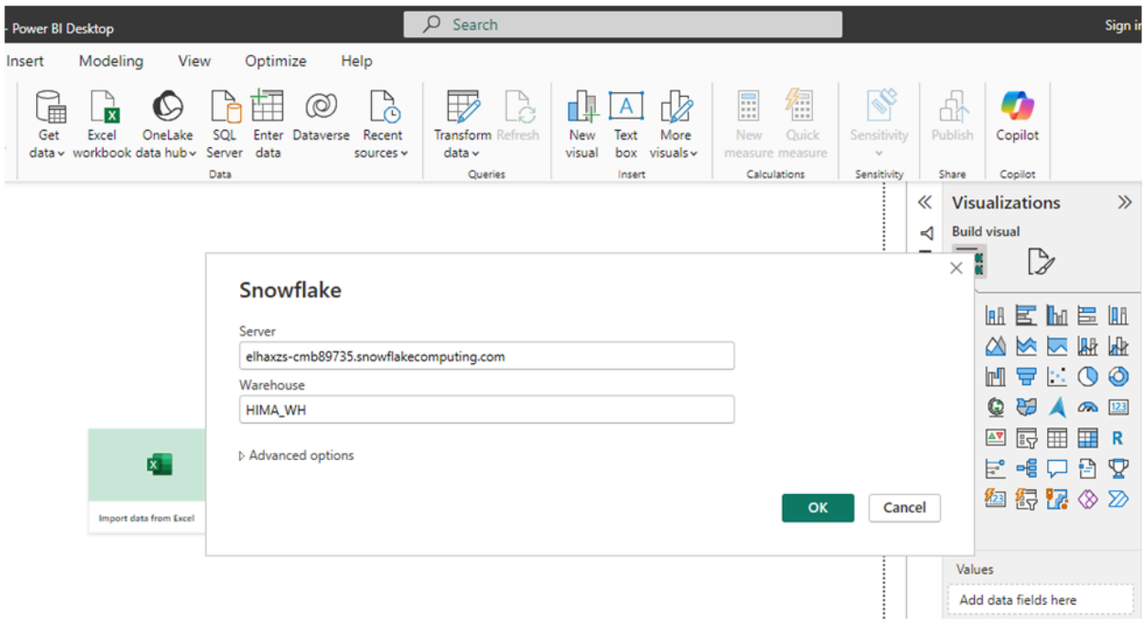
Additional KPI metrics such as Total Sales, Total Revenue, and Customers Left were added, utilizing the respective sales and customer data.

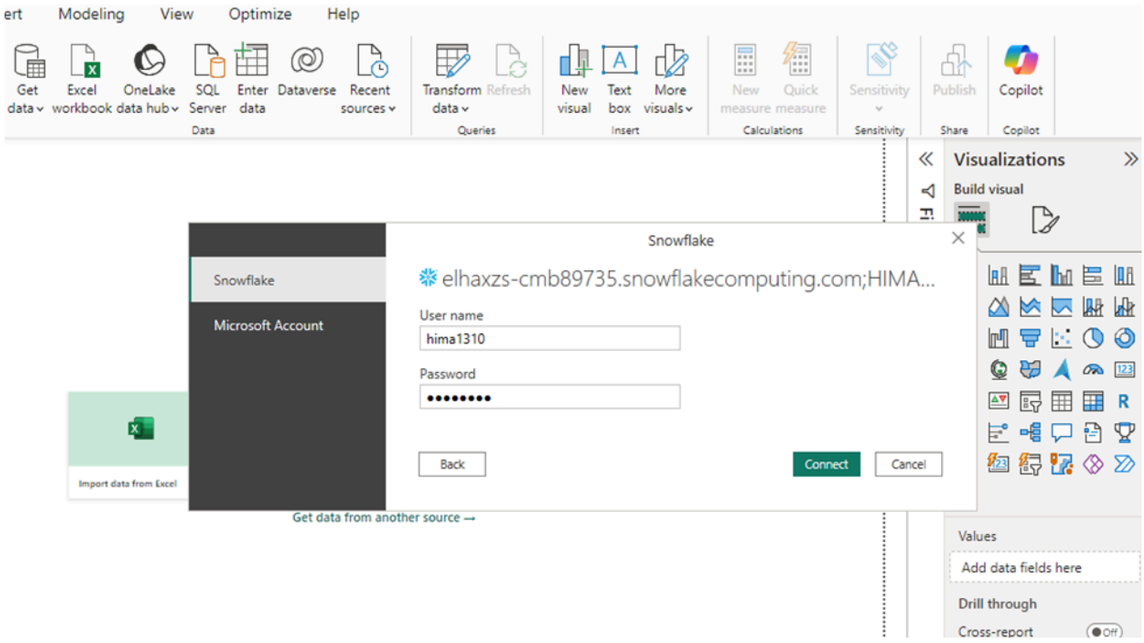
4. Top 10 Sold Products Table: Built a table to display the Top 10 Sold Products by sales quantity, using PRODUCT_NAME and QUANTITY fields. Applied a top 10 filter to showcase the best-performing products in terms of sales volume.

