

ASSIGNMENT COVER SHEET

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Course Code and Description | UECS 3203 ADVANCED DATABASE SYSTEMS | | | |
| Lecturer Name | Dr. Sugumaran a/l Nallusamy | | | |
| Assignment Title | Assignment 1 | | | |
| **DECLARATION** | | | | |
| We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part been written for us by another person. | | | | |
| Programme | | Student ID Numbers | Student Names | Practical  Group |
| SE | | 2002759 | AVEN DING XIAN KAI | 1 |
| SE | | 2400056 | CHOO WEI XIANG | 1 |
| SE | | 2301374 | SIA LI ZE | 1 |
| SE | | 2003632 | TH’NG ZI QIN | 1 |

Table of Contents

[CHAPTER 1 DATA SELECTION 4](#_Toc175311447)

[CHAPTER 2 DATA IMPORT 5](#_Toc175311448)

[2.1 Request the path and filename 5](#_Toc175311449)

[2.2 Validate File 6](#_Toc175311450)

[2.3 Read & Import Data from File 7](#_Toc175311451)

[CHAPTER 3 TABLE CREATION & INDEXES 8](#_Toc175311452)

[3.1 Table Structure 8](#_Toc175311453)

[3.2 Indexes Suggestion 10](#_Toc175311454)

[CHAPTER 4 CRUD OPERATIONS 12](#_Toc175311455)

[4.1 Create 12](#_Toc175311456)

[4.2 Read 20](#_Toc175311457)

[4.3 Update 22](#_Toc175311458)

[4.4 Delete 26](#_Toc175311459)

[CHAPTER 5 TRANSACTION MANAGEMENT 28](#_Toc175311460)

[CHAPTER 6 TRIGGERS 29](#_Toc175311461)

[6.1 Shipping\_Audit\_Table 29](#_Toc175311462)

[6.2 Shipping\_Audit\_Sequence 29](#_Toc175311463)

[6.3 Shipping\_Audit\_ID\_Trigger 29](#_Toc175311464)

[6.4 Shipping\_Audit\_Sequence 30](#_Toc175311465)

[6.5 Shipping\_After\_Insert\_Update 30](#_Toc175311466)

[6.6 Shipping\_Before\_Input\_Update 31](#_Toc175311467)

[6.7 OrderItem\_Before\_Input\_Update 32](#_Toc175311468)

[6.8 Shipping\_Before\_Delete 32](#_Toc175311469)

[6.9 Output: Shipping\_Audit 33](#_Toc175311470)

[CHAPTER 7 DATA ANALYSIS 34](#_Toc175311471)

[7.1 Sales Report for Monthly or Yearly 34](#_Toc175311472)

[7.1.1 Sales Report for Monthly or Yearly 34](#_Toc175311473)

[7.1.2 Procedure to Generate Report 35](#_Toc175311474)

[7.1.3 Sample Outputs 36](#_Toc175311475)

[7.2 Sales by Market Report 37](#_Toc175311476)

[7.2.1 Execution Way 37](#_Toc175311477)

[7.2.2 Procedure 37](#_Toc175311478)

[7.2.3 Sample Outputs 38](#_Toc175311479)

[7.3 Top 5 Sales Products in Each Market 39](#_Toc175311480)

[7.3.1 User Input Interface 39](#_Toc175311481)

[7.3.2 Procedure 39](#_Toc175311482)

[7.3.3 Sample Outputs 42](#_Toc175311483)

[7.4 Total Sales in Each Shipping Mode Report 43](#_Toc175311484)

[7.4.1 Way to Execute 43](#_Toc175311485)

[7.4.2 Procedure 43](#_Toc175311486)

[7.4.3 Sample Outputs 44](#_Toc175311487)

[7.5 End of Report 44](#_Toc175311488)

[7.5.1 User Input Interface 45](#_Toc175311489)

[7.5.2 Procedure 45](#_Toc175311490)

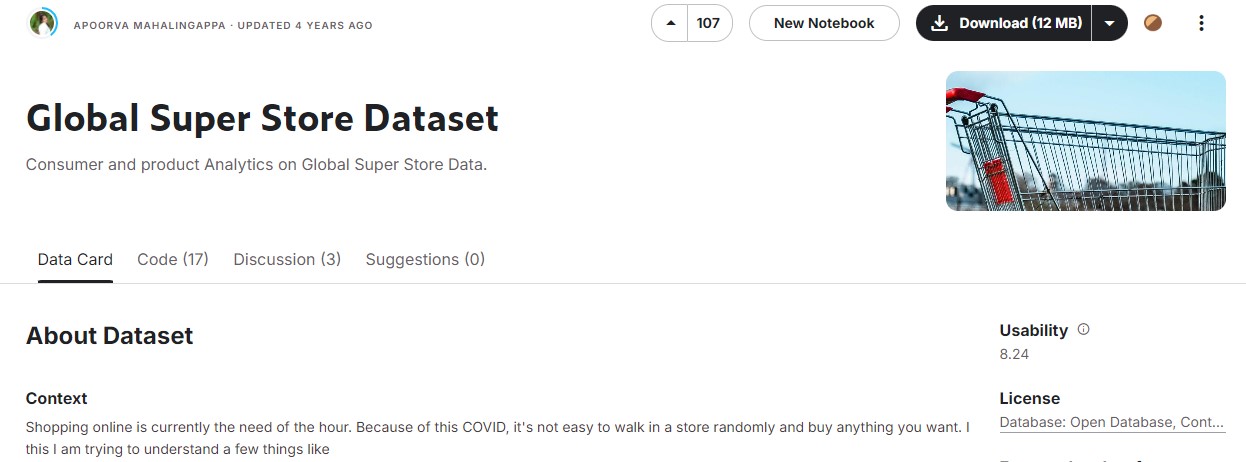
[7.5.3 Sample Outputs 48](#_Toc175311491)

[CHAPTER 8 REFERENCES 50](#_Toc175311492)

# DATA SELECTION

The Global Superstore Dataset was chosen by our group from Kaggle for this assignment. The dataset includes information on detailed order records, shipping details, customer profiles, market research, and product details. The data is contained in a single CSV file and is organized into 51,291 rows, each with 23 attributes.

The reason we decided on this dataset was that every single one of the columns had complete and rich data. The dataset is also appropriate from a variety of analytical angles because it provides a broad range of attributes. This versatility helps us achieve our goals for data analysis by enabling us to extract valuable insights from a variety of perspectives. This dataset is a great place for novices to start learning about e-commerce analytics in the field of data science. It offers lots of chances to extract substantial.



*Figure 1.1: Global Super Store Dataset* (*Global Super Store Dataset*, n.d.)

# DATA IMPORT

In this task, we have implemented PL/SQL programs to fulfill the needs of importing data from a CSV file into our Oracle database. These programs include a few steps which are to accept path and filename from a user, validate the existence and format of the CSV file, and import data with error handling.

## Request the path and filename

The user is first prompted to enter the path and filename of the CSV file that needs to be imported in this step. The file\_path and file\_name will be assigned to users once they have entered the path and name. To enable the database to access the path, a directory called my\_dir will be created in the database using the file\_path. Invokes the function validate\_file\_path(file\_name) to verify the validity and accessibility of the file.

