Normalizing an E-Commerce Relational Database for Seasonality, Profitability, and Regional Analysis

Joanne Senoren

Master of Science, Data Analytics

Table of Contents

Part 1: Design Document

A1. Business Problem

Marketing strategy efforts can bolster EcoMart's commitment to promoting sustainable and ethically sourced products. A database solution that helps to make sense of the provided raw data can efficiently offer insights into which products to promote and where to focus marketing strategies for a business such as EcoMart. The business problem we will pursue in this paper is: How can EcoMart leverage its current data to better understand the company's sales trends and know which products to promote based on seasonality, profits, and region? By creating a database and organizing data in a structured format, we can transform and manipulate data to explore helpful information to help direct and guide EcoMart's marketing approach.

Since EcoMart's priority is to connect conscious consumers with their partner brands, they can use data-driven marketing to understand which products resonate with their customers. Seasonality and location are advantageous features to consider when customizing targeting campaigns. EcoMart can fine-tune its product offerings by understanding which products sell most in specific markets and when these sales peak. These adjustments can improve customer engagement and conversion (Measured: Marketing Attribution & Incrementality Testing, 2024).

A2. Data Structure

A relational database can organize the raw data and manipulate it with SQL for advanced analytics. A relational model that arranges data in rows and columns within respective tables allows someone such as a data analyst to efficiently perform logical operations in the form of queries that can help with our business problem (Malik et al., 2019). We can access these related tables of data from a database such as a PostgreSQL database that manages the tables with

identifiers like primary and foreign keys. A relational database is a good choice for raw data that tracks transactions and is already organized in a defined format, such as a CSV file. Tables 1 and 2 below demonstrate some initial functional and non-functional requirements to guide our relational database approach.

Table 1

Functional Requirements

Aspect	Functional Requirement
Focus	The database user should be able to store, manage, and access order data such as IDs, products, and locations with capabilities across adding, updating, and removing data and tables.
Nature	The database should allow users to query data and generate reports with data accurately.
Data Relationships	The database supports one-to-many relationships, and users can identify and join tables based on their data analysis needs.
Data Consistency	The system should be able to perform data validation based on established constraints and data type rules.

Table 2

Non-Functional Requirements

Aspect	Non-Functional Requirements
Security	Only users with the appropriate security clearance can make permanent changes within the database.
Maintainability	The database should have established relationship rules to maintain data consistency and accuracy across relations within the database.
Performance	The database system should be able to retrieve data queries and respond quickly.

Reliability	The system should be able to check data integrity rules and send messages to users if they violate database rules.

The target categories within the data (location, order date, product) are identifiable within the functional requirements above. To access the data efficiently, we can separate the database into tables with relevant values based on our business problem. Location, timing, and product information can all be separated into their own space, or tables, within the database. The conceptual data model in Figure 1 shows how we can establish a structured database to organize and analyze EcoMart's data more efficiently. It consists of the target data we want to explore and the related tables we can expect to exist within the database. To see the final entity relationship diagram and logical data model, see Figure 5.

Figure 1

Conceptual Data Model

A3. Justification for Relational Database

We cannot make sound marketing decisions with our current raw data in CSV format. Raw data are simple records, and only when we arrange them to reveal patterns can we make informed decisions (Malik et al., 2019). Establishing a relational database using EcoMart's raw data will be a beneficial choice for the company. Using a query language such as SQL with the

data tables, we can efficiently perform some exploratory data analysis by assessing patterns using queries to manipulate the data within the database.

A relational database is also a flexible solution because a user can create, update, and change specific tables without affecting other parts of the database. Relational databases also support ACID (Atomicity, Consistency, Isolation, and Durability). The ACID properties are essential in maintaining data integrity, especially regarding transactional data such as EcoMart's orders. With sound data integrity supported by a relational database, we can ensure that our marketing data analysis will be accurate (Coronel & Morris, 2022).

A4. How Business Data Will Be Used

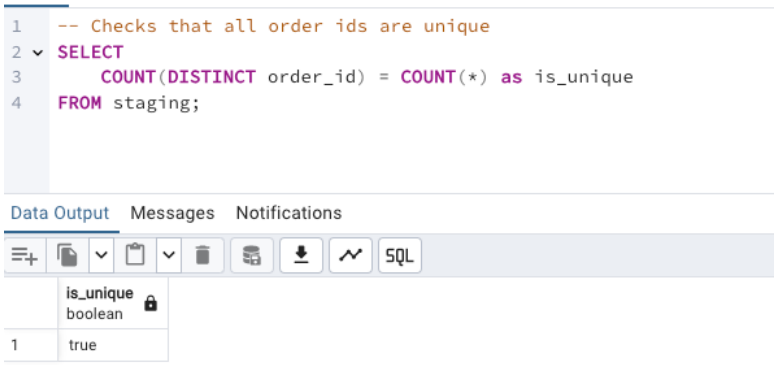
We can perform marketing analysis from EcoMart's transaction data once imported and organized across related tables within a database. Product categories and columns such as profit and revenue will be able to tell us which categories sell the most items. The region, ship date, and order date columns will allow us to customize and segment data based on time and location. Since the data is in a CSV format, it must be normalized and distributed across related tables. Users can join these tables into a more digestible format for efficient analysis. Normalizing data within a database is important for eliminating redundancies and minimizing data errors found in flat file systems (Coronel & Morris, 2022). Utilizing SQL queries to generate aggregated data by region, product vertical, and profits will help answer questions related to the target marketing business problem. Questions such as which products generate the most profit and which regions buy the most can guide marketing strategy decisions.

B. Logical Model

Normalization techniques were employed to achieve the final logical model. Normalization is an important step to remove redundancy and create efficiency in the database (Coronel & Morris, 2022). The data's first normal form and its attributes are demonstrated in Figure 3, where all cells within the relation have one and only one value that meets the 1NF requirement. The first form consists of only one table at this point. A review of the data values identified the primary key, a non-null and unique attribute in the relation. The SQL query shown in Figure 2 checked for uniqueness and non-null cells within the 'order_id' attribute. Since 'order_id' has met non-null and uniqueness constraints, it is a good candidate for a primary key.

Figure 2

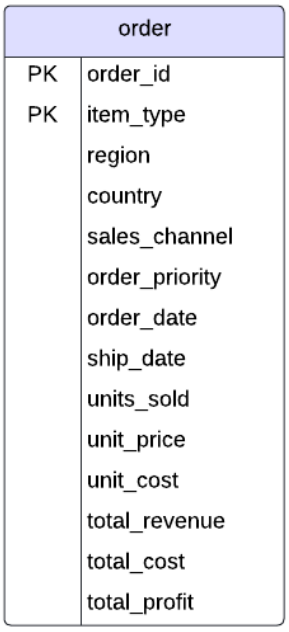
Query to Check for 'order_id' Uniqueness and Nulls



Although the query above shows' order_id' to be unique, the 'item_type' attribute is inherently just as important to record uniqueness. There is no evidence of order IDs having multiple item types to date, but there is no indication that it will not happen in the future. The database should be flexible for that possibility, so it must be able to map multiple items to order IDs. The solution is to make a primary key for the *order* table a composite key of 'order_id' and 'item_type.'

Figure 3

Attributes in 1NF



The second normal form states that all non-prime attributes depend on the key and only the key. Achieving 2NF involves separating the primary (composite) keys and their dependent attributes into their own tables. The resulting tables eliminate partial dependency on the composite key in the *order* table (Coronel & Morris, 2022). The attributes' order_date,' 'ship_date,' 'sales_channel,' 'order_priority,' 'country,' and 'region' partially depend on 'order_id .' The generated *order_detail* table consists of these attributes along with 'order_id' as a primary key to the table.

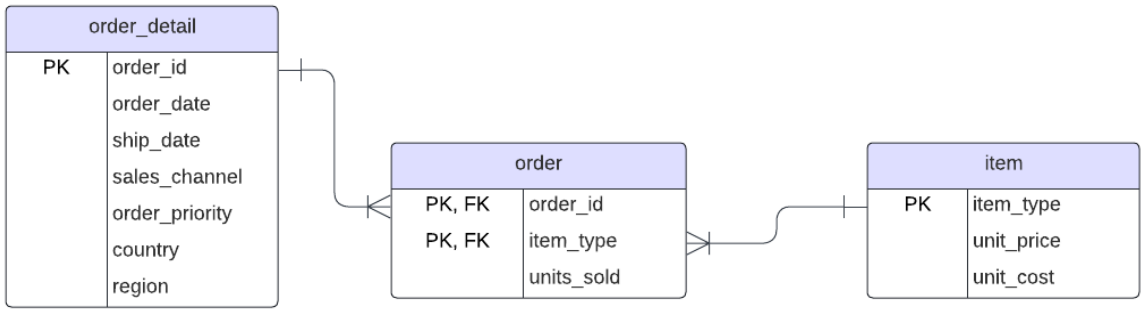
The second partially dependent group of attributes are 'item_type,' 'unit_cost,' and 'unit_price .' The relation has twelve unique item types, each with a unique and singular unit cost and price. Establishing a table for item types will eliminate insertion anomalies when additional

items to EcoMart's orders appear in new orders. This new table is aptly named *item* (Decomplexify, 2021).

The attribute 'units_sold' is the only attribute dependent on the composite key, so it remains in the original 'order' table. The attribute totals ('total_cost,' 'total_profit,' 'total_revenue') were removed since they are derived attributes that can be calculated with a query language. Removing these totals helps to scale down the database and ensure accuracy on the latest data by calculating instantaneously, which captures new changes as orders are updated (Coronel & Morris, 2022). The two new tables are linked to the *order* table using foreign keys, as shown in Figure 4.

Figure 4

Attributes and Entities in 2NF

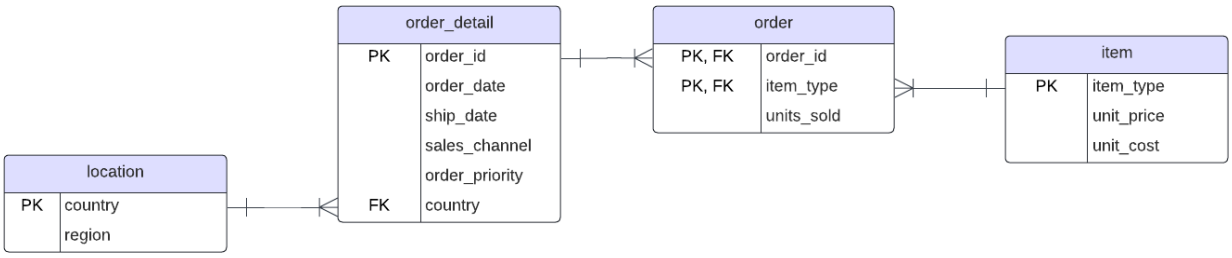


The 3NF rule states that a database must be in 2NF and not consist of transitively dependent attributes. Transitively dependent attributes are attributes that are dependent on the primary key but also dependent on a non-prime attribute. The attribute' region' is dependent on 'order_id' but can also be determined by 'country.' Therefore, 'country' and 'region' are moved to a fourth table named *location*. The primary key for this new table is 'country,' which acts as a

foreign key in the *order_detail* table. At this point, the tables in Figure 5 are in their third normal form and normalized.