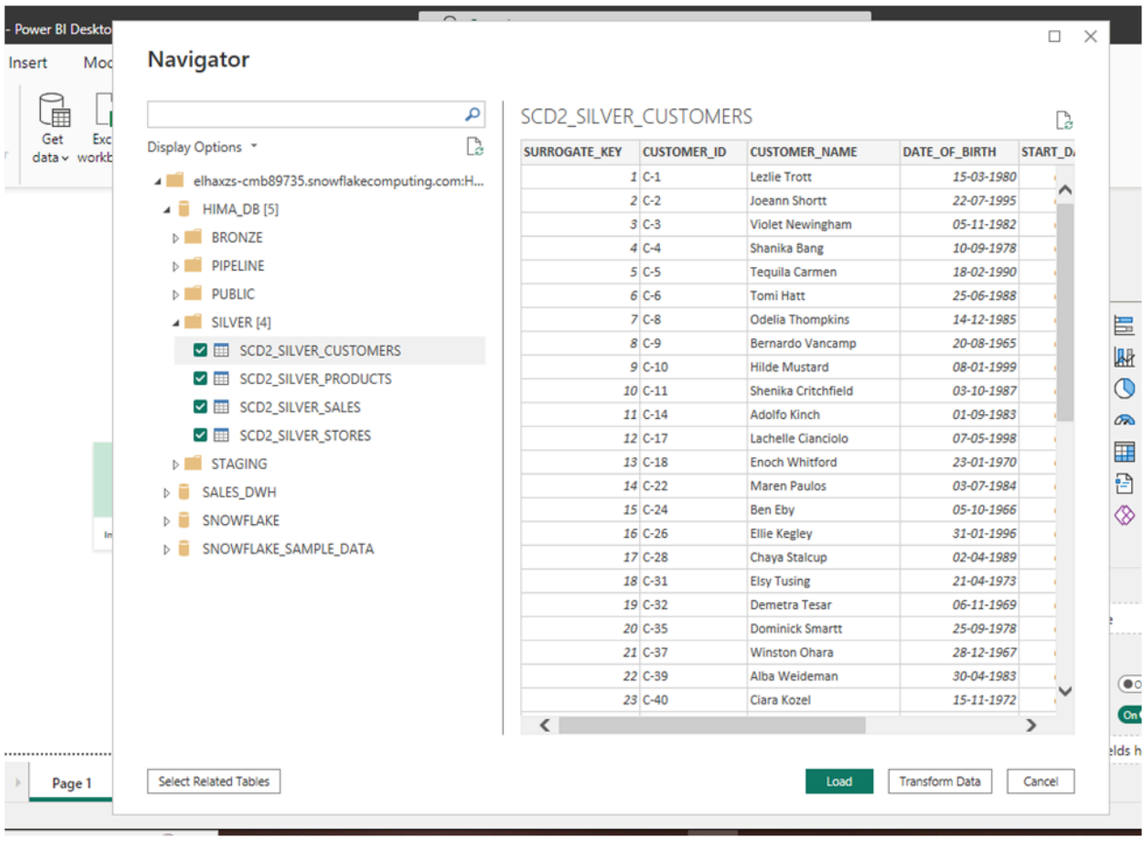
5. Bar Chart for Daily Sales: Created a Daily Sales Bar Chart, visualizing units sold per day. T_DATE was used on the X-axis to represent dates, and QUANTITY on the Y-axis to display the number of units sold. This chart provides insights into sales trends over time.

6. Table for Transaction Data: Added a Table Visualization to display individual transaction details. Fields such as CUSTOMER_NAME, PRODUCT_NAME, PRICE, T_DATE, and QUANTITY were included, allowing users to inspect and explore specific transactions in the dataset.