SET SERVEROUTPUT ON

    ACCEPT file\_path char PROMPT "Enter File Path: ";

    ACCEPT file\_name char PROMPT "Enter Filename: ";

DECLARE

    file\_path VARCHAR2(255);

    file\_name VARCHAR2(255);

    result\_import VARCHAR2(100);

BEGIN

    file\_path := '&file\_path';  -- Prompt for input

    file\_name := '&file\_name';  -- Prompt for input

    -- Create the directory

    EXECUTE IMMEDIATE 'CREATE OR REPLACE DIRECTORY my\_dir AS '''

                      || file\_path

                      || '''';

    dbms\_output.put\_line('Directory my\_dir created successfully.');

    -- Validate the file

    IF validate\_file\_path(file\_name) THEN

        dbms\_output.put\_line('File is valid and accessible.');

        result\_import:= process\_csv\_file(file\_name);

        dbms\_output.put\_line(result\_import);

    ELSE

        dbms\_output.put\_line('File is invalid or inaccessible.');

    END IF;

EXCEPTION

    WHEN OTHERS THEN

        dbms\_output.put\_line('Error creating directory: ' || sqlerrm);

END;

/

## Validate File

The filename will be passed to a validate\_file\_path function to verify that the file exists and can be accessed before importing the data form. By running the *validate\_file\_path* function, the existence and accessibility of a CSV file from the input of the user will be verified. By reference to Oracle *UTL\_FILE* documentation (Kannan et al., 2024), this validation can be done by *UTL\_FILE.FOPEN* method. If the CSV file opens successfully, it will return true back. However, if any errors occur during this process such as an invalid path or operation, the exception function will log out to the user and return false. This can ensure that any issues that happen during validation can be caught by the system and feedback to the user.

create or replace FUNCTION validate\_file\_path(p\_file\_name IN VARCHAR2)

RETURN BOOLEAN

IS

    csv\_file    UTL\_FILE.FILE\_TYPE;

BEGIN

--    DBMS\_OUTPUT.PUT\_LINE(p\_file\_path);

    csv\_file := UTL\_FILE.FOPEN('MY\_DIR', p\_file\_name, 'R');

    UTL\_FILE.FCLOSE(csv\_file);

    RETURN TRUE;

EXCEPTION

    WHEN UTL\_FILE.INVALID\_PATH THEN

        DBMS\_OUTPUT.PUT\_LINE('Invalid directory path: MY\_DIR');

        RETURN FALSE;

    WHEN UTL\_FILE.INVALID\_MODE THEN

        DBMS\_OUTPUT.PUT\_LINE('Invalid file open mode.');

        RETURN FALSE;

    WHEN UTL\_FILE.INVALID\_OPERATION THEN

        DBMS\_OUTPUT.PUT\_LINE('Invalid operation.');

        RETURN FALSE;

    WHEN UTL\_FILE.READ\_ERROR THEN

        DBMS\_OUTPUT.PUT\_LINE('Read error.');

        RETURN FALSE;

    WHEN UTL\_FILE.ACCESS\_DENIED THEN

        DBMS\_OUTPUT.PUT\_LINE('Access denied.');

        RETURN FALSE;

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

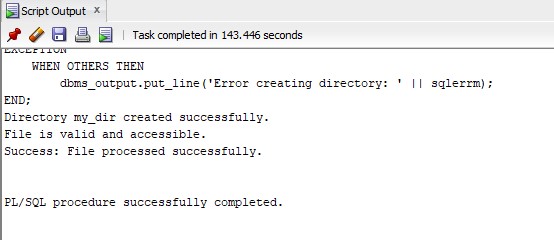
        RETURN FALSE;

END validate\_file\_path;

## Read & Import Data from File

Import the data using the process\_csv\_file function into the database if the file is valid. Data processing and import into database tables from a CSV file is the purpose of this function. The CSV file input by the user will be opened from the directory *my\_dir* and skip the header line. It will read each line from the CSV file and split the data by using the Oracle regular expression (12 Using Regular Expressions With Oracle Database, n.d.). One row of data will be split and inserted into four tables which are customer, shipping, product, and *orderitem* tables. Before inserting the data into the table, it will check the existing data with each id, *orderitem* will check with the order id and product id because one order may have multiple products.

During the operation to get file data and insert data into the table, a save point *savepoint\_before\_line* will be recorded and the errors will be handled by the exception function. It will give feedback to the user by logging out the error message for the user and roll back to the *savepoint\_before\_line*. Once the error for NOT\_DATA\_FOUND happens in the get line from the CSV file, the loop will end since it means the CSV file completes reading by function. It will return a success message to the user.

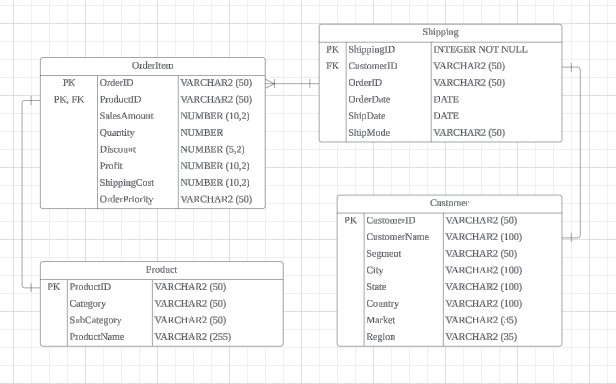


*Figure 2.1: Data Import to Tables*

# TABLE CREATION & INDEXES

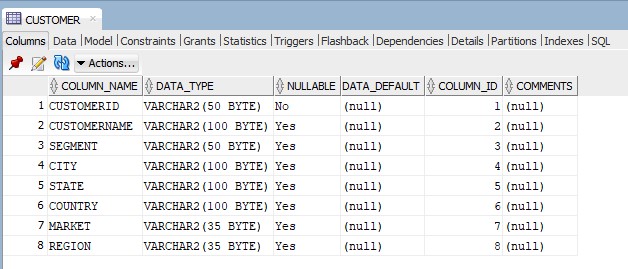
## Table Structure

The subsequent step after importing the dataset is to normalize the data into several tables to minimize redundancy and improve the efficiency of the analysis. The dataset was divided into four important tables, which we designated Customer, Shipping, Product, and OrderItem.



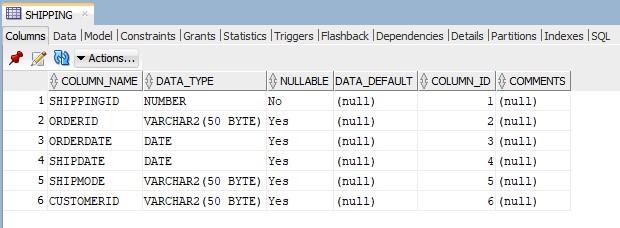
*Figure 3.1: Tables Structure*

The Customer table contains customer-related information, including CustomerID as the primary key, and fields such as CustomerName, Segment, City, State, Country, Market, and Region. This table stores all the necessary details about each customer, ensuring that customer data is organized and easily accessible.



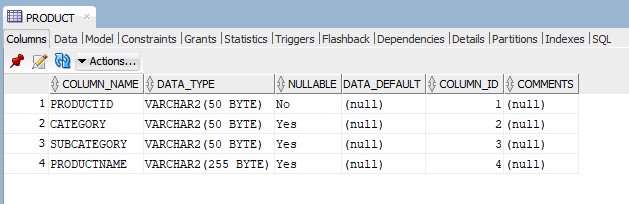
*Figure 3.2: Customer Table*

Details about each order's shipment are recorded in the Shipping table. It establishes a parent-child relationship between the two tables using ShippingID as the primary key and foreign key references to the CustomerID from the Customer table. It is simple to follow the logistics of each order with the help of this table, which also includes information like the OrderID, OrderDate, ShipDate, and ShipMode. Since a single shipment can cover multiple orders, OrderID is inappropriate for use as a foreign key in the Shipping table. It would be misleading to indicate that each shipment is associated with a single order if OrderID were a foreign key. In the OrderItem table, orders are tracked individually, but ShippingID continues to be the primary key in the Shipping table. By avoiding redundancy and keeping a flexible system that accurately reflects the many-to-one relationship between shipments and orders, this structure enables us to track shipments independently of specific orders.