Figure 5

Attributes and Entities in 3NF



The normalized tables demonstrate consideration towards insert, update, and deletion anomalies. For example, there are 195 countries in the world today. Based on a unique count of countries in the 'country' attribute, 185 are present in the database. EcoMart has yet to fulfill orders to all countries in the world. Regarding scalability, future orders may involve adding those ten countries. The *location* table gives a few viable options for addressing these missing countries. The diagram above allows users to insert or delete a country in the location without updating the related tables and violating non-null or other constraints. If updates to specific regions need to be made, users would only have to update one country entry in the *location* table for all relevant orders to have that updated information. The normalized entity relationship diagram in Figure 5 shows how such actions within a database minimize data redundancy and maintain consistency.

C. Database Objects and Storage

The proposed relational database in Figure 5 consists of five tables and eleven total attributes. The database schema is a snowflake schema with a fact table connected to dimension

tables that can also be fact tables. The primary fact table for this database is the *order* table that directly connects to the *order_detail* table and *item_type* table. The *order* table contains three attributes – 'order_id,' 'item_type,' and 'unit_cost.' The *item_type* table is related to the *order* table via the 'item_type' attribute, which acts as a primary key and foreign key in the fact table and as a primary key in the *item_type* table. The two tables exhibit a one-to-many relationship, where multiple instances of 'item_type' can exist in the *order* table, but only one instance of 'item_type' exists in the *item* table.

Order and *order_detail* are related through the 'order_id' key. The *order* table may consist of multiple line items within the same order ID for future scalability purposes. Therefore, 'order_id' and 'item_type' are composite primary keys in the *order* table. The 'order_id' attribute acts as a primary key for *order_detail* while it acts as a composite primary and foreign key in the *order* table. The current data exhibits that of a one-to-one relationship between *order* and *order_detail*. The relationship shown in Figure 5 shows the flexible version of the database where we assume that multiple items could exist in one order. Thus, we will implement the database with a one-to-many relationship from *order* to *order_detail*.

The *order_detail* table is also a fact table in relation to the *location* table. The attribute 'country' acts as a primary key in the *location* table and as a foreign key in the *order_detail* table. We established that regions are functionally dependent on country, so the attributes' region' and 'country' are in the *location* table together. *Order_detail* and *country* have a one-to-many relationship where there are multiple instances of one country across numerous orders, but an order is shipped to only one country.

Table 3 below demonstrates the data types of attributes per table, constraints, examples, descriptions, and estimated storage bytes. Figure 6 shows the physical data model with top-level constraints (keys), data types, attributes, and relationships.

Table 3

Database Tables, Attributes, and Descriptions

order					
Attribute	***Key Type (as necessary)***	***Expected Constraints***	***Description***	***Example***	***Data Type***
order_id	PK (order) | FK (order_detail)	not null, FK, PK	The ids for completed and shipped orders.	284384381	integer
item_type	PK (order) | FK (item_type)	not null, FK, PK	The category name for type of ordered items.	Snacks	varchar(255)
units_sold		not null	The number of item units sold per order.	4513	integer

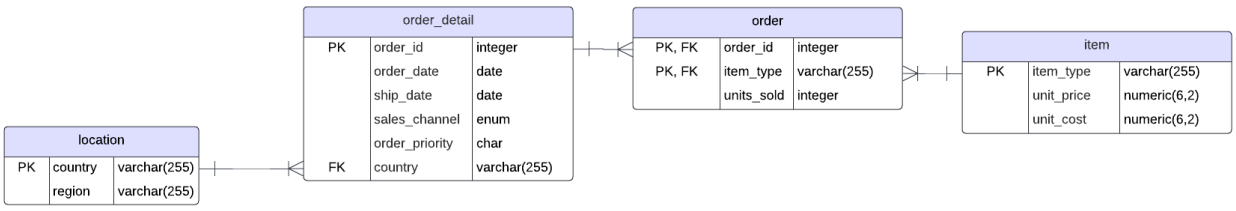
order_detail					
Attribute	***Key Type (as necessary)***	***Expected Constraints***	***Description***	***Example***	***Data Type***
order_id	PK	not null, unique, PK	The ids for completed orders.	284384381	integer
order_date		less than ship date	The date that an order was completed.	2012-02-02	date
ship_date		more than order date	The date that an order was shipped.	2012-02-10	date
sales_channel			The type of channel where a sale was made.	online	enum
order_priority			The order priority code per order.	C	char
country	FK (location)	not null, FK	The country where the orders are completed.	Italy	varchar(255)

item					
Attribute	***Key Type (as necessary)***	***Expected Constraints***	***Description***	***Example***	***Data Type***
item_type	PK	not null, unique, PK	The category name for type of ordered items.	Snacks	varchar(255)
unit_price		not null	The price per item of each other.	152.58	numeric (6, 2)
unit_cost		not null	The cost per item of each other.	97.44	numeric (6, 2)

location					
Attribute	***Key Type (as necessary)***	***Expected Constraints***	***Description***	***Example***	***Data Type***
country	PK	not null, unique	The country where the orders are completed.	Italy	varchar(255)
region			The region location for each country.	Europe	varchar(255)

Figure 6

Final Data Model



The data types for all attributes were established using the PostgreSQL dictionary. Since the values are whole numbers, the attributes' order_id' and 'units_sold' have integer data types. EcoMart's data tells us that their product units are whole in whole numbers, so using the integer data type makes sense. The integers' item_type,' 'country,' and 'region' have the varchar(255) data type since the values are limited characters texts but with varying lengths. The attributes' order_date' and 'ship_date' have the date data type since the values in the sales staging represent a day. 'The numeric data type was used for 'unit_price' and 'unit_cost' and accounts for future additions of prices with larger values. Currently, the maximum value for both attribute exhibits consists of five total digits with two digits to the right of the decimal point. The current maximum value presents a numeric(5, 2) data type. The physical data model consists of a numeric(6,2) data type for unit price and cost adjusted for scalability purposes. A user can alter this data type as necessary.

The values in 'order_priority' are codes demonstrated with single capital letters. Since all values are fixed alphabet codes, we can assume that additional inputs will be the same. This attribute has a char fixed character data type of one character. 'Sales_channel' only has two values that repeat – online and offline. The enumerated data type for this attribute is appropriate since it will maintain data consistency and integrity by limiting the inputs to the enumerated list (Hardy-Françon, 2024).

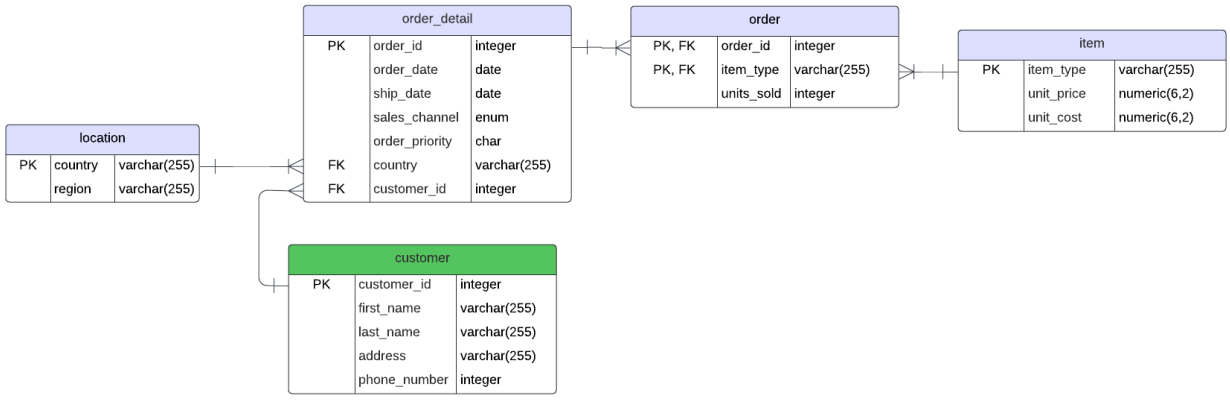
D. Database Design and Scalability

The database schema, as it stands, exhibits some strategic scaling considerations when it comes to data changes and queries. As mentioned in section B, the *order* table is connected to the *item_type* and *order_detail* table to account for potential additions of orders that consist of numerous items. Thus, we use the *order* as a fact table with a composite key using the 'item_type'

and 'order_detail' attributes. The schema is set up as a snowflake schema so that additional dimension tables can be added. For example, suppose we add a table with the customer's name and address. In that case, we can easily link the *customer* table to the *order_detail* table using a 'customer_id' attribute as a foreign key in the *order_detail* table and as a primary key in the *customer* table. The diagram in Figure 7 demonstrates this example.

Figure 7

Example: Adding A Table to Current Data Schema



The current database schema matches the data consisting of 100,000 rows. This number is relatively small compared to established sales data or transaction data. Our current data is far from being described as big data. The definition of big data is utilizing multiple complex datasets that grow exponentially over time, consisting of structured, unstructured, and semi-structured data that cannot be processed by traditional data processes (Abhijit, 2024). There are some considerations towards big data to discuss as we can expect EcoMart's business and data to increase. Some primary considerations for big data include data warehousing, scaling vertically, and using an OLAP service layer.

Data warehouses and online analytical processing (OLAP) can go hand-in-hand with business intelligence dealing with extensive, complex data. The data warehouse is a high-volume database that stores both historical and other data from external sources (Coronel & Morris, 2022). A data exploration that investigates long-term product trends to expand our business question would use a business intelligence solution such as a data warehouse and OLAP service layer. Additionally, as more data comes in and business grows, we must accommodate the increased data write load. A solution to consider in the future is scaling vertically, which involves adding more CPU, memory, and storage to the existing database. The added storage is a suitable scaling solution for data that grows in volume and continues to be predictable (Goyal, 2024).

The database schema is set up for online transactional processing (OLTP) since the data primarily records sales and orders (*OLTP Vs OLAP - Difference Between Data Processing Systems - AWS*, n.d.). Although our business scenario could develop to utilize more complex OLAP (online analytical processing) services, the question is simple enough to answer in an OLTP environment such as PgAdmin4 with SQL queries. Additionally, OLAP is best suited for large (in terabytes) data analysis, but our current data is still small. As EcoMart's data grows and marketing efforts become more multi-faceted, a strong consideration for an OLAP layer is recommended for future, more involved analysis.

E. Privacy and Security Measures

Babitz defines data security as a focus "on protecting data from unauthorized access, theft, or damage through technical measures like firewalls, encryption, and intrusion detection." It is imperative that any database system is protected, yet usable and secure, yet efficient (Babitz 2024). Based on these security needs, home high-level touchpoints surrounding data security include applying access controls, monitoring user roles, and data encryption and masking.

We can minimize malicious or accidental usage by adapting the "principle of least privilege" (Babitz, 2024). Implementing this principle involves authorizing limited access to necessary data based on user roles with specified privileges. Multi-factor authentication (MFA) enforces additional security when checking user access. Monitoring user access through automated procedures, such as identity and access management (IAM) services, allow database owners to verify user login and database actions through generated activity logs.

Data encryption and masking are important for sensitive data such as personally identifiable information (PII). Our current data does not contain sensitive customer names and contact information. However, encryption and masking are important considerations when EcoMart adds customer data information. Sensitive data must be encrypted as it transits a web client to a database and vice versa. If malicious actors intercept the data mid-transit, encrypted data will be unreadable (Babitz, 2024). Additionally, masking values within the database preserves the security of users and sensitive data. Masking could prevent data breaches from acquiring personally identifiable information (Pant, 2024).

Part 2: Implementation

F1. Database Creation

There are several ways to create a table, including using the graphical user interface within pgAdmin. The script in Figure 8 generated the database name "D597 Task 1". Mapping the data to the physical model took two steps, including importing the raw data to a staging table and then accurately transforming the data to map to the model tables. A staging table is a temporary holding table that mimics the structure of the raw file. Linhartová (2024) states that staging tables "provide a controlled environment for purging, transforming and loading data and

ensure that the final database remains consistent and performs well." Figure 9 shows the code for the staging table and importing CSV file data.