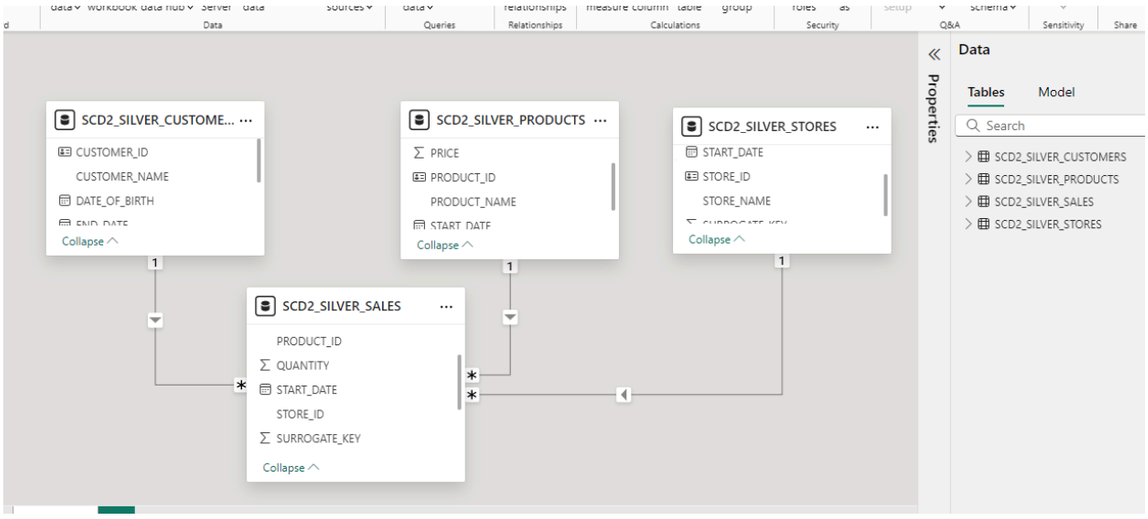
7. Data Integrity: Ensured that the data used in these visualizations adhered to the Silver schema's principles of historical and current state analysis, with Slowly Changing Dimension (SCD) Type 2 logic for capturing changes over time.