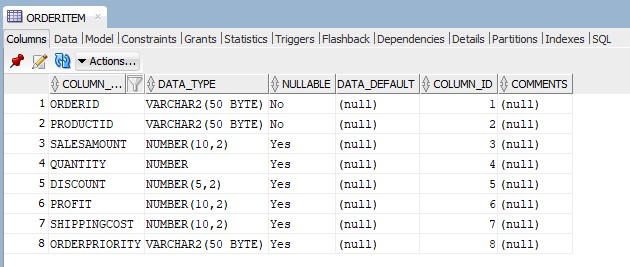
*Figure 3.3: Shipping Table*

The Product table stores data about the products, with ProductID as the primary key. It categorizes products into categories and subcategories through fields like Category and SubCategory, including the ProductName. This table provides a structure for managing product information independently from customer and order data.



*Figure 3.4: Product Table*

Last but not least, every single order's specific products are stored in the OrderItem table. OrderID and ProductID make up its composite primary key, which guarantees the uniqueness of every order and product combination. Important metrics like SalesAmount, Quantity, Discount, Profit, ShippingCost, and OrderPriority are included in this table. Each order item is linked to its corresponding product through a foreign key reference from the OrderItem table to the Product table via ProductID.



*Figure 3.5: OrderItem Table*

## Indexes Suggestion

1. Index on CustomerID in the Shipping table (idx\_shipping\_customerid)

CustomerID column in the Shipping table act as a foreign key from the Customer table that connects each shipping record to a customer. The efficiency of queries that filter or join the Shipping table based on CustomerID can be increased by indexing on CustomerID. This is especially helpful for tasks that include merging shipment records with customer information or searching through all shipments for a particular customer. These operations would need a full table scan, which can be time-consuming, particularly for large datasets, if there was no index.

1. Index on CustomerName in the Customer table (idx\_customer\_name)

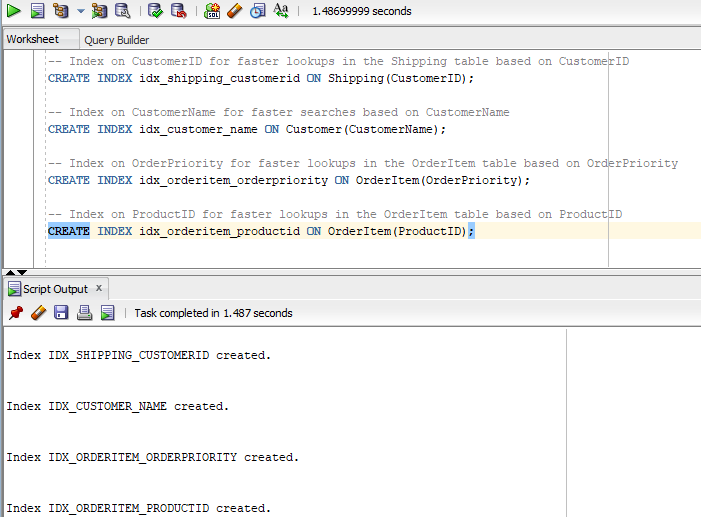
The CustomerName column in the Customer table is often used in queries for searching, filtering, or reporting purposes. Operations that involve looking up customers by name perform better when CustomerName is indexed. This index, as opposed to a full table scan, guarantees that operations such as finding a customer by name or generating reports that filter customers by name are completed more quickly if you regularly need to do so.

1. Index on OrderPriority in the OrderItem table (idx\_orderitem\_orderpriority)

Orders can be filtered or sorted according to their priority level using the OrderPriority column in the OrderItem table. The performance of queries involving filtering or sorting according to order priority is enhanced by creating an index in this column. This index is useful for operations that need to quickly retrieve orders with a certain priority level or for creating reports or analytics that classify orders according to their priority.

1. Index on ProductID in the OrderItem table (idx\_orderitem\_productid)

Each order item in the OrderItem table is associated with a unique product in the Product table via a foreign key found in the ProductID column. Queries that filter order items based on the product or join OrderItem with Product are sped up by indexing ProductID. When you have to look up every order item connected to a specific product or carry out operations involving product details, this is essential for performance.



*Figure 3.6: Indexes Creation*

# CRUD OPERATIONS

## Create

This PL/SQL script allow user to enter new value into customer, shipping, product and orderItem table. The script will start prompt user to enter user’s First Name, Last Name, Segment, City, State, Country, Market, and Region. The figures below show the scripts prompt user to enter the relative values.

A screenshot of a computer error

Description automatically generated A screenshot of a computer error

Description automatically generated

*Figure 4.1: Request User’s First Name Figure 4.2: Request User’s Last Name*

A screenshot of a computer error

Description automatically generated A screenshot of a computer

Description automatically generated

*Figure 4.3: Request Segment Figure 4.4: Request City*

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

*Figure 4.5: Request State Figure 4.6: Request Country*

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

*Figure 4.7: Request Market Figure 4.8: Request Region*

After user successfully enter the values, the script will auto generate the customerID by selecting the first alphabet of user’s first and last name and follow by 5 unique sequence number. The figure below shows the example outputs.

A screenshot of a computer

Description automatically generated

*Figure 4.9: Sample output of Customer Table*

Secondly, the script will prompt users to enter the ship date and mode. The figures below show that the scripts prompt the user to enter the relative values.

A screenshot of a computer error message

Description automatically generated A screenshot of a computer error

Description automatically generated

*Figure 4.10: Request Ship Date Figure 4.11: Request Ship Mode*

After the user successfully enters the values, the script will auto-generate the shippingID using the sequence number. The figure below shows the example outputs.

A close up of a number

Description automatically generated

*Figure 4.12: Sample output of Shipping Table*

Thirdly, the script will prompt user to input the product category, product subcategory, and product name. The figures below show that the scripts prompt the user to enter the relative values.

A screenshot of a computer

Description automatically generated A screenshot of a computer screen

Description automatically generated

*Figure 4.13: Request Product Category Figure 4.14: Request Product Subcategory*

A screenshot of a computer

Description automatically generated

*Figure 4.15: Request Product Name*

After the user successfully enters the values, the script will auto-generate the productID based on the first 3 alphabets of the product category and product subcategory followed by an 8-digit sequence number. The figure below shows the example outputs.

A white background with black text

Description automatically generated

*Figure 4.16: Sample Output of Product Table*

Fourthly, the script will prompt the user to enter the product quantity, discount value, profit, shipping cost, and order priority. The figures below show the scripts prompt the user to enter the relative values.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

*Figure 4.17: Request Product Quantity Figure 4.18: Request Discount Rate*

A screenshot of a computer error

Description automatically generated A screenshot of a computer screen

Description automatically generated

*Figure 4.19: Request Profit Figure 4.20: Request Shipping Cost*

A screenshot of a computer error

Description automatically generated

*Figure 4.21: Request Order Priority*

After user successfully enter the values, the script will auto generate the orderID based on the first 2 alphabets of user’s market and the year of user’s order data and followed by 4 digits sequence number. The figure below shows the example outputs.

A screen shot of a number

Description automatically generated

*Figure 4.22: Sample Output of Order Table*

Finally, the script will display any error when the user input any false or incorrect value into the system. For example, the value for discount should only read 0.10 as 10%, 0.25 as 25% and so on. If the user input value larger than 1, it will prompt error due to the maximum value for discount is 100%. Next example is if user enter invalid shipping mode to the system such as “Air”, it will display error message due to shipping mode does not have “Air” in the script.