Figure 8

Create Database

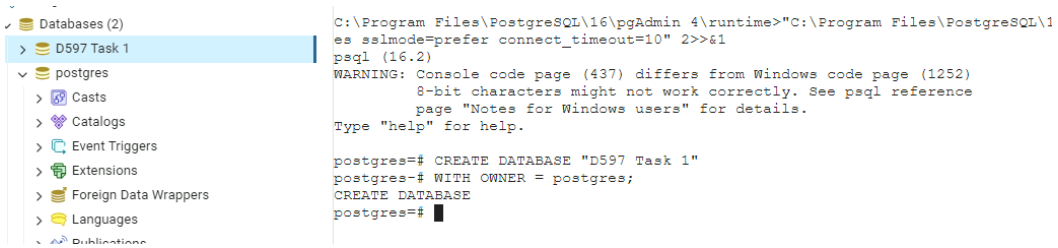


Figure 9

Create Staging Table & and CSV Data Import

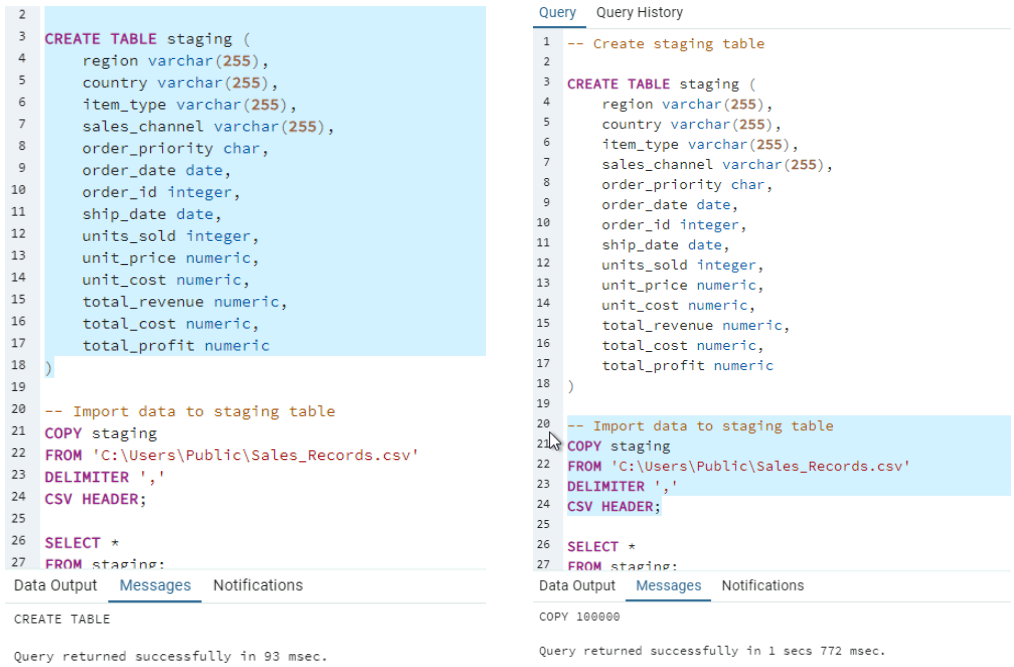
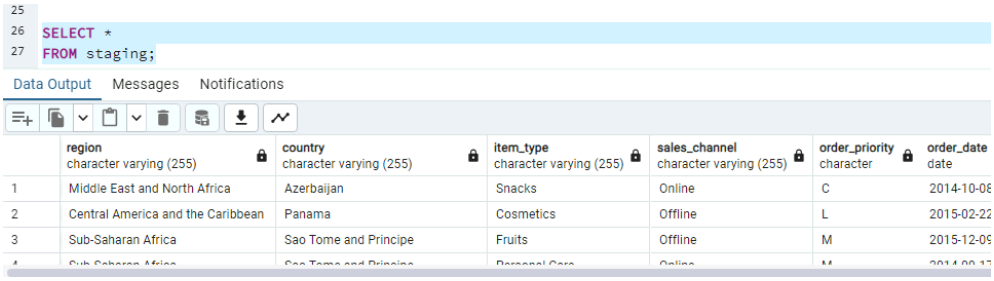


Figure 10 displays the query output confirming that the data was loaded with a SELECT statement.

Figure 10

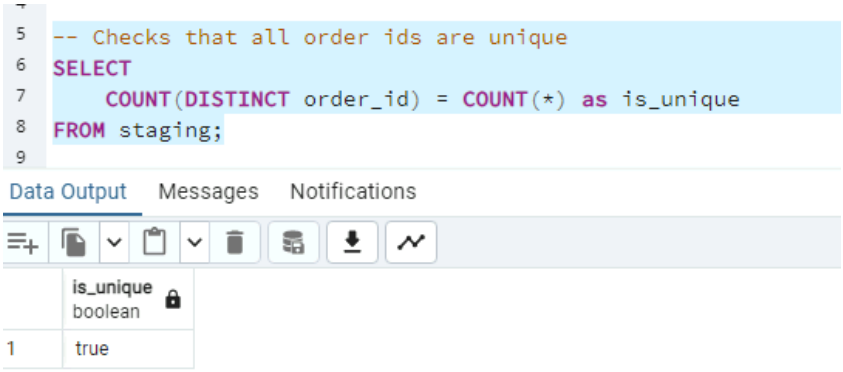
SELECT Statement to Confirm Import



A series of SQL queries checked nulls, duplications, and uniqueness. For instance, Figure 11 shows the code to confirm that all values in 'order_id' are unique. Additional queries confirmed data validation, such as checking that all ship dates occurred after their respective order dates. All queries that check raw data in the staging table are in the "Raw_Staging_Data_Check_PA1" file.

Figure 11

Confirmation That All Order IDs are Unique

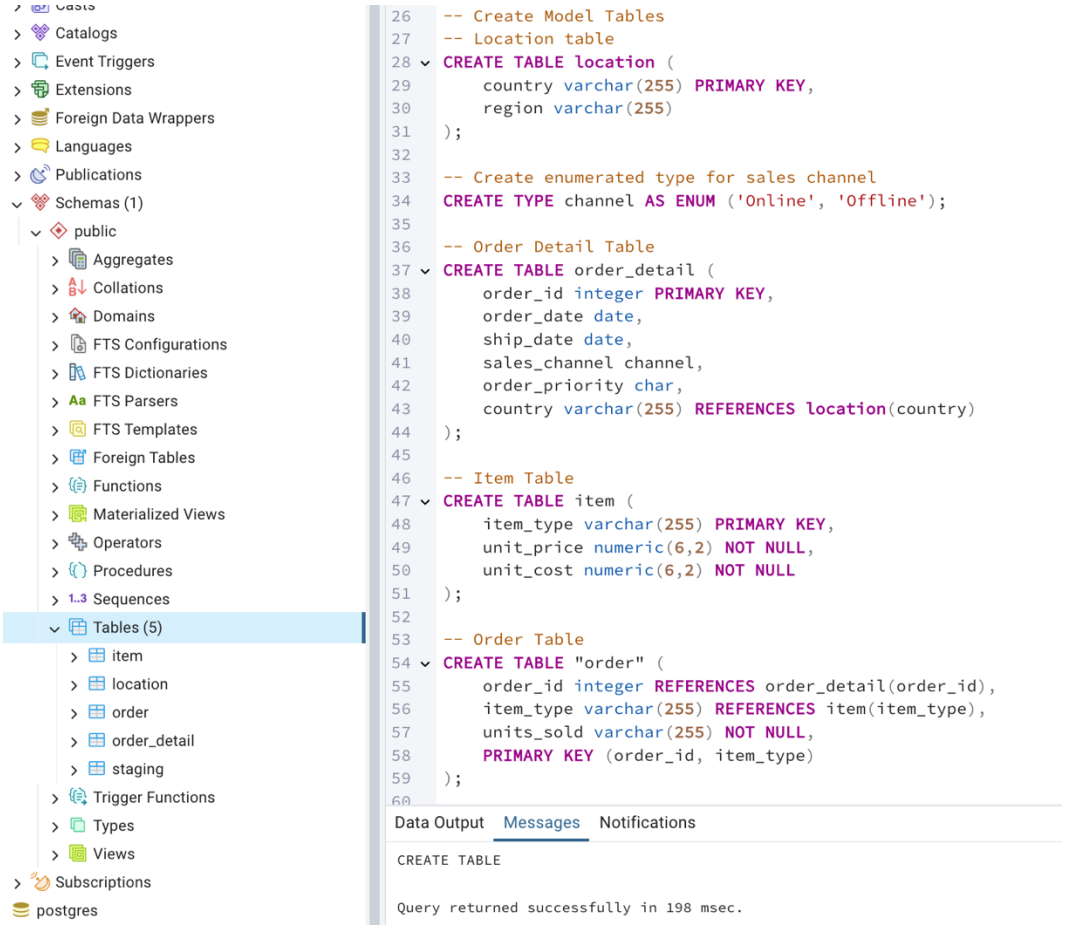


F2. Table Imports

The next step involves creating the model tables and mapping the appropriate data to each table based on the entity relationship diagram. Consequently, this step also includes establishing data and relationship constraints. Figure 12 demonstrates the code used to create all the model tables.

Figure 12

Create Model Tables



Figures 13 to 16 show the scripts that insert data into the model tables from the staging table. Figures 13 and 15 demonstrate how a SELECT statement imports unique rows to their respective tables.

Figure 13

Data Import for Location Table

```
64  -- Insert Data From Staging to Tables
65  -- location table
66  INSERT INTO location (country, region)
67  SELECT DISTINCT country, region
68  FROM staging;
69
70  -- Check inserted values
71  SELECT *
72  FROM "location";
73
```

Data Output | Messages | Notifications

	country [PK] character varying (255)	region character varying (255)
1	Greenland	North America
2	Nauru	Australia and Oceania
3	South Africa	Sub-Saharan Africa
4	Italy	Europe
5	Seychelles	Sub-Saharan Africa
6	China	Asia
7	Tajikistan	Asia
8	Pakistan	Middle East and North Africa
9	New Zealand	Australia and Oceania
10	Serbia	Europe
11	Mongolia	Asia
12	Russia	Europe
13	Montenegro	Europe

Total rows: 185 of 185 Query complete 00:00:00.349 Ln 65, Col 1

Figure 14 shows the imported values for order details. The staging table 'sales_channel' attribute consists of a varchar type. It was altered to use the enumerated type 'channel' to match the attribute 'sales_channel' in the *order_detail* table to insert the data.

