D597 MKN1 Task 1

Relational Database Design and Implementation

Shanay Murdock
smurd32@wgu.edu
011377935

D597 Data Management
MS, Data Analytics

A. Scenario Context

A1. Business Problem

EcoMart is an online marketplace focused on fostering sustainability and environmental consciousness, serving as a hub for consumers seeking ethically sourced, sustainable, and eco-friendly products across a variety of categories.

Currently, and despite its range of metrics, EcoMart is using one table to hold all its data. The data is not optimized for storage or analysis and EcoMart is seeking assistance in designing and optimizing a relational database model that will allow it flexibility as new products and metrics are introduced and as the business grows and scales.

A normalized relational database is the best solution for EcoMart, addressing the company's concerns over flexibility, scalability, performance, simultaneous users, reliability, security, compliance, monitoring, maintenance, and analytics needs.

A2. Data Structure

To address the above concerns, a normalized relational database solution is proposed via the following list of tables and their corresponding column names:

- `region`
    - region_id
    - region_name
- `country`
    - country_id
    - country_name
    - region_id (references `region` table)
- `item_type`
    - item_type_id
    - item_type_name
- `sales_channel`
    - sales_channel_id
    - sales_channel_name
- `orders`
    - order_id
    - order_priority
    - order_date
    - ship_date
    - country_id (references `country` table)
    - item_type_id (references `item_type` table)
    - sales_channel_id (references `sales_channel` table)
- `sales_details`

- sales_detail_id
- order_id (references `orders` table
- units_sold
- unit_price
- unit_cost
- total_revenue
- total_cost
- total_profit

The structure of these tables addresses the functional and non-functional needs of the database. It minimizes redundancy and respects the principles of normalization, ensuring that each piece of data is stored only once and relationships are maintained through foreign keys. Additional constraints have been added to ensure that necessary names cannot be null or empty strings.

The schema will be described in more detail in section B.

A3. Database Justification

The original data was provided in the format of a Comma-Separated Values (CSV) file, which presents several significant challenges. Depending on the environment in which it is used (e.g., Excel), the spreadsheet may not be capable of handling multiple users concurrently, and there is no guarantee that a user is working with the latest version.

Version control in spreadsheets is inferior to that of databases and other dedicated version control tools such as Git. Data can easily be modified in a CSV or spreadsheet with minimal constraints, failing to ensure data integrity. Furthermore, CSV data is susceptible to data loss in the event of a software or computer crash, resulting in substantial and costly data loss.

It is also unknown if the data is being stored in an Excel file or if the data is an export of a denormalized database. Spreadsheets are eventually limited by the amount of data they can hold (in this case, rows) and will stop being a manageable way of storing data based on the projections of EcoMart's growth.

Given the above issues, a relational database is the most suitable solution for EcoMart. It provides structured transactional and inventory data that scales with large amounts of new data while preserving data relationships and integrity. This model facilitates rapid and efficient data retrieval, updates, and a structured approach to adding new product and transactional data as new products are added to inventory and product sales and returns are processed. The relational database is designed for large volumes of data, multiple simultaneous users, and data integrity without compromising performance. Furthermore, relational databases can incorporate constraints that enforce specific data formats and ensure compliance with these constraints.

The PostgreSQL database environment also allows the company to create database roles and role permissions that can be granted or revoked, maintaining security. PostgreSQL also provides built-in encryption tools to keep data safe from breaches.