A screenshot of a computer

Description automatically generated

*Figure 4.23: Invalid value to Discount*

A black text on a white background

Description automatically generated

*Figure 4.24: Error Handling for Discount*

A screenshot of a computer error

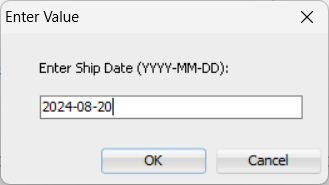
Description automatically generated

*Figure 4.25: Invalid value to Ship Mode*

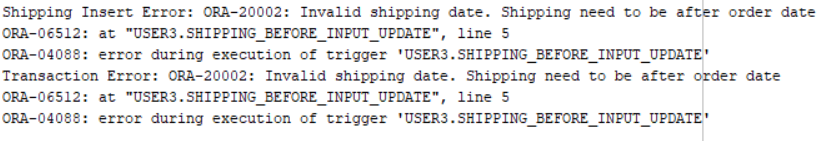
A close-up of a text

Description automatically generated

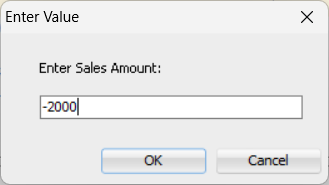
*Figure 4.26: Error Handling for Ship Mode*



*Figure 4.27: Invalid Date to Ship Date*



*Figure 4.25: Error Handling for Ship Date*

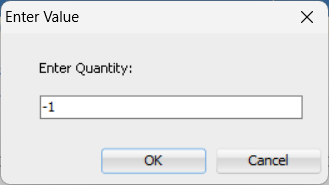


*Figure 4.28: Invalid Value to Sales Amount*

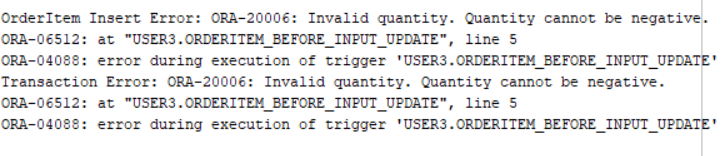
A close-up of a text

Description automatically generated

*Figure 4.29: Error Handling for Sales Amount*



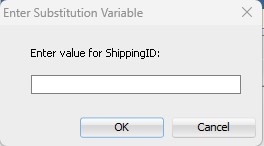
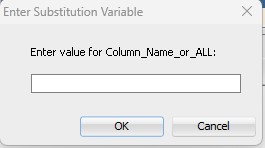
*Figure 4.30: Invalid Value to Quantity*



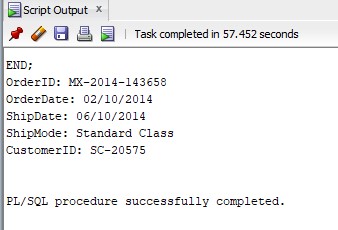
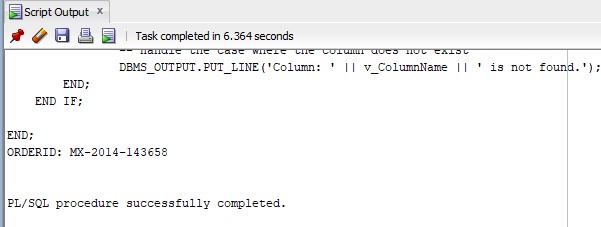
*Figure 4.31: Error Handling for Quantity*

## Read

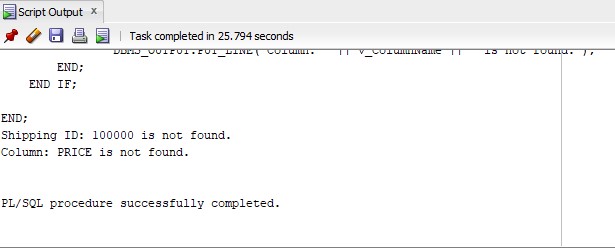
This PL/SQL script enables users to obtain the value of a single column or an entire row by providing the ShippingID. The script begins by verifying the ShippingID entered to make sure it exists in the table. The user has the option to obtain a single column or the complete row of data if the ShippingID is valid. The script uses dynamic SQL to select and display the value of a specified column when that column is requested. The script retrieves and displays all relevant columns for the specified ShippingID, such as OrderID, OrderDate, ShipDate, ShipMode, and CustomerID, if the user chooses to retrieve the entire row. To handle situations in which the ShippingID is invalid or the specified column name is incorrect, the script also includes error handling. When this happens, relevant error messages are shown to users so they are aware of any problems that may have arisen while retrieving data.

*Figure 4.32: Request Shipping ID Figure 4.33: Request Column name*



*Figure 4.34: Retrieve Specific Column Figure 4.35: Retrieve All Column*



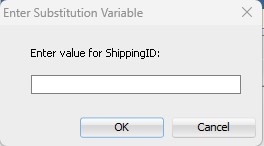
*Figure 4.36: Error Handling for Read*

## Update

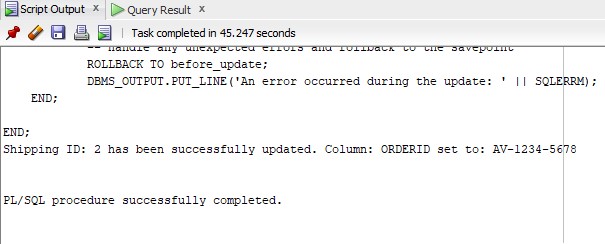
This PL/SQL script is designed to update a specific column in a record within the Shipping table using user input. Its error handling and validation capabilities are robust. Firstly, the script requests that the user enter the ShippingID. It then checks to see if the ShippingID is present in the Shipping table. If the ShippingID cannot be found, the processing is halted, rolling back to a savepoint and displaying the appropriate error message.

Following that, the user is prompted to enter the column name that requires updating. The script checks the Shipping table to see if this column exists using an ALL\_TAB\_COLUMNS view query. The script rolls back to the savepoint and notifies the user if there are any unexpected errors during this validation if the column is missing.

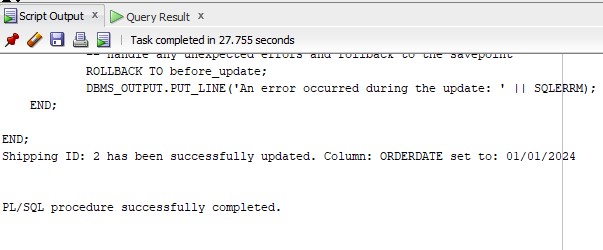
The script asks the user to enter the new value after validating the column name and the ShippingID. It formats the input in date-type columns to the correct format. The script uses dynamic SQL to update the designated column for the record that the ShippingID identifies with the new value. After verifying that the update was successful, it commits the transaction to save the modifications. If there are any problems with the update, the script rolls back to the last savepoint and displays a clear error message. By allowing errors to be handled gracefully and keeping the state constant throughout execution, this method guarantees data integrity.



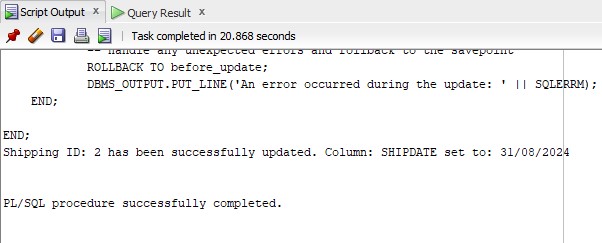
*Figure 4.37: Request Shipping ID*



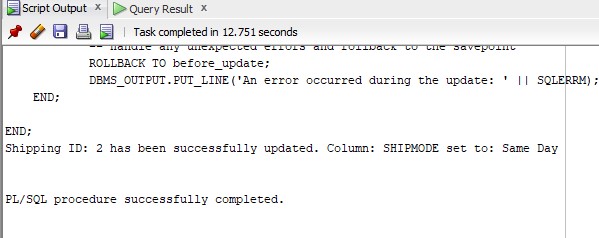
*Figure 4.38: Successfully Updated OrderID*



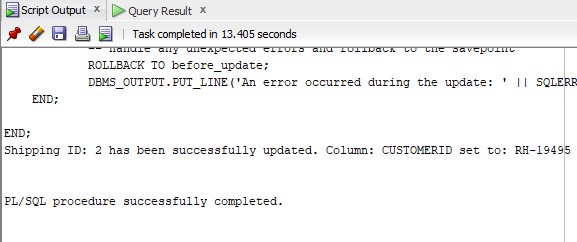
*Figure 4.39: Successfully Updated OrderDate*