Figure 14

Data Import for Order Detail Table

```sql
-- order_detail
-- Alter table varchar to enumerated sales_channel in staging
ALTER TABLE staging
    ALTER COLUMN sales_channel TYPE channel USING sales_channel::channel;

INSERT INTO order_detail (
    order_id,
    order_date,
    ship_date,
    sales_channel,
    order_priority,
    country )
SELECT
    order_id,
    order_date,
    ship_date,
    sales_channel,
    order_priority,
    country
FROM staging;

-- Check inserted values
SELECT *
FROM order_detail;
```

Data Output | Messages | Notifications

	order_id [PK] integer	order_date date	ship_date date	sales_channel channel	order_priority character (1)	country character varying (255)
1	535113847	2014-10-08	2014-10-23	Online	C	Azerbaijan
2	874708545	2015-02-22	2015-02-27	Offline	L	Panama
3	854349935	2015-12-09	2016-01-18	Offline	M	Sao Tome and Principe
4	892836844	2014-09-17	2014-10-12	Online	M	Sao Tome and Principe
5	129280602	2010-02-04	2010-03-05	Offline	H	Belize
6	473105037	2013-02-20	2013-02-28	Online	C	Denmark
7	754046475	2013-03-31	2013-05-03	Offline	M	Germany
8	772153747	2012-03-26	2012-04-07	Online	C	Turkey
9	847788178	2012-12-29	2013-01-15	Online	H	United Kingdom

Total rows: 1000 of 100000 Query complete 00:00:00.628 Ln 74, Col 1

Figure 15

Data Import for Item Detail Table

```
-- Insert to item Table
INSERT INTO item (
    item_type,
    unit_price,
    unit_cost)
SELECT DISTINCT
    item_type,
    unit_price,
    unit_cost
FROM staging;

-- Check inserted values
SELECT *
FROM item;
```

Data Output | Messages | Notifications

	item_type [PK] character varying (255)	unit_price numeric (6,2)	unit_cost numeric (6,2)
1	Office Supplies	651.21	524.96
2	Baby Food	255.28	159.42
3	Clothes	109.28	35.84
4	Household	668.27	502.54
5	Meat	421.89	364.69
6	Cereal	205.70	117.11
7	Snacks	152.58	97.44
8	Personal Care	81.73	56.67
9	Vegetables	154.06	90.93
10	Cosmetics	437.20	263.33
11	Beverages	47.45	31.79
12	Fruits	9.33	6.92

Total rows: 12 of 12 Query complete 00:00:00.198 Ln 99, Col 1

Figure 16

Data Import for Order Table

```
-- Insert to order Table
INSERT INTO "order" (
    order_id,
    item_type,
    units_sold)
SELECT
    order_id,
    item_type,
    units_sold
FROM staging;

-- Check inserted values
SELECT *
FROM "order";
```

Data Output | Messages | Notifications

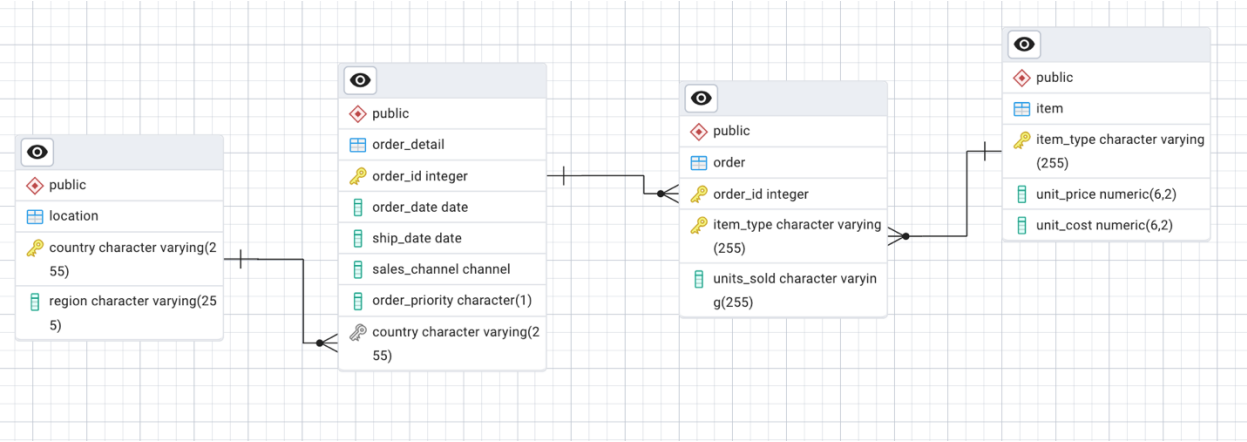
	order_id [PK] integer	item_type [PK] character varying (255)	units_sold character varying (255)
1	535113847	Snacks	934
2	874708545	Cosmetics	4551
3	854349935	Fruits	9986
4	892836844	Personal Care	9118
5	129280602	Household	5858
6	473105037	Clothes	1149
7	754046475	Cosmetics	7964
8	772153747	Fruits	6307
9	847788178	Snacks	8217
10	471623599	Cosmetics	2758
11	554646337	Cosmetics	1031

Total rows: 1000 of 100000 Query complete 00:00:00.472 Ln 114, Col 1

We can check the database ERD in pgAdmin to compare to the data model in Figure 6. Figure 17 confirms that the two ERDs match.

Figure 17

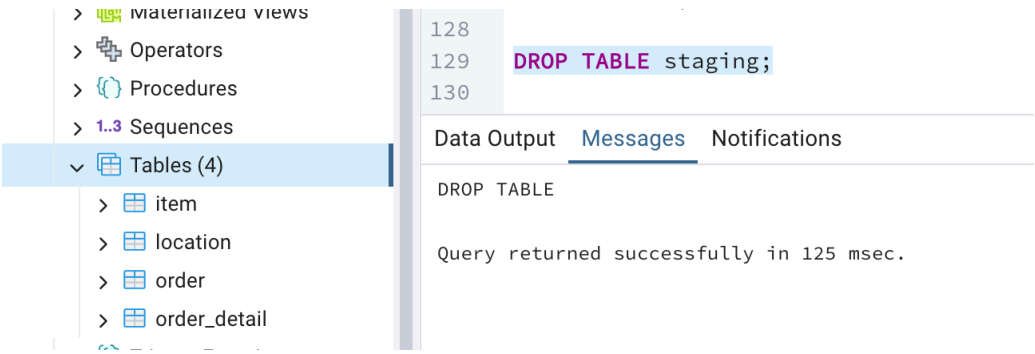
PgAdmin ERD



The staging table can be dropped from the database since it is a temporary table that loads data into their respective tables. Deleting the staging table also eliminates data redundancy. Figure 18 shows the script to remove the staging table from the database. Refer to the file titled "Create_Tables_PA1" to review the scripts presented in section F.

Figure 18

Drop Staging Table



F3. Business Problem Queries

Let us revisit the business problem: How can EcoMart leverage its current data to better understand the company's sales trends and know which products to promote based on seasonality, profits, and region? This question can elicit many related inquiries that SQL queries can help answer. We can explore the three questions and subsequent answers below:

1. What are the top three grossing item types sold based on revenue?
2. Which items sold the most (by units) per quarter?
3. What is the total revenue per region?

The inquiries focus specifically on each group within the initial research question. Question 1 provides insight into profits by items sold. Question 2 sheds information on item sales by time, specifically in increments of three months through the year. Question 3 provides details on the highest-grossing orders based on region.

Figures 19 to 21 show the executed scripts and generated results related to these business questions.

Figure 19

Top Three Grossing Item Types

```sql
-- What are the top three grossing item types sold based on revenue?
SELECT
    o.item_type,
    SUM(o.units_sold) AS total_units_sold,
    ROUND(AVG(i.unit_price),2) AS unit_price,
    -- All items priced by item type category so this unit price is the same per item
    ROUND(SUM(o.units_sold * i.unit_price),2) AS total_revenue
FROM "order" AS o
LEFT JOIN item AS i
ON o.item_type = i.item_type
GROUP BY o.item_type
ORDER BY total_revenue DESC
-- Show the top three highest first
LIMIT 3;
```

Data Output | Messages | Notifications

SQL

	item_type character varying (255)	total_units_sold bigint	unit_price numeric	total_revenue numeric
1	Household	41458795	668.27	27705668934.65
2	Office Supplies	42293330	651.21	27541839429.30
3	Cosmetics	41924464	437.20	18329375660.80

Figure 20

Most Items Sold by Units Per Calendar Quarter

```sql
-- Which items sold the most (by units) per quarter?
WITH item_units_sold_by_quarter AS (
SELECT
    item_type,
    -- EXTRACT function generates calendar quarter based on datetime (Q1, Q2, Q3, Q4)
    EXTRACT(quarter FROM order_date) AS calendar_quarter,
    -- Adds units of items sold by quarter
    SUM(o.units_sold) AS total_units_sold
FROM "order" AS o
LEFT JOIN order_detail AS od
ON o.order_id = od.order_id
GROUP BY item_type, calendar_quarter
ORDER BY calendar_quarter, total_units_sold DESC
),
ranked_units_sold AS (
SELECT
    *,
    -- Ranks total units sold by highest to lowest (1 being highest), grouped by quarter
    DENSE_RANK() OVER (PARTITION BY calendar_quarter ORDER BY total_units_sold DESC) as total_units_sold_rank_by_quarter
FROM item_units_sold_by_quarter
)
-- Generate max total units sold by quarter
SELECT
    item_type,
    calendar_quarter,
    total_units_sold
FROM
    ranked_units_sold
WHERE
    total_units_sold_rank_by_quarter = 1;
```

Data Output | Messages | Notifications

	item_type character varying (255)	calendar_quarter numeric	total_units_sold bigint
1	Cereal	1	11300905
2	Office Supplies	2	11308382
3	Meat	3	10460464
4	Baby Food	4	10338290

Figure 21

Total Items Sold and Total Revenue by Revenue

```sql
-- What is the total revenue per region?
WITH country_region_by_order_id AS (
    SELECT
        l.region,
        od.order_id
    FROM order_detail AS od
    LEFT JOIN "location" AS l
    ON od.country = l.country
),
price_units_sold_by_region AS (
    SELECT
    i.unit_price,
    o.units_sold,
    cr_id.region
    FROM
        "order" AS o
    LEFT JOIN
        item AS i
    ON
        o.item_type = i.item_type
    LEFT JOIN
        country_region_by_order_id as cr_id
    ON
        o.order_id = cr_id.order_id
)
SELECT
    SUM(units_sold) AS total_units_sold,
    -- AVG unit price is the average price of all units regardless of item type
    ROUND(AVG(unit_price),2) AS avg_unit_price,
    ROUND(SUM(units_sold * unit_price),2) AS total_revenue,
    region
FROM price_units_sold_by_region
GROUP BY
    region
ORDER BY
    total_revenue DESC;
```

Data Output | Messages | Notifications

	total_units_sold bigint	avg_unit_price numeric	total_revenue numeric	region character varying (255)
1	129890681	267.65	34958453406.17	Sub-Saharan Africa
2	128664731	266.85	34241150923.39	Europe
3	72822259	265.79	19293401219.82	Asia
4	63251625	266.14	16921412794.52	Middle East and North Africa
5	53871798	268.14	14553730165.29	Central America and the Caribbean
6	40797618	262.77	10701522223.73	Australia and Oceania
7	10845905	270.68	2937002333.49	North America

Total rows: 7 of 7 Query complete 00:00:00.661 Ln 86, Col 1

To review the scripts in detail, refer to the file titled "Business_Queries."

F4. Optimization Techniques

Database and query optimization is important to the scope of any analysis project. Long-running queries that take up resources can push back ad-hoc analysis timing and project timelines (Malik et al., 2019). Although the data in this paper is limited to 100,000 rows, some performant adjustments can determine whether efficiencies can optimize the queries. Any future

analytical queries within this dataset can benefit from optimizations established early in the database development stage.