A4. Data Usage

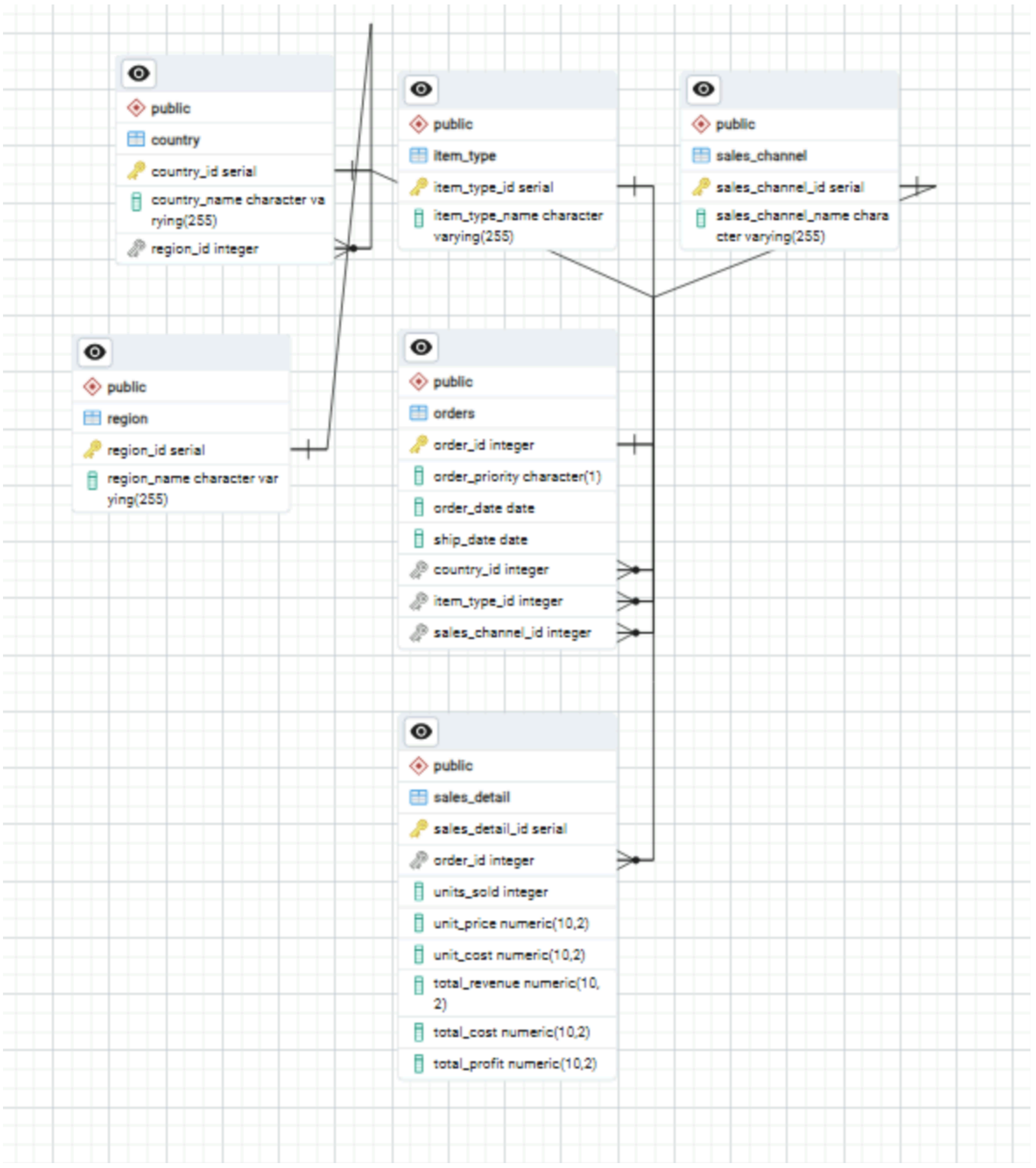
The proposed database solution will relationally store data, allowing tables to be joined together based on primary and foreign key connections. This keeps tables as precise as possible, joining data only where relevant.

The database solution will also feature predetermined data types and constraints, ensuring all existing data, in addition to any new data, is entered consistently. This helps with descriptive and predictive analysis based on current data and readies the databases to expand with EcoMart's growing needs.

The updated database structure will also allow users to interact with the data in ways relevant to their roles. For those who can and should be able to insert, update, remove data, or create temporary tables, permissions will be granted to allow them to make changes. For those whose roles strictly center around analyzing the data, role permissions can be granted to allow them to query the data without access to change or delete any data. These roles help ensure that EcoMart can protect its data through enhanced security and by ensuring data is kept as long as necessary for compliance and reporting.