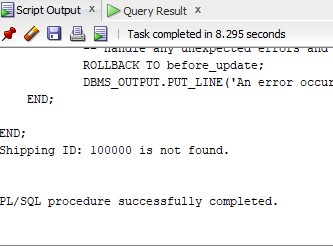
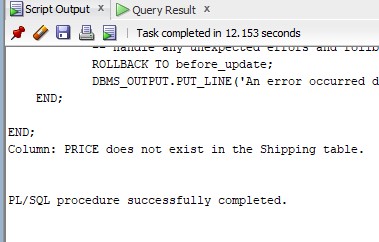
*Figure 4.40: Successfully Updated ShipID*



*Figure 4.41: Successfully Updated ShipMode*



*Figure 4.42: Successfully Updated CustomerID*

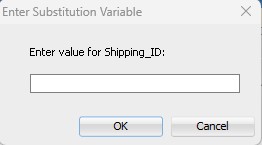
 

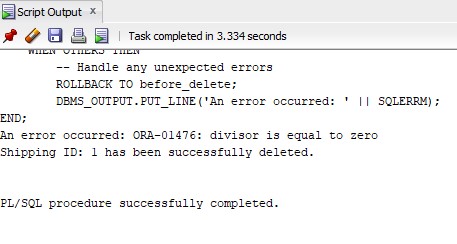
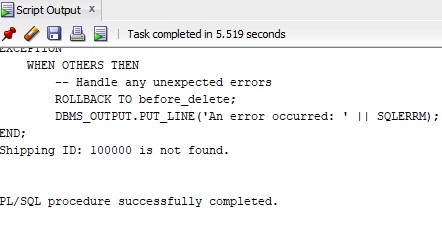
*Figure 4.43: ShippingID Not Found Figure 4.44: Column Not Found*

## Delete

This PL/SQL script is designed to safely delete a record from the Shipping table based on a given ShippingID. Firstly, the script counts the number of records that have the specified ShippingID to determine if it is present in the Shipping table. The script rolls back to a predetermined savepoint and ends, displaying a message stating that the ShippingID was not found, if no records are found (v\_Count = 0).

The script creates a savepoint called before\_delete if the ShippingID is found, allowing any changes to be undone if needed. If there isn't a problem, the script removes the record from the Shipping table that has the given ShippingID. Once the record has been successfully deleted, the transaction is committed, making the change permanent and confirming the deletion with a message.



*Figure 4.45: Request Shipping ID*  

*Figure 4.46: Successfully Deleted Figure 4.47: Error Handling for Delete*

# TRANSACTION MANAGEMENT

In this assignment, we have included strong error-handling capabilities for insert, update, and delete operations by utilizing the SAVEPOINT and ROLLBACK functions. Every operation starts with the setting of the SAVEPOINT, which creates a point at which, in the event of an error, the transaction can be rolled back.

If something goes wrong, the system will quickly perform ROLLBACK to the SAVEPOINT that was previously set and will produce an error message. Statements to insert, update, or delete are performed; the outcome is only committed to permanent storage when it is error-free.

By doing so, any changes made during the transaction up to the SAVEPOINT are undone and consistency in the database is maintained. This method allows recovery from unforeseen problems that may arise when making database modifications, safeguards against incorrect or incomplete updates, and preserves data integrity.

# TRIGGERS

## Shipping\_Audit\_Table

The code snippet below will be regarding After & Before Triggers which contains the purpose of the code. We will be mainly applying the triggers to “shipping” and “orderItem”. As well as displaying the output for Shipping\_Audit.

CREATE TABLE shipping\_audit (

    auditID NUMBER PRIMARY KEY,

    shippingID NUMBER,

    shipDate DATE,

    shipMode VARCHAR2(35),

    action VARCHAR2(10),

    actionTimestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

    customerID VARCHAR2(200),

    orderID VARCHAR2(35),

    orderDate DATE,

);

**Purpose**: This table is used to log actions (INSERT, UPDATE, DELETE) performed on the wx.shipping table.

## Shipping\_Audit\_Sequence

-- Create Sequence for Shipping Audit

CREATE SEQUENCE shipping\_audit\_seq

START WITH 1

INCREMENT BY 1

NOCACHE

NOCYCLE;

**Purpose**: The sequence above is used to auto-generate unique “auditID” values for the “wx.Shipping\_audit” table.

## Shipping\_Audit\_ID\_Trigger

-- Create Trigger for Shipping Audit

CREATE OR REPLACE TRIGGER shipping\_audit\_id\_trigger

BEFORE INSERT ON shipping\_audit

FOR EACH ROW

BEGIN

    :NEW.auditID := shipping\_audit\_seq.NEXTVAL;

END;

**Purpose**: This trigger ensures that every new record inserted into the “wx.shipping\_audit” table gets a unique “auditID” from the sequence “wx.shipping\_audit\_seq”.

## Shipping\_Audit\_Sequence

-- SHIPPING\_AFTER\_DELETE

create or replace TRIGGER shipping\_after\_delete

AFTER INSERT OR UPDATE OR DELETE ON shipping

FOR EACH ROW

BEGIN

    IF DELETING THEN

        INSERT INTO shipping\_audit (auditID, shippingID, orderID, orderDate, shipDate, shipMode, customerID, action)

        VALUES (shipping\_audit\_seq.NEXTVAL, :OLD.shippingID, :OLD.orderID, :OLD.orderDate, :OLD.shipDate, :OLD.shipMode, :OLD.customerID, 'DELETE');

    END IF;

END;

**Purpose:** This is after trigger logs deletions from the wx“.shipping” table into the “wx.shipping\_audit” table.

## Shipping\_After\_Insert\_Update

-- SHIPPING\_AFTER\_INSERT\_UPDATE

create or replace TRIGGER shipping\_after\_Insert\_Update

AFTER INSERT OR UPDATE on shipping

FOR EACH ROW

BEGIN

    --- Will insert into the audit table

    IF INSERTING THEN

        INSERT INTO shipping\_audit (auditID, shippingID, orderID, orderDate, shipDate, shipMode, customerID, action)

        VALUES (shipping\_audit\_seq.NEXTVAL, :NEW.shippingID, :NEW.orderID, :NEW.orderDate, :NEW.shipDate, :NEW.shipMode, :NEW.customerID, 'INSERT');

    ELSIF UPDATING THEN

        INSERT INTO shipping\_audit (auditID, shippingID, orderID, orderDate, shipDate, shipMode, customerID, action)

        VALUES (shipping\_audit\_seq.NEXTVAL, :NEW.shippingID, :NEW.orderID, :NEW.orderDate, :NEW.shipDate, :NEW.shipMode, :NEW.customerID, 'UPDATE');

    END IF;

END;

**Purpose**: This after trigger logs insertions and updates on the “wx.shipping” table into the “wx.shipping\_audit” table.

## Shipping\_Before\_Input\_Update

-- For Shipping Table

create or replace TRIGGER shipping\_before\_Input\_Update

BEFORE INSERT OR UPDATE ON shipping

FOR EACH ROW

BEGIN

    -- Validate ShipDate and ShipMode

    IF INSERTING THEN

        IF :NEW.shipDate IS NULL OR :NEW.shipDate < :NEW.orderDate THEN

            RAISE\_APPLICATION\_ERROR(-20002, 'Invalid shipping date. Shipping needs to be after order date: ' || TO\_CHAR(:NEW.shipDate, 'DD-MON-YYYY HH24:MI:SS') || ' - ' || TO\_CHAR(:NEW.orderDate, 'DD-MON-YYYY HH24:MI:SS'));

        ELSIF :NEW.shipDate < TO\_DATE('1900-01-01', 'YYYY-MM-DD') THEN

            RAISE\_APPLICATION\_ERROR(-20004, 'Invalid shipping date. The date is too far in the past.');

        ELSIF :NEW.shipMode NOT IN ('Same Day', 'Standard Class', 'First Class', 'Second Class') THEN

            RAISE\_APPLICATION\_ERROR(-20003, 'Invalid shipping mode. Allowed values are Same Day, Standard Class, First Class, Second Class.');