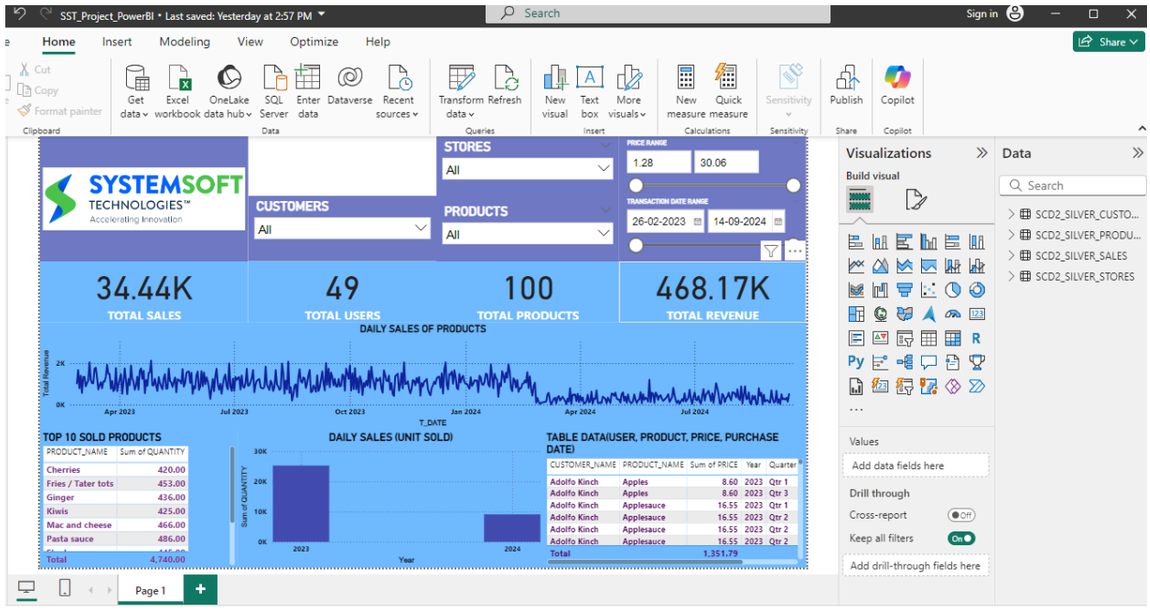


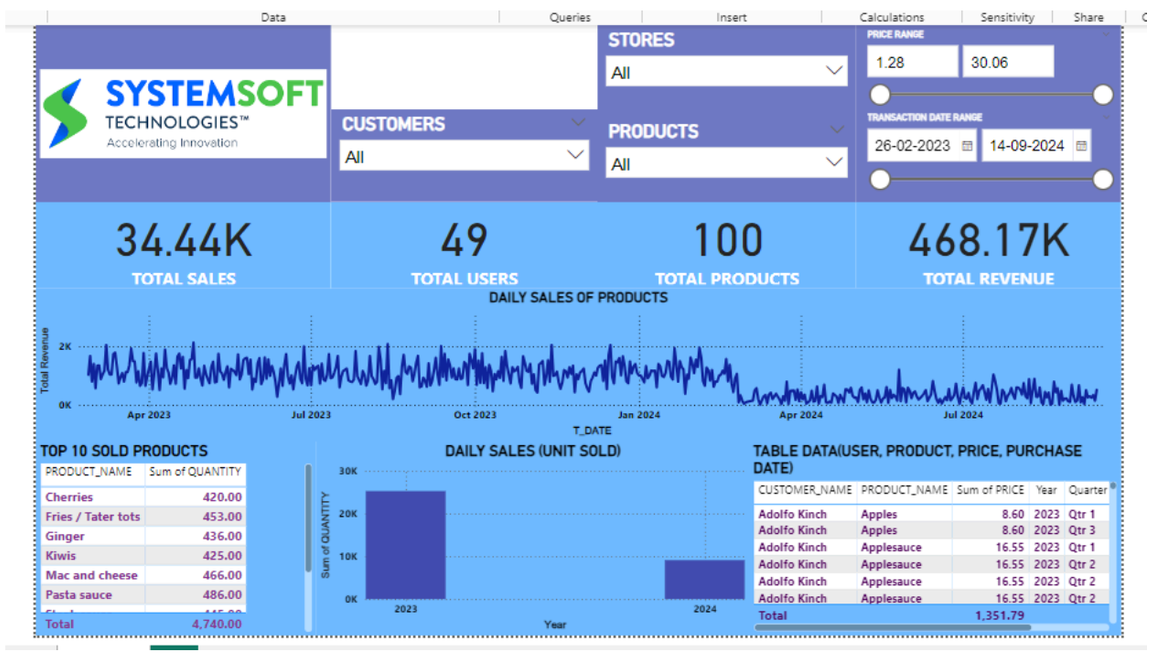

Navigator
Display Options
elhaxzs-cmb89735.snowflakecomputing.com:H...
HIMA_DB [5]
BRONZE
PIPELINE
PUBLIC
SILVER [4]
SCD2_SILVER_CUSTOMERS
SCD2_SILVER_PRODUCTS
SCD2_SILVER_SALES
SCD2_SILVER_STORES
STAGING
SALES_DWH
SNOWFLAKE
SNOWFLAKE_SAMPLE_DATA
SCD2_SILVER_CUSTOMERS
SURROGATE_KEY CUSTOMER_ID CUSTOMER_NAME DATE_OF_BIRTH START_DA
1 C-1 Lezlie Trott 15-03-1980
2 C-2 Joeann Shortt 22-07-1995
3 C-3 Violet Newingham 05-11-1982
4 C-4 Shanika Bang 10-09-1978
5 C-5 Tequila Carmen 18-02-1990
6 C-6 Tomi Hatt 25-06-1988
7 C-8 Odelia Thompkins 14-12-1985
8 C-9 Bernardo Vancamp 20-08-1965
9 C-10 Hilde Mustard 08-01-1999
10 C-11 Shenika Critchfield 03-10-1987
11 C-14 Adolfo Kinch 01-09-1983
12 C-17 Lachelle Cianciolo 07-05-1998
13 C-18 Enoch Whitford 23-01-1970
14 C-22 Maren Paulos 03-07-1984
15 C-24 Ben Eby 05-10-1966
16 C-26 Ellie Kegley 31-01-1996
17 C-28 Chaya Stalcup 02-04-1989
18 C-31 Elsy Tusing 21-04-1973
19 C-32 Demetra Tesar 06-11-1969
20 C-35 Dominick Smartt 25-09-1978
21 C-37 Winston Ohara 28-12-1967
22 C-39 Alba Weideman 30-04-1983
23 C-40 Ciara Kozel 15-11-1972
Page 1
Select Related Tables
Load
Transform Data
Cancel