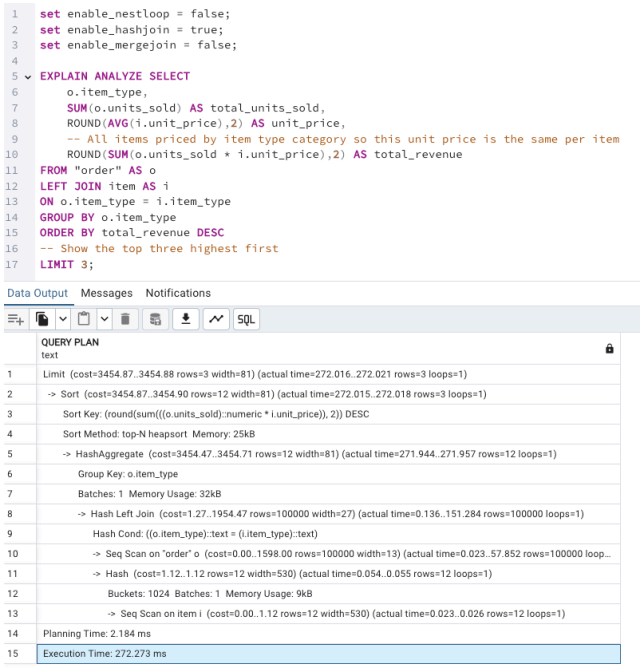
The first consideration in optimizations includes whether indexed columns can make querying faster. Indexing a non-primary key column establishes references to quickly search values based on the index (Kaur, 2024). PostgreSQL indexes generally optimize queries using the WHERE clause, sometimes in conjunction with ORDER BY and LIMIT (*PostgreSQL 16.6 Documentation*, 2024). Since the business queries for this project do not consist of WHERE clauses, indexing cannot significantly optimize them. Additionally, most columns used to join and order tables are aggregated, which cannot be indexed or are already indexed since they are primary keys.

Join tuning involves utilizing the EXPLAIN ANALYZE command to generate a query plan and observe which JOIN algorithm executes the script. We can enable different JOIN algorithms and explicitly tell PostgreSQL which one to use while observing the query plan to determine if a more efficient algorithm is available (Sullivan, 2023a). For this project, improvements made with join tuning, if any, were slight across hash join as the default, nest loop, and merge join, as demonstrated in Figures 22 to 24 for all three queries. The default hash join algorithm proved the most efficient process based on execution times.

A third option involves refactoring the query script to be more performant. For this project, generating materialized views can save a lot of execution time. The business queries for this project are complex common table expressions with aggregations, joins, subqueries, and window functions. Every time common table expressions execute, they must go through all its operations to generate the results. However, query results saved as materialized views act as tables to minimize these operations, allowing PostgreSQL to access data quickly.

Figure 22

Join Tuning Results: Business Query 1 (Top to Bottom: Hash Join, Nest Loop, Merge Join)



```
set enable_nestloop = false;
set enable_hashjoin = true;
set enable_mergejoin = false;

EXPLAIN ANALYZE SELECT
    o.item_type,
    SUM(o.units_sold) AS total_units_sold,
    ROUND(AVG(i.unit_price),2) AS unit_price,
    -- All items priced by item type category so this unit price is the same per item
    ROUND(SUM(o.units_sold * i.unit_price),2) AS total_revenue
FROM "order" AS o
LEFT JOIN item AS i
ON o.item_type = i.item_type
GROUP BY o.item_type
ORDER BY total_revenue DESC
-- Show the top three highest first
LIMIT 3;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=3454.87..3454.88 rows=3 width=81) (actual time=272.016..272.021 rows=3 loops=1)
2	-> Sort (cost=3454.87..3454.90 rows=12 width=81) (actual time=272.015..272.018 rows=3 loops=1)
3	Sort Key: (round(sum(((o.units_sold)::numeric * i.unit_price)), 2)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> HashAggregate (cost=3454.47..3454.71 rows=12 width=81) (actual time=271.944..271.957 rows=12 loops=1)
6	Group Key: o.item_type
7	Batches: 1 Memory Usage: 32kB
8	-> Hash Left Join (cost=1.27..1954.47 rows=100000 width=27) (actual time=0.136..151.284 rows=100000 loops=1)
9	Hash Cond: ((o.item_type)::text = (i.item_type)::text)
10	-> Seq Scan on "order" o (cost=0.00..1598.00 rows=100000 width=13) (actual time=0.023..57.852 rows=100000 loop...
11	-> Hash (cost=1.12..1.12 rows=12 width=530) (actual time=0.054..0.055 rows=12 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Seq Scan on item i (cost=0.00..1.12 rows=12 width=530) (actual time=0.023..0.026 rows=12 loops=1)
14	Planning Time: 2.184 ms
15	Execution Time: 272.273 ms

```
set enable_nestloop = true;
set enable_hashjoin = false;
set enable_mergejoin = false;

EXPLAIN ANALYZE SELECT
    o.item_type,
    SUM(o.units_sold) AS total_units_sold,
    ROUND(AVG(i.unit_price),2) AS unit_price,
    -- All items priced by item type category so this unit price is the same per item
    ROUND(SUM(o.units_sold * i.unit_price),2) AS total_revenue
FROM "order" AS o
LEFT JOIN item AS i
ON o.item_type = i.item_type
GROUP BY o.item_type
ORDER BY total_revenue DESC
-- Show the top three highest first
LIMIT 3;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=5494.59..5494.60 rows=3 width=81) (actual time=358.578..358.582 rows=3 loops=1)
2	-> Sort (cost=5494.59..5494.62 rows=12 width=81) (actual time=358.577..358.579 rows=3 loops=1)
3	Sort Key: (round(sum(((o.units_sold)::numeric * i.unit_price)), 2)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> HashAggregate (cost=5494.20..5494.44 rows=12 width=81) (actual time=358.538..358.557 rows=12 loops=1)
6	Group Key: o.item_type
7	Batches: 1 Memory Usage: 32kB
8	-> Nested Loop Left Join (cost=0.15..3994.20 rows=100000 width=27) (actual time=0.068..215.167 rows=100000 loops=1)
9	-> Seq Scan on "order" o (cost=0.00..1598.00 rows=100000 width=13) (actual time=0.026..29.982 rows=100000 loops=1)
10	-> Memoize (cost=0.15..0.16 rows=1 width=530) (actual time=0.001..0.001 rows=1 loops=100000)
11	Cache Key: o.item_type
12	Cache Mode: logical
13	Hits: 99988 Misses: 12 Evictions: 0 Overflows: 0 Memory Usage: 2kB
14	-> Index Scan using item_pkey on item i (cost=0.14..0.15 rows=1 width=530) (actual time=0.005..0.005 rows=1 loops...
15	Index Cond: ((item_type)::text = (o.item_type)::text)
16	Planning Time: 0.323 ms
17	Execution Time: 358.729 ms

```
set enable_nestloop = false;
set enable_hashjoin = false;
set enable_mergejoin = true;

EXPLAIN ANALYZE SELECT
    o.item_type,
    SUM(o.units_sold) AS total_units_sold,
    ROUND(AVG(i.unit_price),2) AS unit_price,
    -- All items priced by item type category so this unit price is the same per item
    ROUND(SUM(o.units_sold * i.unit_price),2) AS total_revenue
FROM "order" AS o
LEFT JOIN item AS i
ON o.item_type = i.item_type
GROUP BY o.item_type
ORDER BY total_revenue DESC
-- Show the top three highest first
LIMIT 3;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=12904.61..12904.62 rows=3 width=81) (actual time=298.139..298.143 rows=3 loops=1)
2	-> Sort (cost=12904.61..12904.64 rows=12 width=81) (actual time=298.138..298.140 rows=3 loops=1)
3	Sort Key: (round(sum(((o.units_sold)::numeric * i.unit_price)), 2)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> GroupAggregate (cost=9904.16..12904.46 rows=12 width=81) (actual time=137.307..298.099 rows=12 loops=1)
6	Group Key: o.item_type
7	-> Merge Left Join (cost=9904.16..11404.22 rows=100000 width=27) (actual time=129.050..219.616 rows=100000 loops=1)
8	Merge Cond: ((o.item_type)::text = (i.item_type)::text)
9	-> Sort (cost=9902.82..10152.82 rows=100000 width=13) (actual time=128.989..167.056 rows=100000 loops=1)
10	Sort Key: o.item_type
11	Sort Method: external merge Disk: 2456kB
12	-> Seq Scan on "order" o (cost=0.00..1598.00 rows=100000 width=13) (actual time=0.947..57.670 rows=100000 loop...
13	-> Sort (cost=1.34..1.37 rows=12 width=530) (actual time=0.055..0.062 rows=12 loops=1)
14	Sort Key: i.item_type
15	Sort Method: quicksort Memory: 25kB
16	-> Seq Scan on item i (cost=0.00..1.12 rows=12 width=530) (actual time=0.020..0.023 rows=12 loops=1)
17	Planning Time: 0.303 ms
18	Execution Time: 299.470 ms

Figure 23

Join Tuning Results: Business Query 2 (Top to Bottom: Hash Join, Nest Loop, Merge Join)

```
set enable_nestloop = false;
set enable_hashjoin = true;
set enable_mergejoin = false;

-- Which items sold the most (by units) per quarter?

EXPLAIN ANALYZE WITH item_units_sold_by_quarter AS (
SELECT
    item_type,
    -- EXTRACT function generates calendar quarter based on datetime (Q1, Q2, Q3, Q4)
    EXTRACT(quarter FROM order_date) AS calendar_quarter,
    -- Adds units of items sold by quarter
    SUM(o.units_sold) AS total_units_sold
FROM "order" AS o
LEFT JOIN order_detail AS od
ON o.order_id = od.order_id
GROUP BY item_type, calendar_quarter
```

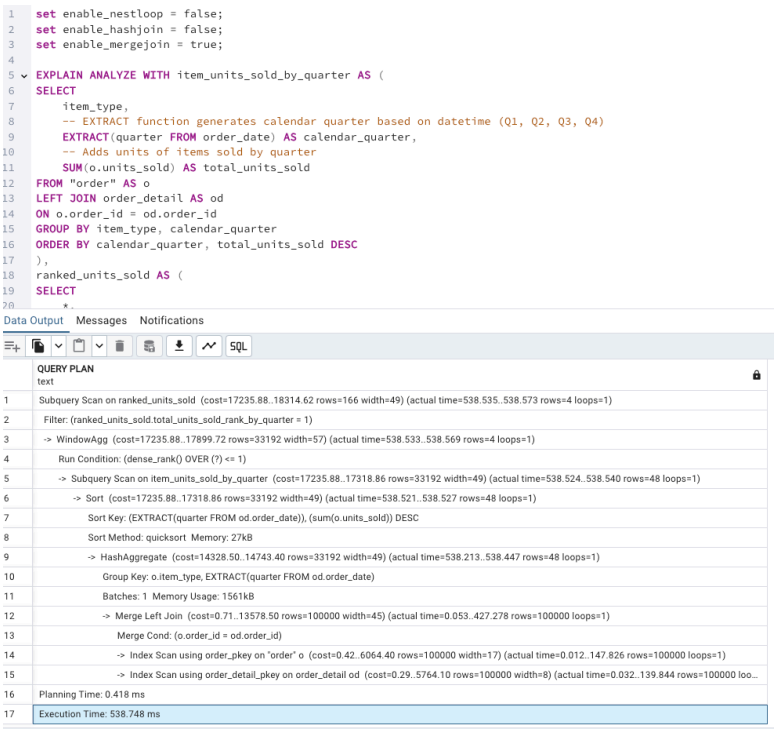
Data Output | Messages | Notifications

	QUERY PLAN text
1	Subquery Scan on ranked_units_sold (cost=8753.89..9832.63 rows=166 width=49) (actual time=482.296..482.335 rows=4 loops=1)
2	Filter: (ranked_units_sold.total_units_sold_rank_by_quarter = 1)
3	-> WindowAgg (cost=8753.89..9417.73 rows=33192 width=57) (actual time=482.295..482.332 rows=4 loops=1)
4	Run Condition: (dense_rank() OVER (?) <= 1)
5	-> Subquery Scan on item_units_sold_by_quarter (cost=8753.89..8836.87 rows=33192 width=49) (actual time=482.267..482.284 rows=48 l...
6	-> Sort (cost=8753.89..8836.87 rows=33192 width=49) (actual time=482.265..482.272 rows=48 loops=1)
7	Sort Key: (EXTRACT(quarter FROM od.order_date)), (sum(o.units_sold)) DESC
8	Sort Method: quicksort Memory: 27kB
9	-> HashAggregate (cost=5846.51..6261.41 rows=33192 width=49) (actual time=481.890..482.182 rows=48 loops=1)
10	Group Key: o.item_type, EXTRACT(quarter FROM od.order_date)
11	Batches: 1 Memory Usage: 1561kB
12	-> Hash Left Join (cost=2986.00..5096.51 rows=100000 width=45) (actual time=58.155..403.970 rows=100000 loops=1)
13	Hash Cond: (o.order_id = od.order_id)
14	-> Seq Scan on "order" o (cost=0.00..1598.00 rows=100000 width=17) (actual time=0.029..24.775 rows=100000 loops=1)
15	-> Hash (cost=1736.00..1736.00 rows=100000 width=8) (actual time=55.590..55.591 rows=100000 loops=1)
16	Buckets: 131072 Batches: 1 Memory Usage: 4931kB
17	-> Seq Scan on order_detail od (cost=0.00..1736.00 rows=100000 width=8) (actual time=0.020..20.531 rows=100000 loop...
18	Planning Time: 0.500 ms
19	Execution Time: 484.634 ms