        END IF;

    ELSIF UPDATING THEN

        IF :NEW.shipDate IS NULL OR :NEW.shipDate < :NEW.orderDate OR :NEW.shipDate < :OLD.orderDate OR :OLD.shipDate < :NEW.orderDate THEN

            RAISE\_APPLICATION\_ERROR(-20002, 'Invalid shipping date. Shipping need to be after order date');

        ELSIF :NEW.shipDate < TO\_DATE('1900-01-01', 'YYYY-MM-DD') THEN

            RAISE\_APPLICATION\_ERROR(-20004, 'Invalid shipping date. The date is too far in the past.');

        END IF;

        IF :NEW.shipMode IS NULL THEN

            RAISE\_APPLICATION\_ERROR(-20003, 'Invalid shipping mode. Shipping mode cannot be NULL.');

        ELSIF :NEW.shipMode NOT IN ('Same Day', 'Standard Class', 'First Class', 'Second Class') THEN

            RAISE\_APPLICATION\_ERROR(-20003, 'Invalid shipping mode. Allowed values are Same Day, Standard Class, First Class, Second Class.');

        END IF;

    END IF;  -- End of INSERTING/UPDATING block

END;

**Purpose:** This before trigger validates the “shipDate” and “shipMode” fields before inserting or updating records in the “wx.shipping” table, ensuring dates are within valid ranges and shipping modes are within allowed values. As well as Order Date cannot be after Shipping Date.

## OrderItem\_Before\_Input\_Update

-- For OrderItem Table

create or replace TRIGGER orderItem\_before\_Input\_Update

BEFORE INSERT OR UPDATE ON orderItem

FOR EACH ROW

BEGIN

    -- Validate Quantity, SalesAmount, and Discount for Inserts

    IF INSERTING THEN

        IF :NEW.quantity < 0 THEN

            RAISE\_APPLICATION\_ERROR(-20006, 'Invalid quantity. Quantity cannot be negative.');

        ELSIF :NEW.salesAmount < 0 THEN

            RAISE\_APPLICATION\_ERROR(-20007, 'Invalid sales amount. Sales amount cannot be negative.');

        ELSIF :NEW.discount > 1.0 THEN

            RAISE\_APPLICATION\_ERROR(-20008, 'Invalid discount. Discount must be not exceed 100%.');

        END IF;

    -- Validate Quantity, SalesAmount, and Discount for Updates

    ELSIF UPDATING THEN

        IF :NEW.quantity < 0 THEN

            RAISE\_APPLICATION\_ERROR(-20006, 'Invalid quantity. Quantity cannot be negative.');

        ELSIF :NEW.salesAmount < 0 THEN

            RAISE\_APPLICATION\_ERROR(-20007, 'Invalid sales amount. Sales amount cannot be negative.');

        ELSIF :NEW.discount > 1.0 THEN

            RAISE\_APPLICATION\_ERROR(-20008, 'Invalid discount. Discount must be not exceed 100%.');

        END IF;

    END IF;  -- End of INSERTING/UPDATING block

END;

**Purpose:** This before trigger validates the “quantity”, “salesAmount”, and “discount” fields before inserting or updating records in the “wx.orderItem” table, ensuring values are within allowed ranges.

## Shipping\_Before\_Delete

-- SHIPPING\_BEFORE\_DELETE

create or replace TRIGGER shipping\_before\_delete

BEFORE DELETE ON shipping

FOR EACH ROW

DECLARE

    order\_count NUMBER;

BEGIN

    -- change to user admin username

    IF USER != 'wx' THEN

        RAISE\_APPLICATION\_ERROR(-20006, 'Only admin can delete orders.');

    END IF;

END;

**Purpose:** This trigger restricts deletion of records from the “wx.shipping” table to users with the username wx, ensuring that only admins can perform delete operations.

## Output: Shipping\_Audit

A screenshot of a computer screen

Description automatically generated

*Figure 6.1 Shipping\_Audit Output*

# DATA ANALYSIS

In data analysis, we use the PL/SQL queries to do aggregations, filter, sort, and join the database tables to carry out a report for users. There are 5 procedures we made for analyzing the data in database tables which are sales report for monthly or yearly, sales by market report, top 5 sales products in each market, total sales in each shipping mode report, and end of today report.

## Sales Report for Monthly or Yearly

The purpose of this procedure is to show users the total sales with the options for each month or year. This enables users to analyze sales performance over different periods and make informed business decisions.

### Sales Report for Monthly or Yearly

The type of range for the sales report must be entered by the user when the SQL query is launched. For a month-by-month sales report, the user can enter "A" or "Monthly," and for a year-by-year sales report, "B" or "Yearly."

ACCEPT range\_type char PROMPT "Enter Sales Report Range Type (A for Monthly, B for Yearly): ";

DECLARE

    range\_type VARCHAR2(10);

BEGIN

range\_type := UPPER('&range\_type');

range\_sales\_report(range\_type);

END;

/

A screenshot of a computer error message

Description automatically generated

*Figure 7.1: Request Sales Report Range Type*

### Procedure to Generate Report

After the user inputs the type, it will run the *range\_sales\_report* procedure with the type as a parameter. The *range\_type* will be checked with an if else statement to ensure the valid input. If the input is invalid, the invalid message will show back to the user. Otherwise, it will continue to query the selected range type to produce the sales report. To optimize the query, operations like projection, and selection have been done in the joining operation.

create or replace PROCEDURE range\_sales\_report (

    range\_type IN VARCHAR2

) IS

    CURSOR monthly\_cursor IS

    SELECT

        TO\_CHAR(s.ORDERDATE, 'YYYY-MM') AS month,

        SUM(oi.SALESAMOUNT) AS totalsales

    FROM

        SHIPPING s

        JOIN (SELECT o.ORDERID, o.SALESAMOUNT FROM ORDERITEM o) oi ON s.ORDERID = oi.ORDERID

    GROUP BY

        TO\_CHAR(s.ORDERDATE, 'YYYY-MM')

    ORDER BY

        month DESC;

    CURSOR yearly\_cursor IS

    SELECT

        TO\_CHAR(s.ORDERDATE, 'YYYY') AS year,

        SUM(oi.SALESAMOUNT) AS totalsales

    FROM

        SHIPPING s

        JOIN (SELECT o.ORDERID, o.SALESAMOUNT FROM ORDERITEM o) oi ON s.ORDERID = oi.ORDERID

    GROUP BY

        TO\_CHAR(s.ORDERDATE, 'YYYY')

    ORDER BY

        year DESC;

    monthly\_record monthly\_cursor%ROWTYPE;

    yearly\_record yearly\_cursor%ROWTYPE;

BEGIN

    IF range\_type = 'MONTHLY' OR range\_type = 'A' THEN

        OPEN monthly\_cursor;

        DBMS\_OUTPUT.PUT\_LINE('Month' || CHR(9) || 'Total Sales');

        DBMS\_OUTPUT.PUT\_LINE('--------------------------');

        LOOP

            FETCH monthly\_cursor INTO monthly\_record;

            EXIT WHEN monthly\_cursor%NOTFOUND;

            DBMS\_OUTPUT.PUT\_LINE(monthly\_record.month || CHR(9) || 'RM ' || monthly\_record.totalsales);

        END LOOP;

        CLOSE monthly\_cursor;

    ELSIF range\_type = 'YEARLY' OR range\_type = 'B' THEN

        OPEN yearly\_cursor;

        DBMS\_OUTPUT.PUT\_LINE('Year' || CHR(9) || 'Total Sales');

        DBMS\_OUTPUT.PUT\_LINE('--------------------------');

        LOOP

            FETCH yearly\_cursor INTO yearly\_record;

            EXIT WHEN yearly\_cursor%NOTFOUND;

            DBMS\_OUTPUT.PUT\_LINE(yearly\_record.year || CHR(9) || 'RM ' || yearly\_record.totalsales);

        END LOOP;

        CLOSE yearly\_cursor;

    ELSE

        DBMS\_OUTPUT.PUT\_LINE('Invalid range type. Please use "MONTHLY" or "YEARLY".');

    END IF;

EXCEPTION

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

END;

/

### Sample Outputs

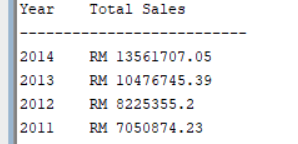
1. Monthly Sales Report

A screenshot of a computer

Description automatically generated

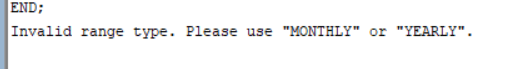
*Figure 7.2: Sample Output for Monthly Sales Report*

1. Yearly Sales Report



*Figure 7.3: Sample Output for Yearly Sales Report*

1. Invalid Input Message



*Figure 7.4 Invalid Range Type*

## Sales by Market Report

The purpose of the *sales\_by\_market\_report* procedure is to show the total sales from each market.