B. Data Model

```sql
-- Create `region` table
CREATE TABLE region (
    region_id SERIAL PRIMARY KEY,
    region_name VARCHAR(255) NOT NULL,
    -- Add constraint
    CONSTRAINT "region_name_not_empty" CHECK (LENGTH(TRIM(region_name)) >=
0)
);

-- Create `country` table
CREATE TABLE country (
    country_id SERIAL PRIMARY KEY,
    country_name VARCHAR(255) NOT NULL,
    region_id INT,
    -- Add constraint
    CONSTRAINT "country_name_not_empty" CHECK (LENGTH(TRIM(country_name))
>= 0),
    -- Add foreign key references
    FOREIGN KEY (region_id) REFERENCES region(region_id)
);

-- Create `item_type` table
CREATE TABLE item_type (
    item_type_id SERIAL PRIMARY KEY,
    item_type_name VARCHAR(255) NOT NULL,
    -- Add constraint
    CONSTRAINT "item_type_name_not_empty" CHECK
(LENGTH(TRIM(item_type_name)) >= 0)
);

-- Create `sales_channel` table
CREATE TABLE sales_channel (
    sales_channel_id SERIAL PRIMARY KEY,
    sales_channel_name VARCHAR(255) NOT NULL,
  -- Add constraint
    CONSTRAINT "channel_name_not_empty" CHECK
(LENGTH(TRIM(sales_channel_name)) >= 0)
);

-- Create `orders` table
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    order_priority CHAR(1),
    order_date DATE,
    ship_date DATE,
    country_id INT,
    item_type_id INT,
    sales_channel_id INT,
    -- Add foreign key references
    FOREIGN KEY (country_id) REFERENCES country(country_id),
    FOREIGN KEY (item_type_id) REFERENCES item_type(item_type_id),
    FOREIGN KEY (sales_channel_id) REFERENCES
sales_channel(sales_channel_id)
);

-- Create `sales_details`
CREATE TABLE sales_detail (
    sales_detail_id SERIAL PRIMARY KEY,
    order_id INT,
    units_sold INT,
    unit_price DECIMAL(10, 2),
    unit_cost DECIMAL(10, 2),
    total_revenue DECIMAL(10, 2),
    total_cost DECIMAL(10, 2),
    total_profit DECIMAL(10, 2),
    -- Add foreign key references
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```



C. Objects & Storage

Because of the primary and foreign key relationships outlined in the Data Definition Language (DDL) schema listed above, it would be ideal to import the data from the CSV into a temporary table and then transfer the data to each table where the relationships can be established and maintained, as opposed to importing each column using the pgAdmin's CSV import tool.

To do this, a temporary table will need to be made with the intention of dropping it once the data is imported and merged to the appropriate tables.

```sql
-- Create a temporary table for import
CREATE TABLE bad_sales_records (
    region VARCHAR(255),
    country VARCHAR(255),
    item_type VARCHAR(255),
    sales_channel VARCHAR(255),
    order_priority CHAR(1),
    order_date DATE,
    order_id INT,
    ship_date DATE,
    units_sold INT,
    unit_price DECIMAL(10, 2),
    unit_cost DECIMAL(10, 2),
    total_revenue DECIMAL(10, 2),
    total_cost DECIMAL(10, 2),
    total_profit DECIMAL(10, 2)
);
```

The data types chosen for these columns will be further explained in the Relational Database Schema portion of this section.

The following code imports the data into PostgreSQL from the CSV as is. The Implementation section will cover the Data Manipulation Language (DML) needed to transfer the data from the `bad_sales_records` tables to the normalized schema and will cover dropping the `bad_sales_records` table.