```
set enable_nestloop = true;
set enable_hashjoin = false;
set enable_mergejoin = false;

EXPLAIN ANALYZE WITH item_units_sold_by_quarter AS (
SELECT
    item_type,
    -- EXTRACT function generates calendar quarter based on datetime (Q1, Q2, Q3, Q4)
    EXTRACT(quarter FROM order_date) AS calendar_quarter,
    -- Adds units of items sold by quarter
    SUM(o.units_sold) AS total_units_sold
FROM "order" AS o
LEFT JOIN order_detail AS od
ON o.order_id = od.order_id
GROUP BY item_type, calendar_quarter
ORDER BY calendar_quarter, total_units_sold DESC
),
ranked_units_sold AS (
SELECT
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Subquery Scan on ranked_units_sold (cost=40769.32..41848.06 rows=166 width=49) (actual time=664.794..664.842 rows=4 loops=1)
2	Filter: (ranked_units_sold.total_units_sold_rank_by_quarter = 1)
3	-> WindowAgg (cost=40769.32..41433.16 rows=33192 width=57) (actual time=664.792..664.838 rows=4 loops=1)
4	Run Condition: (dense_rank() OVER (?) <= 1)
5	-> Subquery Scan on item_units_sold_by_quarter (cost=40769.32..40852.30 rows=33192 width=49) (actual time=664.739..664.755 rows=48 loops=1)
6	-> Sort (cost=40769.32..40852.30 rows=33192 width=49) (actual time=664.737..664.743 rows=48 loops=1)
7	Sort Key: (EXTRACT(quarter FROM od.order_date)), (sum(o.units_sold)) DESC
8	Sort Method: quicksort Memory: 27kB
9	-> HashAggregate (cost=37861.94..38276.84 rows=33192 width=49) (actual time=664.417..664.640 rows=48 loops=1)
10	Group Key: o.item_type, EXTRACT(quarter FROM od.order_date)
11	Batches: 1 Memory Usage: 1561kB
12	-> Nested Loop Left Join (cost=0.29..37111.94 rows=100000 width=45) (actual time=0.105..577.549 rows=100000 loops=1)
13	-> Seq Scan on "order" o (cost=0.00..1598.00 rows=100000 width=17) (actual time=0.059..19.206 rows=100000 loops=1)
14	-> Index Scan using order_detail_pkey on order_detail od (cost=0.29..0.35 rows=1 width=8) (actual time=0.005..0.005 rows=1 loops=10...
15	Index Cond: (order_id = o.order_id)
16	Planning Time: 0.690 ms
17	Execution Time: 665.251 ms



```
set enable_nestloop = false;
set enable_hashjoin = false;
set enable_mergejoin = true;

EXPLAIN ANALYZE WITH item_units_sold_by_quarter AS (
SELECT
    item_type,
    -- EXTRACT function generates calendar quarter based on datetime (Q1, Q2, Q3, Q4)
    EXTRACT(quarter FROM order_date) AS calendar_quarter,
    -- Adds units of items sold by quarter
    SUM(o.units_sold) AS total_units_sold
FROM "order" AS o
LEFT JOIN order_detail AS od
ON o.order_id = od.order_id
GROUP BY item_type, calendar_quarter
ORDER BY calendar_quarter, total_units_sold DESC
),
ranked_units_sold AS (
SELECT
    *,
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Subquery Scan on ranked_units_sold (cost=17235.88..18314.62 rows=166 width=49) (actual time=538.535..538.573 rows=4 loops=1)
2	Filter: (ranked_units_sold.total_units_sold_rank_by_quarter = 1)
3	-> WindowAgg (cost=17235.88..17899.72 rows=33192 width=57) (actual time=538.533..538.569 rows=4 loops=1)
4	Run Condition: (dense_rank() OVER (?) <= 1)
5	-> Subquery Scan on item_units_sold_by_quarter (cost=17235.88..17318.86 rows=33192 width=49) (actual time=538.524..538.540 rows=48 loops=1)
6	-> Sort (cost=17235.88..17318.86 rows=33192 width=49) (actual time=538.521..538.527 rows=48 loops=1)
7	Sort Key: (EXTRACT(quarter FROM od.order_date)), (sum(o.units_sold)) DESC
8	Sort Method: quicksort Memory: 27kB
9	-> HashAggregate (cost=14328.50..14743.40 rows=33192 width=49) (actual time=538.213..538.447 rows=48 loops=1)
10	Group Key: o.item_type, EXTRACT(quarter FROM od.order_date)
11	Batches: 1 Memory Usage: 1561kB
12	-> Merge Left Join (cost=0.71..13578.50 rows=100000 width=45) (actual time=0.053..427.278 rows=100000 loops=1)
13	Merge Cond: (o.order_id = od.order_id)
14	-> Index Scan using order_pkey on "order" o (cost=0.42..6064.40 rows=100000 width=17) (actual time=0.012..147.826 rows=100000 loops=1)
15	-> Index Scan using order_detail_pkey on order_detail od (cost=0.29..5764.10 rows=100000 width=8) (actual time=0.032..139.844 rows=100000 loo…
16	Planning Time: 0.418 ms
17	Execution Time: 538.748 ms

Figure 24

Join Tuning Results: Business Query 3 (Top to Bottom: Hash Join, Nest Loop, Merge Join)

Query | Query History

```
set enable_nestloop = false;
set enable_hashjoin = true;
set enable_mergejoin = false;

EXPLAIN ANALYZE WITH country_region_by_order_id AS (
    SELECT
        l.region,
        od.order_id
    FROM order_detail AS od
    LEFT JOIN "location" AS l
    ON od.country = l.country
),
price_units_sold_by_region AS (
    SELECT
    i.unit_price,
    o.units_sold,
```

Data Output | Messages | Notifications

```
QUERY PLAN
text
Group Key: l.region
Batches: 1  Memory Usage: 24kB
-> Hash Left Join  (cost=2993.43..5477.67 rows=100000 width=34) (actual time=60.313..267.480 rows=100000 loops=1)
Hash Cond: ((od.country)::text = (l.country)::text)
-> Hash Left Join  (cost=2987.27..5202.99 rows=100000 width=27) (actual time=56.836..207.609 rows=100000 loops=1)
Hash Cond: (o.order_id = od.order_id)
-> Hash Left Join  (cost=1.27..1954.47 rows=100000 width=22) (actual time=0.090..87.372 rows=100000 loops=1)
Hash Cond: ((o.item_type)::text = (i.item_type)::text)
-> Seq Scan on "order" o  (cost=0.00..1598.00 rows=100000 width=17) (actual time=0.015..21.219 rows=100000 loops=1)
-> Hash  (cost=1.12..1.12 rows=12 width=530) (actual time=0.031..0.033 rows=12 loops=1)
Buckets: 1024  Batches: 1  Memory Usage: 9kB
-> Seq Scan on item i  (cost=0.00..1.12 rows=12 width=530) (actual time=0.018..0.020 rows=12 loops=1)
-> Hash  (cost=1736.00..1736.00 rows=100000 width=13) (actual time=56.669..56.670 rows=100000 loops=1)
Buckets: 131072  Batches: 1  Memory Usage: 5648kB
-> Seq Scan on order_detail od  (cost=0.00..1736.00 rows=100000 width=13) (actual time=0.018..24.146 rows=100000 loop…
-> Hash  (cost=3.85..3.85 rows=185 width=25) (actual time=3.407..3.407 rows=185 loops=1)
Buckets: 1024  Batches: 1  Memory Usage: 19kB
-> Seq Scan on location l  (cost=0.00..3.85 rows=185 width=25) (actual time=1.363..1.928 rows=185 loops=1)
Planning Time: 5.848 ms
Execution Time: 370.151 ms
```

Total rows: 24 of 24 Query complete 00:00:00.426 Ln 9, Col 25

```
set enable_nestloop = true;
set enable_hashjoin = false;
set enable_mergejoin = false;

EXPLAIN ANALYZE WITH country_region_by_order_id AS (
    SELECT
        l.region,
        od.order_id
    FROM order_detail AS od
    LEFT JOIN "location" AS l
    ON od.country = l.country
),
price_units_sold_by_region AS (
    SELECT
    i.unit_price,
    o.units_sold,
    cr_id.region
    FROM
        "order" AS o
```

Data Output | Messages | Notifications

```
QUERY PLAN
text
-> Seq Scan on "order" o  (cost=0.00..1598.00 rows=100000 width=17) (actual time=0.063..22.275 rows=100000 loops=1)
-> Memoize  (cost=0.15..0.16 rows=1 width=530) (actual time=0.000..0.000 rows=1 loops=100000)
Cache Key: o.item_type
Cache Mode: logical
Hits: 99988  Misses: 12  Evictions: 0  Overflows: 0  Memory Usage: 2kB
-> Index Scan using item_pkey on item i  (cost=0.14..0.15 rows=1 width=530) (actual time=0.010..0.010 rows=1 loops=12)
Index Cond: ((item_type)::text = (o.item_type)::text)
-> Index Scan using order_detail_pkey on order_detail od  (cost=0.29..0.35 rows=1 width=13) (actual time=0.004..0.004 rows=1 loops=100…
Index Cond: (order_id = o.order_id)
-> Memoize  (cost=0.15..0.17 rows=1 width=25) (actual time=0.001..0.001 rows=1 loops=100000)
Cache Key: od.country
Cache Mode: logical
Hits: 99815  Misses: 185  Evictions: 0  Overflows: 0  Memory Usage: 24kB
-> Index Scan using location_pkey on location l  (cost=0.14..0.16 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=185)
Index Cond: ((country)::text = (od.country)::text)
Planning Time: 0.914 ms
Execution Time: 904.354 ms
```

```
set enable_nestloop = false;
set enable_hashjoin = false;
set enable_mergejoin = true;