### Execution Way

execute sales\_by\_market\_report();

### Procedure

In this procedure, there will be a cursor to loop all the results from the selection, so that we can log out the result to the user. The query involves three tables which are customer table for the market, orderitem table for sales amount and shipping table that links two tables. The query is optimised by using the projection and selection before joining the table to reduce the cost of the query.

create or replace PROCEDURE sales\_by\_market\_report IS

    CURSOR market\_cursor IS

    SELECT

        c.MARKET,

        SUM(oi.SALESAMOUNT) AS total\_sales

    FROM

        SHIPPING s

        JOIN (SELECT o.SALESAMOUNT, o.ORDERID FROM ORDERITEM o) oi ON s.ORDERID = oi.ORDERID

        JOIN (SELECT c.CUSTOMERID, c.MARKET FROM CUSTOMER c) c ON s.CUSTOMERID = c.CUSTOMERID

    GROUP BY

        c.MARKET

    ORDER BY

        total\_sales DESC;

    market\_record market\_cursor%ROWTYPE;

BEGIN

    OPEN market\_cursor;

    DBMS\_OUTPUT.PUT\_LINE(RPAD('MARKET', 20) || 'Total Sales');

    DBMS\_OUTPUT.PUT\_LINE('------------------------------------------');

    LOOP

        FETCH market\_cursor INTO market\_record;

        EXIT WHEN market\_cursor%NOTFOUND;

        DBMS\_OUTPUT.PUT\_LINE(RPAD(market\_record.MARKET, 20) || 'RM ' || market\_record.total\_sales);

    END LOOP;

    CLOSE market\_cursor;

EXCEPTION

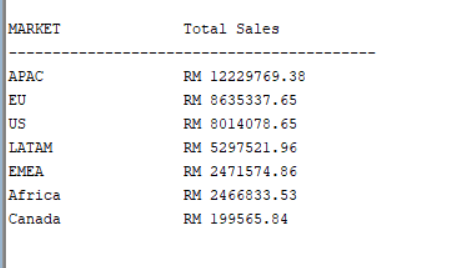
    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

END;

/

### Sample Outputs



*Figure 7.5: Sample Output for Market*

## Top 5 Sales Products in Each Market

The purpose is for users to get the top 5 highest sales in each market or across all the markets.

### User Input Interface

ACCEPT market char DEFAULT "ALL" PROMPT "Enter Market to check TOP 5 Products (All(DEFAULT), APAC, Africa, Canada, EMEA, EU, LATAM, US): ";

DECLARE

    market VARCHAR2(10);

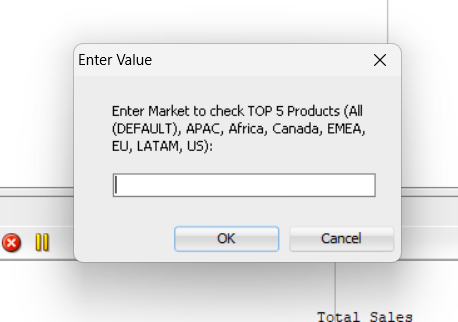
BEGIN

market := UPPER('&market');

TOP\_5\_PRODUCTS\_BY\_MARKET(market);

END;

/



*Figure 7.6: Request Market*

### Procedure

The input will be validated by the procedure to ensure it is a valid market or “ALL”. If the input value is invalid it will prompt a message to alert the user. Otherwise, the query will be run based on the market. For a specific market, the query will involve a customer table, product table, shipping table, and orderitem table. For checking across all markets, it will not involve a customer table to get the marketplace. The result will be ordered by the total sales quantity of product descending and get the first five products.

create or replace PROCEDURE top\_5\_products\_by\_market (p\_market VARCHAR2 DEFAULT 'ALL') IS

    market\_count INTEGER;

    -- Cursor to check if the market exists

    CURSOR market\_check\_cursor IS

    SELECT COUNT(\*)

    FROM CUSTOMER c

    WHERE UPPER(c.MARKET) = p\_market;

    -- Cursor to retrieve the top 5 products for a specific market

    CURSOR product\_cursor (market\_name VARCHAR2) IS

    SELECT \*

    FROM (

        SELECT

            p.PRODUCTID,

            p.PRODUCTNAME,

            SUM(oi.quantity) AS total\_sales

        FROM

            ORDERITEM oi

            JOIN (SELECT p.productID,p.PRODUCTNAME from PRODUCT p) p ON oi.PRODUCTID = p.PRODUCTID

            JOIN (SELECT s.orderID, s.customerID from SHIPPING s) s ON oi.ORDERID = s.ORDERID

            JOIN (SELECT c.customerID, c.market FROM CUSTOMER c) c ON s.CUSTOMERID = c.CUSTOMERID

        WHERE

            UPPER(c.MARKET) = UPPER(market\_name)

        GROUP BY

            p.PRODUCTID, p.PRODUCTNAME

        ORDER BY

            total\_sales DESC

    )

    WHERE ROWNUM <= 5;

    -- Cursor to retrieve the top 5 products accross all markets

    CURSOR product\_cursor\_all IS

        SELECT \* FROM

        (

        SELECT

            p.productid,

            p.productname,

            SUM(oi.quantity) AS total\_sales

        FROM

            orderitem oi

        JOIN (SELECT p.productid, p.productname FROM product p) p ON oi.productid = p.productid

        JOIN (SELECT s.orderid FROM shipping s) s ON oi.orderid = s.orderid

        GROUP BY

            p.productid, p.productname

        ORDER BY

            total\_sales DESC

        )

    WHERE ROWNUM <= 5;

    all\_product\_record product\_cursor\_all%ROWTYPE;

    product\_record product\_cursor%ROWTYPE;

BEGIN

    IF p\_market = 'ALL' THEN

            DBMS\_OUTPUT.PUT\_LINE('Market: ALL MARKET');

            DBMS\_OUTPUT.PUT\_LINE(RPAD('Product ID',18) || RPAD('Product Name',60) || 'Total Sales');

            DBMS\_OUTPUT.PUT\_LINE('-----------------------------------------------------------------------------------');

            OPEN product\_cursor\_all;

            LOOP

                FETCH product\_cursor\_all INTO all\_product\_record;

                EXIT WHEN product\_cursor\_all%NOTFOUND;

                DBMS\_OUTPUT.PUT\_LINE(RPAD(all\_product\_record.PRODUCTID,18) || RPAD(all\_product\_record.PRODUCTNAME,60) || all\_product\_record.total\_sales);

            END LOOP;

            CLOSE product\_cursor\_all;

    ELSE

        -- Check if the specified market exists

        OPEN market\_check\_cursor;

        FETCH market\_check\_cursor INTO market\_count;

        CLOSE market\_check\_cursor;

        IF market\_count > 0 THEN

            -- Process the specific market

            DBMS\_OUTPUT.PUT\_LINE('Market: ' || p\_market);