Model View

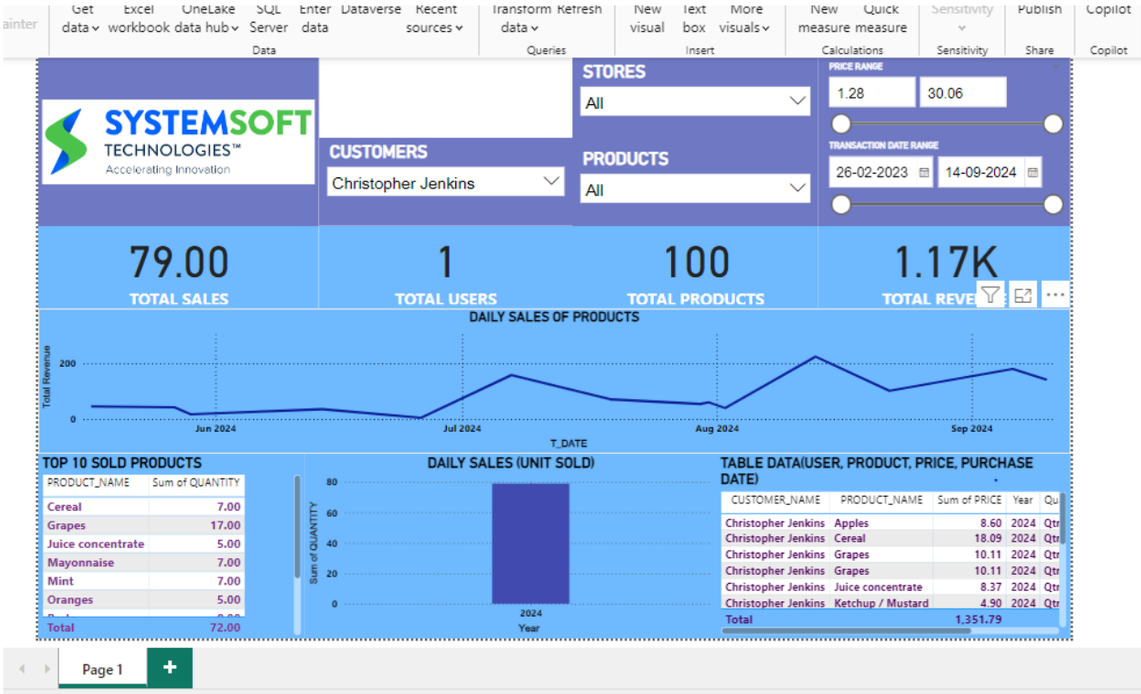

SCD2_SILVER_CUSTOME...
CUSTOMER_ID
CUSTOMER_NAME
DATE_OF_BIRTH
Collapse
SCD2_SILVER_PRODUCTS
PRICE
PRODUCT_ID
PRODUCT_NAME
Collapse
SCD2_SILVER_STORES
START_DATE
STORE_ID
STORE_NAME
Collapse
SCD2_SILVER_SALES
PRODUCT_ID
QUANTITY
START_DATE
STORE_ID
SURROGATE_KEY
Collapse
Properties
Data
Tables
Model
Search
SCD2_SILVER_CUSTOMERS
SCD2_SILVER_PRODUCTS
SCD2_SILVER_SALES
SCD2_SILVER_STORES

Report View





I applied a filter to the Power BI report to focus on a single customer. This dynamically updated all the visualizations, including KPIs, tables, and charts, to display data exclusively related to that customer. The transaction table now shows the selected customer's purchases, including product details, prices, transaction dates, and quantities. Other elements, such as the top sold products and daily sales chart, were also refined to reflect the specific customer's activity.



Adding Store and Product Details:

To provide a comprehensive view, we extended the visualizations to include details of products and stores associated with the selected customer. The transaction table now displays relevant product names and store names, reflecting the customer's purchases. KPIs and charts, such as those showing top-selling products and daily sales by store, are dynamically updated to highlight metrics specifically for the selected customer.