EXPLAIN ANALYZE WITH country_region_by_order_id AS (
    SELECT
        l.region,
        od.order_id
    FROM order_detail AS od
    LEFT JOIN "location" AS l
    ON od.country = l.country
),
price_units_sold_by_region AS (
    SELECT
    i.unit_price,
    o.units_sold,
    cr_id.region
    FROM
        "order" AS o
    LEFT JOIN
```

Data Output | Messages | Notifications

	QUERY PLAN text
12	-> Merge Left Join (cost=16441.86..24505.84 rows=100000 width=29) (actual time=245.244..492.460 rows=100000 loops=1)
13	Merge Cond: (o.order_id = od.order_id)
14	-> Index Scan using order_pkey on "order" o (cost=0.42..6064.40 rows=100000 width=17) (actual time=0.015..110.525 rows=100000 loops=1)
15	-> Materialize (cost=16441.44..16941.44 rows=100000 width=20) (actual time=245.221..302.349 rows=100000 loops=1)
16	-> Sort (cost=16441.44..16691.44 rows=100000 width=20) (actual time=245.215..267.511 rows=100000 loops=1)
17	Sort Key: od.order_id
18	Sort Method: external merge Disk: 3024kB
19	-> Merge Left Join (cost=0.44..6085.62 rows=100000 width=20) (actual time=0.035..124.306 rows=100000 loops=1)
20	Merge Cond: ((od.country)::text = (l.country)::text)
21	-> Index Scan using country_idx on order_detail od (cost=0.29..4816.24 rows=100000 width=13) (actual time=0.020..86.666 rows=100000 loo…
22	-> Index Scan using location_pkey on location l (cost=0.14..18.92 rows=185 width=25) (actual time=0.012..0.146 rows=185 loops=1)
23	-> Sort (cost=1.34..1.37 rows=12 width=530) (actual time=0.033..0.039 rows=12 loops=1)
24	Sort Key: i.item_type
25	Sort Method: quicksort Memory: 25kB
26	-> Seq Scan on item i (cost=0.00..1.12 rows=12 width=530) (actual time=0.016..0.019 rows=12 loops=1)
27	Planning Time: 0.738 ms
28	Execution Time: 786.412 ms

Common table expressions executed in business queries are helpful for data that must be updated frequently, such as when calculating weekly transactions. However, materialized views are more appropriate for this project's business queries, which look at historical trends with data that only needs to be updated a few times a year. Materialized views are also suitable for multiple users needing access to the results data, such as the sales, marketing, and strategy teams, who may quickly need the results saved in a specific format for further querying. Refactoring the common table expressions to reusable and indexable materialized views can improve query performance. Figure 25 demonstrates the script to generate the materialized views for the business questions. Figures 26 to 28 show the performance difference between the original queries with common table expressions and those utilized to query the materialized views.

Figure 25

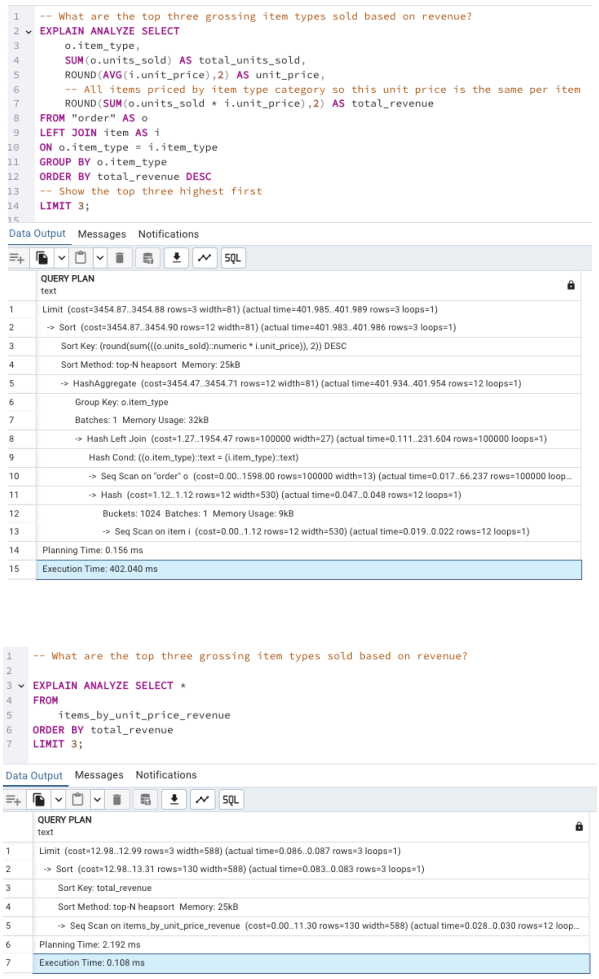
Materialized Views

```sql
-- What are the top three grossing item types sold based on revenue?

CREATE MATERIALIZED VIEW items_by_unit_price_revenue AS (
-- What are the top three grossing item types sold based on revenue?
SELECT
    o.item_type,
    SUM(o.units_sold) AS total_units_sold,
    ROUND(AVG(i.unit_price),2) AS unit_price,
    -- All items priced by item type category so this unit price is the same per item
    ROUND(SUM(o.units_sold * i.unit_price),2) AS total_revenue
FROM "order" AS o
LEFT JOIN item AS i
ON o.item_type = i.item_type
GROUP BY o.item_type
);

-- Which items sold the most (by units) per quarter?

CREATE MATERIALIZED VIEW item_units_sold_by_quarter AS(
SELECT
    item_type,
    -- EXTRACT function generates calendar quarter based on datetime (Q1, Q2, Q3, Q4)
    EXTRACT(quarter FROM order_date) AS calendar_quarter,
    -- Adds units of items sold by quarter
    SUM(o.units_sold) AS total_units_sold
FROM "order" AS o
LEFT JOIN order_detail AS od
ON o.order_id = od.order_id
GROUP BY item_type, calendar_quarter
ORDER BY calendar_quarter, total_units_sold DESC
)

-- What is the total revenue per region?
CREATE MATERIALIZED VIEW units_sold_price_by_region AS (
    SELECT
        od.order_id,
        region,
        units_sold,
        unit_price
    FROM order_detail AS od
    INNER JOIN
        "location" AS l
    ON od.country = l.country
    INNER JOIN
        "order" AS o
    ON o.order_id = od.order_id
    INNER JOIN
        item AS i
    ON o.item_type = i.item_type
)
```

Figure 26

Optimization Results: Business Query 1

(Top to Bottom: Plan Before Optimization, Plan After Implementing Materialized View)



```
-- What are the top three grossing item types sold based on revenue?
EXPLAIN ANALYZE SELECT
    o.item_type,
    SUM(o.units_sold) AS total_units_sold,
    ROUND(AVG(i.unit_price),2) AS unit_price,
    -- All items priced by item type category so this unit price is the same per item
    ROUND(SUM(o.units_sold * i.unit_price),2) AS total_revenue
FROM "order" AS o
LEFT JOIN item AS i
ON o.item_type = i.item_type
GROUP BY o.item_type
ORDER BY total_revenue DESC
-- Show the top three highest first
LIMIT 3;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=3454.87..3454.88 rows=3 width=81) (actual time=401.985..401.989 rows=3 loops=1)
2	-> Sort (cost=3454.87..3454.90 rows=12 width=81) (actual time=401.983..401.986 rows=3 loops=1)
3	Sort Key: (round(sum(((o.units_sold)::numeric * i.unit_price)), 2)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> HashAggregate (cost=3454.47..3454.71 rows=12 width=81) (actual time=401.934..401.954 rows=12 loops=1)
6	Group Key: o.item_type
7	Batches: 1 Memory Usage: 32kB
8	-> Hash Left Join (cost=1.27..1954.47 rows=100000 width=27) (actual time=0.111..231.604 rows=100000 loops=1)
9	Hash Cond: ((o.item_type)::text = (i.item_type)::text)
10	-> Seq Scan on "order" o (cost=0.00..1598.00 rows=100000 width=13) (actual time=0.017..66.237 rows=100000 loop...
11	-> Hash (cost=1.12..1.12 rows=12 width=530) (actual time=0.047..0.048 rows=12 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Seq Scan on item i (cost=0.00..1.12 rows=12 width=530) (actual time=0.019..0.022 rows=12 loops=1)
14	Planning Time: 0.156 ms
15	Execution Time: 402.040 ms

```
-- What are the top three grossing item types sold based on revenue?

EXPLAIN ANALYZE SELECT *
FROM
    items_by_unit_price_revenue
ORDER BY total_revenue
LIMIT 3;
```

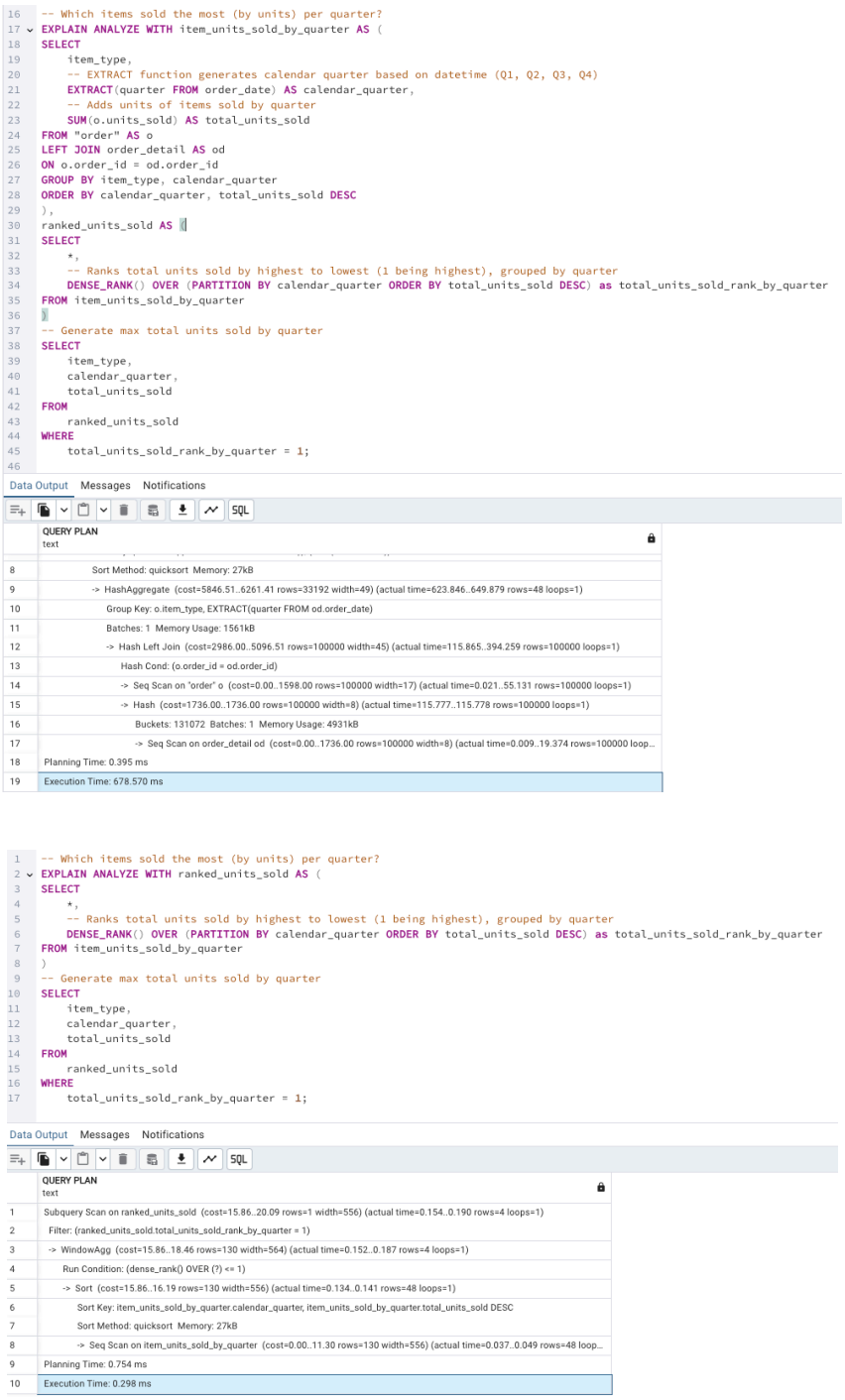
Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=12.98..12.99 rows=3 width=588) (actual time=0.086..0.087 rows=3 loops=1)
2	-> Sort (cost=12.98..13.31 rows=130 width=588) (actual time=0.083..0.083 rows=3 loops=1)
3	Sort Key: total_revenue
4	Sort Method: top-N heapsort Memory: 25kB
5	-> Seq Scan on items_by_unit_price_revenue (cost=0.00..11.30 rows=130 width=588) (actual time=0.028..0.030 rows=12 loop...
6	Planning Time: 2.192 ms
7	Execution Time: 0.108 ms