```sql
-- Import the CSV data to the temporary table
COPY bad_sales_records (
    region, country, item_type, sales_channel, order_priority,
    order_date, ship_date, units_sold, unit_price,
    unit_cost, total_revenue, total_cost, total_profit
)
FROM "C:\WGU\D597\Task 1\Scenario 2.csv"
DELIMITED ','
CSV HEADER;
```

Relational Database Schema

Region table

- `region_idSERIAL PRIMARY KEY`
  - The Region ID is a primary key that automatically increments each additional row. This primary key can be referenced by the `country` table.
- `region_name VARCHAR(255) NOT NULL`
  - The Region Name is a unique name for each region and countries are each paired with a region. It is a varying character type that can receive up to 255 characters.
- `CONSTRAINT "region_name_not_empty" CHECK (LENGTH(TRIM(region_name)) >= 0)`
  - This constraint ensures that a blank value or empty string can't be used as a valid Region Name.

Country table

- `country_id SERIAL PRIMARY KEY,`
  - The Country ID is a primary key that uniquely identifies a country name and automatically increments each added row. This primary key can be referenced by the `orders` table.
- `country_name VARCHAR(255) NOT NULL,`
  - The Country Name is a unique name for each country. It is a varying character type that can receive up to 255 characters.
- `region_id INT,`
  - The Region ID is a foreign key associated with the attribute of the same name in the Regions table, linking the two tables together. It is of an integer type so that it can match the type provided by the SERIAL keyword.
- `CONSTRAINT "country_name_not_empty" CHECK (LENGTH(TRIM(country_name)) >= 0),`
  - This constraint ensures that a blank value or empty string can't be used as a valid Country Name.
- `FOREIGN KEY (region_id) REFERENCES region(region_id)`
  - This establishes the relationship of the region_id key as a foreign key to the primary key in the `region` table.

Item Type table

- `item_type_id SERIAL PRIMARY KEY,`
  - The Item Type ID is a primary key that uniquely identifies an item type and automatically increments each added row. This primary key can be referenced by the `orders` table.
- `item_type_name VARCHAR(255) NOT NULL,`
  - The Item Type Name is a unique name for each item type. It is a varying character type that can receive up to 255 characters.
- `CONSTRAINT "item_type_name_not_empty" CHECK (LENGTH(TRIM(item_type_name)) >= 0)`
  - This constraint ensures that a blank value or empty string can't be used as a valid Item Type Name.

Sales Channel table

- `sales_channel_id SERIAL PRIMARY KEY,`
  - The Sales Channel ID is a primary key that uniquely identifies an item type and automatically increments each added row. This primary key can be referenced by the `orders` table.
- `sales_channel_name VARCHAR(255) NOT NULL,`
  - The Sales Channel Name is a unique name for each sales channel. It is a varying character type that can receive up to 255 characters.
- `CONSTRAINT "channel_name_not_empty" CHECK (LENGTH(TRIM(sales_channel_name)) >= 0)`
  - This constraint ensures that a blank value or empty string can't be used as a valid Sales Channel Name.

Orders table

- `order_id SERIAL PRIMARY KEY,`
  - The Order ID is a primary key that uniquely identifies an order and automatically increments each added row. This primary key can be referenced by the `sales_details` table.
- `order_priority CHAR(1),`
  - The Order Priority is a one-letter code that identifies the priority level. It is a character type of 1 value.
- `order_date DATE,`
  - The Order Date is of the Date type, identifying when the sales order was placed.
- `ship_date DATE,`
  - The Ship Date is of the Date type, identifying when EcoMart initiated the outbound shipment of an order.
- `country_id INT,`
  - The Country ID is a foreign key integer that references the Country ID primary key in the `country` table.

- `item_type_id INT,`
    - The Item Type ID is a foreign key integer that references the Item Type ID primary key in the `item_type` table.
- `sales_channel_id INT,`
    - The Sales Channel ID is a foreign key integer that references the Sales Channel ID primary key in the `sales_channel` table.
- `FOREIGN KEY (country_id) REFERENCES country(country_id),`
    - This establishes the relationship of the Country ID key as a foreign key to the primary key in the `country` table.
- `FOREIGN KEY (item_type_id) REFERENCES item_type(item_type_id),`