Figure 27

Optimization Results: Business Query 2

(Top to Bottom: Plan Before Optimization, Plan After Implementing Materialized View)



```
-- Which items sold the most (by units) per quarter?
EXPLAIN ANALYZE WITH item_units_sold_by_quarter AS (
SELECT
    item_type,
    -- EXTRACT function generates calendar quarter based on datetime (Q1, Q2, Q3, Q4)
    EXTRACT(quarter FROM order_date) AS calendar_quarter,
    -- Adds units of items sold by quarter
    SUM(o.units_sold) AS total_units_sold
FROM "order" AS o
LEFT JOIN order_detail AS od
ON o.order_id = od.order_id
GROUP BY item_type, calendar_quarter
ORDER BY calendar_quarter, total_units_sold DESC
),
ranked_units_sold AS (
SELECT
    *,
    -- Ranks total units sold by highest to lowest (1 being highest), grouped by quarter
    DENSE_RANK() OVER (PARTITION BY calendar_quarter ORDER BY total_units_sold DESC) as total_units_sold_rank_by_quarter
FROM item_units_sold_by_quarter
)
-- Generate max total units sold by quarter
SELECT
    item_type,
    calendar_quarter,
    total_units_sold
FROM
    ranked_units_sold
WHERE
    total_units_sold_rank_by_quarter = 1;
```

Data Output | Messages | Notifications

	QUERY PLAN text
8	Sort Method: quicksort Memory: 27kB
9	-> HashAggregate (cost=5846.51..6261.41 rows=33192 width=49) (actual time=623.846..649.879 rows=48 loops=1)
10	Group Key: o.item_type, EXTRACT(quarter FROM od.order_date)
11	Batches: 1 Memory Usage: 1561kB
12	-> Hash Left Join (cost=2986.00..5096.51 rows=100000 width=45) (actual time=115.865..394.259 rows=100000 loops=1)
13	Hash Cond: (o.order_id = od.order_id)
14	-> Seq Scan on "order" o (cost=0.00..1598.00 rows=100000 width=17) (actual time=0.021..55.131 rows=100000 loops=1)
15	-> Hash (cost=1736.00..1736.00 rows=100000 width=8) (actual time=115.777..115.778 rows=100000 loops=1)
16	Buckets: 131072 Batches: 1 Memory Usage: 4931kB
17	-> Seq Scan on order_detail od (cost=0.00..1736.00 rows=100000 width=8) (actual time=0.009..19.374 rows=100000 loop...
18	Planning Time: 0.395 ms
19	Execution Time: 678.570 ms

```
-- Which items sold the most (by units) per quarter?
EXPLAIN ANALYZE WITH ranked_units_sold AS (
SELECT
    *,
    -- Ranks total units sold by highest to lowest (1 being highest), grouped by quarter
    DENSE_RANK() OVER (PARTITION BY calendar_quarter ORDER BY total_units_sold DESC) as total_units_sold_rank_by_quarter
FROM item_units_sold_by_quarter
)
-- Generate max total units sold by quarter
SELECT
    item_type,
    calendar_quarter,
    total_units_sold
FROM
    ranked_units_sold
WHERE
    total_units_sold_rank_by_quarter = 1;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Subquery Scan on ranked_units_sold (cost=15.86..20.09 rows=1 width=556) (actual time=0.154..0.190 rows=4 loops=1)
2	Filter: (ranked_units_sold.total_units_sold_rank_by_quarter = 1)
3	-> WindowAgg (cost=15.86..18.46 rows=130 width=564) (actual time=0.152..0.187 rows=4 loops=1)
4	Run Condition: (dense_rank() OVER (?) <= 1)
5	-> Sort (cost=15.86..16.19 rows=130 width=556) (actual time=0.134..0.141 rows=48 loops=1)
6	Sort Key: item_units_sold_by_quarter.calendar_quarter, item_units_sold_by_quarter.total_units_sold DESC
7	Sort Method: quicksort Memory: 27kB
8	-> Seq Scan on item_units_sold_by_quarter (cost=0.00..11.30 rows=130 width=556) (actual time=0.037..0.049 rows=48 loop...
9	Planning Time: 0.754 ms
10	Execution Time: 0.298 ms

Figure 28

Optimization Results: Business Query 1

(Top to Bottom: Plan Before Optimization, Plan After Implementing Materialized View)

```sql
-- What is the total revenue per region?
EXPLAIN ANALYZE WITH country_region_by_order_id AS (
    SELECT
        l.region,
        od.order_id
    FROM order_detail AS od
    LEFT JOIN "location" AS l
    ON od.country = l.country
),
price_units_sold_by_region AS (
    SELECT
    i.unit_price,
    o.units_sold,
    cr_id.region
    FROM
        "order" AS o
    LEFT JOIN
        item AS i
    ON
        o.item_type = i.item_type
    LEFT JOIN
        country_region_by_order_id as cr_id
    ON
        o.order_id = cr_id.order_id
)
SELECT
```

Data Output | Messages | Notifications

	QUERY PLAN text
13	-> Seq Scan on "order" o (cost=0.00..1598.00 rows=100000 width=17) (actual time=0.009..20.995 rows=100000 loops=1)
14	-> Hash (cost=1.12..1.12 rows=12 width=530) (actual time=0.017..0.017 rows=12 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 9kB
16	-> Seq Scan on item i (cost=0.00..1.12 rows=12 width=530) (actual time=0.006..0.009 rows=12 loops=1)
17	-> Hash (cost=1736.00..1736.00 rows=100000 width=13) (actual time=165.775..165.776 rows=100000 loops=1)
18	Buckets: 131072 Batches: 1 Memory Usage: 5648kB
19	-> Seq Scan on order_detail od (cost=0.00..1736.00 rows=100000 width=13) (actual time=0.007..25.079 rows=100000 loop...
20	-> Hash (cost=3.85..3.85 rows=185 width=25) (actual time=0.114..0.115 rows=185 loops=1)
21	Buckets: 1024 Batches: 1 Memory Usage: 19kB
22	-> Seq Scan on location l (cost=0.00..3.85 rows=185 width=25) (actual time=0.020..0.051 rows=185 loops=1)
23	Planning Time: 0.567 ms
24	Execution Time: 755.329 ms

```sql
-- What is the total revenue per region?
EXPLAIN ANALYZE SELECT
    SUM(units_sold) AS total_units_sold,
    -- AVG unit price is the average price of all units regardless of item type
    ROUND(AVG(unit_price),2) AS avg_unit_price,
    ROUND(SUM(units_sold * unit_price),2) AS total_revenue,
    region
FROM units_sold_price_by_region
GROUP BY
    region
ORDER BY
    total_revenue DESC;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=3332.24..3332.26 rows=7 width=88) (actual time=244.836..244.838 rows=7 loops=1)
2	Sort Key: (round(sum(((units_sold)::numeric * unit_price)), 2)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (cost=3332.00..3332.14 rows=7 width=88) (actual time=244.817..244.825 rows=7 loops=1)
5	Group Key: region
6	Batches: 1 Memory Usage: 24kB
7	-> Seq Scan on units_sold_price_by_region (cost=0.00..1832.00 rows=100000 width=27) (actual time=0.014..73.679 rows=100000 loop...
8	Planning Time: 0.280 ms
9	Execution Time: 244.876 ms

Using materialized views instead of common table expressions significantly improved the run times of queries. The queries are now directly querying a "virtual" result table instead of having to generate CTEs and query sequentially. Adding an index to a materialized view can also improve performance when querying with a WHERE clause. Furthermore, users can also update materialized views with added data using the REFRESH MATERIALIZED VIEW command.

The underlying tables related to the materialized views will update these results tables with their new data. Materialized views that focus on answering general business questions and allow users to quickly produce intermittent trend reports for analysis are powerful tools for any growing business. EcoMart will benefit from this specialized database system because it encompasses flexibility, growth, and strategic analysis requirements.

G. References

1. Abhijit. (2024, November 22). *The complete overview of big data*. Intellipaat. https://intellipaat.com/blog/tutorial/hadoop-tutorial/big-data-overview/
2. Babitz, K. (2024, December 10). *How to maintain data security*. www.datacamp.com. Retrieved December 12, 2024, from https://www.datacamp.com/blog/how-to-maintain-data-security
3. Coronel, C., & Morris, S. (2022). *Database systems*.
4. Decomplexify. (2021, November 21). *Learn Database Normalization - 1NF, 2NF, 3NF, 4NF, 5NF* [Video]. YouTube. https://www.youtube.com/watch?v=GFQaEYEc8_8
5. Goyal, A. (2024, November 29). Strategies for Scaling Databases: A Comprehensive guide. *Medium*. https://medium.com/@anil.goyal0057/strategies-for-scaling-databases-a-comprehensive-guide-b69cda7df1d3#:~:text=Vertical%20Scaling%3A%20This%20involves%20adding,downtime%20to%20upgrade%20the%20server.
6. Hardy-Françon, G. (2024, May 13). *Explore Postgres Enums and how to use them*. Forest Admin Blog. https://www.forestadmin.com/blog/postgres-enums-dissected/

7. Kaur, T. (2024, November 30). Optimizing SQL Query Performance: A Comprehensive Guide. *Medium*. https://medium.com/womenintechnology/optimizing-sql-query-performance-a-comprehensive-guide-6cb72b9f52ef
8. Malik, U., Goldwasser, M., & Johnston, B. (2019). *SQL for Data Analytics: Perform Fast and Efficient Data Analysis with the Power of SQL*.
9. Marottickal, T. (2022, August 21). SQL | Data Types - Tom Marottickal - Medium. *Medium*. https://medium.com/@tom-.-me/sql-data-types-4476e394c40c
10. Measured®: Marketing Attribution & Incrementality Testing. (2024, August 13). *What are the Benefits of Data-Driven Marketing?* Measured®. https://www.measured.com/faq/the-benefits-of-data-driven-marketing-for-marketers/
11. *OLTP vs OLAP - Difference Between Data Processing Systems - AWS*. (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/compare/the-difference-between-olap-and-oltp/#:~:text=You%20use%20OLAP%20systems%20to,architecture%2C%20performance%2C%20and%20requirements.
12. Pant, A. (2024, November 25). Masking PII (Personally Identifiable Information) - Arvind Pant - Medium. *Medium*. https://medium.com/@arvindpant/masking-pii-personally-identifiable-information-300f0acebc78
13. *PostgreSQL 16.6 documentation*. (2024, November 21). PostgreSQL Documentation. https://www.postgresql.org/docs/16/index.html