    - This establishes the relationship of the Item Type ID key as a foreign key to the primary key in the `item_type` table.
- `FOREIGN KEY (sales_channel_id) REFERENCES sales_channel(sales_channel_id)`
    - This establishes the relationship of the Sales Channel ID key as a foreign key to the primary key in the `sales_channel` table.

Sales Details table

- `sales_detail_id SERIAL PRIMARY KEY,`
    - The Sales Detail ID is a primary key that uniquely identifies an order and automatically increments each added row.
- `order_id INT,`
    - The Order ID is a foreign key integer that references the Order ID primary key in the `orders` table.
- `units_sold INT,`
    - Units Sold is an integer indicating the number of units sold
- `unit_price DECIMAL(10, 2),`
    - The Unit Price is stored as a decimal with up to 10 digits total, 2 of which are behind the decimal point to represent currency.
- `unit_cost DECIMAL(10, 2),`
    - The Unit Cost is stored as a decimal with up to 10 digits total, 2 of which are behind the decimal point to represent currency.
- `total_revenue DECIMAL(10, 2),`
    - The Total Revenue is stored as a decimal with up to 10 digits total, 2 of which are behind the decimal point to represent currency.
- `total_cost DECIMAL(10, 2),`
    - The Total Cost is stored as a decimal with up to 10 digits total, 2 of which are behind the decimal point to represent currency.
- `total_profit DECIMAL(10, 2),`
    - The Total Profit is stored as a decimal with up to 10 digits total, 2 of which are behind the decimal point to represent currency.

- `FOREIGN KEY (order_id) REFERENCES orders(order_id)`
    - This establishes the relationship of the Order ID key as a foreign key to the primary key in the `orders` table.

D. Scalability

Scalability refers to the ability of a database to handle the growing amount of data (Binni & Neagoie 2024).

This can refer to adding more observations to the existing data (e.g., new orders are placed and tracked), having more operations that need to be performed on the data (e.g., sales data analysis, a data professional team performing simultaneous operations, regression analysis, etc.), and adding new types of data (e.g., adding sustainability certifications, new ranges of products and product attributes, product availability, etc).

The proposed data solution splits the data into related tables as a way of protecting the existing data while adding the capacity to collect and store more data, effectively analyze the data, and make it flexible enough to allow the table structures to be modified to include entirely new information as EcoMart's needs develop. Through SQL queries, the tables can be joined together to provide cohesive query results without the need for data redundancy or complex updates.

E. Privacy & Security

The single best way to ensure that data stays private and secure is to use the principle of least privilege. This refers to the best practice that "a person or system should be given only the privileges and data they need to complete the task at hand and nothing more" (Reis & Housley 2022). Users should not have access to another user's data. Only a select crew of data professionals should have access to any databases or tables that include passwords or personal identifiable information (PII) and those people need to be trusted to have discretion not to use the data in malicious ways. Select roles may need to have the ability to add, change, or remove data in a database depending on the organization, while others may only need to query the data. Therefore roles should be designed to grant only the privileges needed by the user and those privileges should be revoked when/if access is no longer relevant to their role.

It's also important to keep unauthorized users from being able to access databases, to take measures to prevent data corruption (such as keeping database servers in a cool and secure place), maintain audits to detect and stop any malware attacks, and patch any vulnerabilities (Binni & Neagoie 2024).

SQL commands can be used to create roles with specific access and permissions. Then users can be added or removed to those roles so that each role does not need to be created on an ad hoc basis; any changes made to the role will pass on to any user assigned to that role. If a user leaves the organization or changes position, their permissions can be revoked without impacting any other user.

F. Implementation

F1. Database Instance

The following code is used to create the database:

```sql
-- Create Database: "D597_Task_1"
-- (WGU Webinar 2, 2024)
CREATE DATABASE "D597_Task_1"
  WITH
  OWNER = postgres
  ENCODING = "UTF8"
  LC_COLLATE = 'English_United States.1252'
  LC_CTYPE = 'English_United States.1252'
  TABLESPACE = pg_default
  CONNECTION LIMIT = -1
  IS_TEMPLATE = False;
```

The following code is used to create a temporary table to store the data imported via the CSV as it is:

```sql
-- Create a temporary table for import
CREATE TABLE bad_sales_records (
    region VARCHAR(255),
    country VARCHAR(255),
    item_type VARCHAR(255),
    sales_channel VARCHAR(255),
    order_priority CHAR(1),
    order_date DATE,
    order_id INT,
    ship_date DATE,
    units_sold INT,
    unit_price DECIMAL(10, 2),
    unit_cost DECIMAL(10, 2),
    total_revenue DECIMAL(10, 2),
    total_cost DECIMAL(10, 2),
    total_profit DECIMAL(10, 2)
);
```

The following code is used to import the data from the CSV to the bad_sales_records table:

```
-- Import the CSV data to the temporary table
COPY bad_sales_records (
    region, country, item_type, sales_channel, order_priority,
    order_date, ship_date, units_sold, unit_price,
    unit_cost, total_revenue, total_cost, total_profit
)
FROM "C:\Users\student\Downloads\1000000 Sales Records.csv"
DELIMITED ','
CSV HEADER;
```

The above table will be dropped when the data has been split up and moved to the appropriate tables.

The following code is used to create the tables and necessary constraints:

```
-- Create `region` table
CREATE TABLE region (
    region_id SERIAL PRIMARY KEY,
    region_name VARCHAR(255) NOT NULL,
    -- Add constraint
    CONSTRAINT "region_name_not_empty" CHECK (LENGTH(TRIM(region_name)) >=
0)
);

-- Create `country` table
CREATE TABLE country (
    country_id SERIAL PRIMARY KEY,
    country_name VARCHAR(255) NOT NULL,
    region_id INT,
    -- Add constraint
    CONSTRAINT "country_name_not_empty" CHECK (LENGTH(TRIM(country_name))
>= 0),
    -- Add foreign key references
    FOREIGN KEY (region_id) REFERENCES region(region_id) ON DELETE CASCADE
);
```

```
-- Create `item_type` table
CREATE TABLE item_type (
    item_type_id SERIAL PRIMARY KEY,
    item_type_name VARCHAR(255) NOT NULL,
    -- Add constraint
    CONSTRAINT "item_type_name_not_empty" CHECK
(LENGTH(TRIM(item_type_name)) >= 0)
);

-- Create `sales_channel` table
CREATE TABLE sales_channel (
    sales_channel_id SERIAL PRIMARY KEY,
    sales_channel_name VARCHAR(255) NOT NULL,
  -- Add constraint
    CONSTRAINT "channel_name_not_empty" CHECK
(LENGTH(TRIM(sales_channel_name)) >= 0)
);

-- Create `orders` table
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY NOT NULL,
    order_priority CHAR(1),
    order_date DATE,
    ship_date DATE,
    country_id INT,
    item_type_id INT,
    sales_channel_id INT,
    -- Add foreign key references
    FOREIGN KEY (country_id) REFERENCES country(country_id) ON DELETE
CASCADE,
    FOREIGN KEY (item_type_id) REFERENCES item_type(item_type_id) ON DELETE
CASCADE,
    FOREIGN KEY (sales_channel_id) REFERENCES
sales_channel(sales_channel_id) ON DELETE CASCADE
);

-- Create `sales_details`
CREATE TABLE sales_detail (
    sales_detail_id SERIAL PRIMARY KEY,
    order_id INT,
    units_sold INT,
    unit_price DECIMAL(10, 2),
    unit_cost DECIMAL(10, 2),
    total_revenue DECIMAL(10, 2),
    total_cost DECIMAL(10, 2),
    total_profit DECIMAL(10, 2),
    -- Add foreign key references
    FOREIGN KEY (order_id) REFERENCES orders(order_id) ON DELETE CASCADE
);
```

F2. Insert Records

The following blocks of code were used to insert the data from the bad_sales_records tables into the appropriate tables:

```
-- Insert data into `region` table
INSERT INTO region (region_name)
    SELECT DISTINCT region
    FROM bad_sales_records
    WHERE region IS NOT NULL;

-- Insert data into `country` table
INSERT INTO country (country_name, region_id)
    SELECT DISTINCT bsr.country, r.region_id
    FROM bad_sales_records AS bsr
    JOIN region AS r ON bsr.region = r.region_name
    WHERE bsr.country IS NOT NULL;

-- Insert data into `item_type` table
INSERT INTO item_type (item_type_name)
    SELECT DISTINCT item_type
    FROM bad_sales_records
    WHERE item_type IS NOT NULL;

-- Insert data into `sales_channel` table
INSERT INTO sales_channel (sales_channel_name)
    SELECT DISTINCT sales_channel
    FROM bad_sales_records
    WHERE sales_channel IS NOT NULL;

-- Insert data into `orders` table
INSERT INTO orders (order_id, order_priority, order_date, ship_date,
country_id, item_type_id, sales_channel_id)
    SELECT
```

```
        bsr.order_id,
        bsr.order_priority,
        bsr.order_date,
        bsr.ship_date,
        c.country_id,
        it.item_type_id,
        sc.sales_channel_id
    FROM bad_sales_records AS bsr
    JOIN country AS c ON bsr.country = c.country_name
    JOIN item_type AS it ON bsr.item_type = it.item_type_name
    JOIN sales_channel AS sc ON bsr.sales_channel = sc.sales_channel_name;

-- Insert data into `sales_detail` table
INSERT INTO sales_detail (order_id, units_sold, unit_price, unit_cost,
total_revenue, total_cost, total_profit)
    SELECT
        o.order_id,
        bsr.units_sold,
        bsr.unit_price,
        bsr.unit_cost,
        bsr.total_revenue,
        bsr.total_cost,
        bsr.total_profit
    FROM bad_sales_records AS bsr
    JOIN orders AS o ON bsr.order_id = o.order_id;
```

Once the data was transferred to the new tables, queries were run to verify the data:

```
-- Verify data in `regions` table
SELECT *
FROM region;

-- Verify data in `country` table
SELECT *
FROM country
LIMIT 10;

-- Verify data in `item_type` table
SELECT *
FROM item_type
LIMIT 10;
```

```sql
-- Verify data in `sales_channel` table
SELECT *
FROM sales_channel
LIMIT 10;

-- Verify data in `orders` table
SELECT *
FROM orders
LIMIT 10;

-- Verify data in `sales_detail` table
SELECT *
FROM sales_detail
LIMIT 10;
```

After it was verified the data was stored in the appropriate tables, the temporary bad_sales_record table was dropped to avoid redundant data:

```sql
DROP TABLE IF EXISTS bad_sales_records;
```

F3. Queries

This section will discuss the queries run in increasing complexity, comparing the run times before and after optimization. The optimization tactics will be discussed in the following section.

This first query identifies the aggregate total sales per region:

```sql
-- Aggregate the total sales per region
SELECT
    r.region_name,
    SUM(sd.total_revenue) AS total_sales
FROM sales_detail AS sd
JOIN orders AS o ON sd.order_id = o.order_id
JOIN country AS c ON o.country_id = c.country_id
JOIN region AS r ON c.region_id = r.region_id
GROUP BY r.region_name
ORDER BY total_sales DESC;
```

Prior to optimization, the query returned 7 rows and ran in 00:00:00:303 seconds. After optimization, the query ran in 00:00:00:226 seconds, improving slightly.

The second query identifies the profit margin by item type:

```sql
-- Identify profit margin by item type
SELECT
    it.item_type_name,
    SUM(sd.total_profit) / SUM(sd.total_revenue) * 100 AS profit_margin
FROM item_type AS it
JOIN orders AS o ON it.item_type_id = o.item_type_id
JOIN sales_detail AS sd ON o.order_id = sd.order_id
GROUP BY it.item_type_name
ORDER BY profit_margin DESC;
```

Prior to optimization, the query returned 12 rows and ran in 00:00:00:211 seconds. After optimization, the query ran in 00:00:00:203 seconds, improving slightly.

The third query calculates the month-over-month sales by country, calculating the percent change from the previous month:

```sql
-- Identify month-over-month-sales by country and calculate the percent
change from the previous month
WITH monthly_sales AS (
    SELECT
        c.country_name,
        DATE_TRUNC('month', o.order_date) AS month,
        SUM(sd.total_revenue) AS total_sales
    FROM sales_detail AS sd
    JOIN orders AS o ON sd.order_id = o.order_id
    JOIN country AS c ON o.country_id = c.country_id
    GROUP BY
        c.country_name,
        DATE_TRUNC('month', o.order_date)
),
sales_with_lag AS (
    SELECT
        country_name,
        month,
        total_sales,
        LAG(total_sales) OVER (PARTITION BY country_name ORDER BY month) AS
previous_month_sales
    FROM monthly_sales
)
SELECT
```

```
    country_name,
    month,
    total_sales,
    previous_month_sales,
    CASE
        WHEN previous_month_sales IS NULL THEN NULL
        ELSE (total_sales - previous_month_sales) / previous_month_sales *
100
    END AS month_over_month_change
FROM sales_with_lag
ORDER BY
    country_name,
    month;
```

Prior to optimization, the query returned 16,785 rows and ran in 00:00:01:001 seconds. After optimization, the query ran in 00:00:00:830 seconds, which is significant to scale.

F4. Optimization

The following code provides the indexes added to optimize the query performance:

```
-- Index on region_name in the region table
CREATE INDEX idx_region_name ON region(region_name);

-- Index on country_name and region_id in the country table
CREATE INDEX idx_country_name ON country(country_name);
CREATE INDEX idx_country_region_id ON country(region_id);

-- Index on item_type_name in the item_type table
CREATE INDEX idx_item_type_name ON item_type(item_type_name);

-- Index on sales_channel_name in the sales_channel table
CREATE INDEX idx_sales_channel_name ON sales_channel(sales_channel_name);

-- Composite index on order_date and country_id in the orders table
CREATE INDEX idx_orders_order_date_country_id ON orders(order_date,
country_id);

-- Index on order_id in the sales_detail table
CREATE INDEX idx_sales_detail_order_id ON sales_detail(order_id);
```

```sql
-- Composite index on total_revenue and total_profit in the sales_detail
table
CREATE INDEX idx_sales_detail_revenue_profit ON sales_detail(total_revenue,
total_profit);
```

Repeating the queries in section F3 will show an improvement in the run time once the indexes are added.

Something to note is that this is still considered a small dataset with only 100,000 records in this database, meaning we won't see substantial performance enhancements quite yet. According to the Udacity SQL Nanodegree, performance optimizations are likely to be noticeable when you are working with a dataset of a million or more records (Saab 2025). These optimizations are still good to do now, as they will prepare EcoMart's system for exponential growth in their data collection, enhancing overall scalability and flexibility.

G. Panopto Walkthrough

https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=3f9a65e3-554c-4d41-b15c-b262015ad47d

H. References

Binni, Mo & Neagoie, Andrei (2024). *Complete SQL + Databases Bootcamp*, Zero to Mastery. https://zerotomastery.io/courses/sql-bootcamp/.

Reis, J., & Housley, M. (2022). Fundamentals of Data Engineering: Plan and Build Robust Data Systems (1st ed.). O'Reilly Media.

Saab, Ziad (2025). *Management of Relational and Non-relational Databases*, Udacity. https://www.udacity.com/course/management-of-relational-and-non-relational-databases--cd0362.

Sewell, William (2024). *Webinar 2*, Western Governors University.

Shan, J., Goldwasser, M., & Malik, U. (2022). SQL for Data Analytics: Harness the power of SQL to extract insights from data (3rd ed.). Packt Publishing.

Steer, Derek, & Sridhar, Malavica (2024). *Introduction to SQL*, Udacity. https://www.udacity.com/course/introduction-to-sql--cd0021.