            DBMS\_OUTPUT.PUT\_LINE(RPAD('Product ID',18) || RPAD('Product Name',60) || 'Total Sales');

            DBMS\_OUTPUT.PUT\_LINE('-----------------------------------------------------------------------------------');

            OPEN product\_cursor(p\_market);

            LOOP

                FETCH product\_cursor INTO product\_record;

                EXIT WHEN product\_cursor%NOTFOUND;

                DBMS\_OUTPUT.PUT\_LINE(RPAD(product\_record.PRODUCTID,18) || RPAD(product\_record.PRODUCTNAME,60) || product\_record.total\_sales);

            END LOOP;

            CLOSE product\_cursor;

            DBMS\_OUTPUT.PUT\_LINE('');

        ELSE

            -- Market does not exist

            DBMS\_OUTPUT.PUT\_LINE('Market "' || p\_market || '" does not exist.');

        END IF;

    END IF;

EXCEPTION

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

END;

/

### Sample Outputs

1. All Market

A white background with black text

Description automatically generated

*Figure 7.7: Sample Output for All Market*

1. Specific Market

A close-up of a computer code

Description automatically generated

*Figure 7.8: Sample Output for Specific Market*

1. Invalid Input

A screenshot of a computer screen

Description automatically generated 

*Figure 7.9: Invalid Value for Market*

## Total Sales in Each Shipping Mode Report

This report will show users the total orders and profit that earn in each Shipping Mode.

### Way to Execute

execute shipping\_mode\_report();

### Procedure

The `shipping\_mode\_report` procedure, it will generate a report on shipping modes by concluding the amount of orders and the profit for each mode. It selects and aggregates data from the orderitem and shipping tables. After joining the tables, the data will be sorted descending by the profit gained by each shipping mode. Besides that, the procedure also counts the amount of the order for each shipping mode.

create or replace PROCEDURE shipping\_mode\_report IS

    CURSOR quantity\_cursor IS

    SELECT

        s.SHIPMODE,

        COUNT(DISTINCT oi.ORDERID) AS order\_count,

        SUM(oi.PROFIT) AS total\_profit

    FROM

        ORDERITEM oi

        JOIN (SELECT SHIPPING.SHIPMODE, SHIPPING.ORDERID FROM SHIPPING) s ON oi.ORDERID = s.ORDERID

    GROUP BY

        s.SHIPMODE

    ORDER BY

        total\_profit DESC;

    quantity\_record quantity\_cursor%ROWTYPE;

BEGIN

    OPEN quantity\_cursor;

    DBMS\_OUTPUT.PUT\_LINE(RPAD('Shipping Mode',20) || RPAD('Total Order',16) || 'Total Profit');

    DBMS\_OUTPUT.PUT\_LINE('------------------------------------------------------');

    LOOP

        FETCH quantity\_cursor INTO quantity\_record;

        EXIT WHEN quantity\_cursor%NOTFOUND;

        DBMS\_OUTPUT.PUT\_LINE(RPAD(quantity\_record.SHIPMODE, 20) || RPAD(quantity\_record.order\_count,16) || 'RM ' || quantity\_record.total\_profit);

    END LOOP;

    CLOSE quantity\_cursor;

EXCEPTION

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

END;

/

### Sample Outputs

A number on a white background

Description automatically generated

*Figure 7.10: Sample Output for Shipping Mode*

## End of Report

This report can show the analysis report by the end of day, month or year. It has included the amount of order, total of sales, profit and the total order for each shipping mode.

### User Input Interface

It prompts the user to input a date range type with the options which are year, month and day. After that, user is required to enter a specific date in the appropriate date format. The inputs then passed to the *end\_of\_report* procedure.

ACCEPT i\_type CHAR PROMPT 'Enter Range Type (YEAR, MONTH, DAY): ';

ACCEPT i\_date CHAR PROMPT 'Enter Date with Format (YEAR = YYYY, MONTH = YYYY-MM, DAY = YYYY-MM-DD): ';

DECLARE

    v\_type VARCHAR2(10);

    v\_date VARCHAR2(15);

BEGIN

    v\_type := UPPER('&i\_type');

    v\_date := '&i\_date';

    end\_of\_report(v\_type, v\_date);

END;

/

A screenshot of a computer error message

Description automatically generated A screenshot of a computer error

Description automatically generated

*Figure 7.11: Request Range Type Figure 7.12: Request Date*

### Procedure

*The end\_of\_report* procedure generates a report based on a specified date range (YEAR, MONTH, or DAY). It first validates the input date type and format by using the REGEXP\_LIKE (Oracle Live SQL - Script: REGEXP\_LIKE- Examples, n.d.). Then, it calculates and outputs the total order count, sales amount, and profit for the given date. The procedure also includes different shipping modes, counting the number of orders for each mode within the specified date range and displaying the results in descending order by count. The procedure handles exceptions, including cases where no data is found, or other errors occur.

CREATE OR REPLACE PROCEDURE end\_of\_report (

    p\_date\_type IN VARCHAR2,

    p\_date\_value IN VARCHAR2

) IS

    v\_order\_count   NUMBER;

    v\_total\_sales   NUMBER;

    v\_total\_profit  NUMBER;

    v\_mode\_count    NUMBER;

BEGIN

    -- Validate input date type and format

    IF p\_date\_type NOT IN ('MONTH', 'DAY', 'YEAR') THEN

        RAISE\_APPLICATION\_ERROR(-20001, 'Invalid date type. Use "MONTH", "DAY", or "YEAR".');

    END IF;

    IF p\_date\_type = 'MONTH' AND (NOT regexp\_like(p\_date\_value, '^[0-9]{4}-[0-9]{2}$')) THEN

       RAISE\_APPLICATION\_ERROR(-20001, 'Invalid date value. Use YYYY-MM for MONTH.');

    END IF;

    IF p\_date\_type = 'DAY' AND (NOT regexp\_like(p\_date\_value, '^[0-9]{4}-[0-9]{2}-[0-9]{2}$')) THEN

       RAISE\_APPLICATION\_ERROR(-20001, 'Invalid date value. Use YYYY-MM-DD for DAY.');

    END IF;

    IF p\_date\_type = 'YEAR' AND (NOT regexp\_like(p\_date\_value, '^[0-9]{4}$')) THEN

       RAISE\_APPLICATION\_ERROR(-20001, 'Invalid date value. Use YYYY for YEAR.');

    END IF;

    -- Fetch the total order count, sales, and profit

    SELECT

        COUNT(DISTINCT oi.ORDERID),

        SUM(oi.SALESAMOUNT),

        SUM(oi.PROFIT)

    INTO

        v\_order\_count,

        v\_total\_sales,

        v\_total\_profit

    FROM

        ORDERITEM oi

    JOIN SHIPPING s ON oi.ORDERID = s.ORDERID

    WHERE

        (p\_date\_type = 'MONTH' AND TO\_CHAR(s.ORDERDATE, 'YYYY-MM') = p\_date\_value)

        OR

        (p\_date\_type = 'DAY' AND TO\_CHAR(s.ORDERDATE, 'YYYY-MM-DD') = p\_date\_value)

        OR

        (p\_date\_type = 'YEAR' AND TO\_CHAR(s.ORDERDATE, 'YYYY') = p\_date\_value);

    IF v\_order\_count = 0 THEN

       DBMS\_OUTPUT.PUT\_LINE('No data found for the specified date.');

        RETURN;

    END IF;

    -- Output the totals

    DBMS\_OUTPUT.PUT\_LINE('Report for end of ' || LOWER(p\_date\_type) || ' ' || p\_date\_value);

    DBMS\_OUTPUT.PUT\_LINE('');

    DBMS\_OUTPUT.PUT\_LINE('Order Count: ' || v\_order\_count);

    DBMS\_OUTPUT.PUT\_LINE('Total Sales: RM ' || v\_total\_sales);

    DBMS\_OUTPUT.PUT\_LINE('Total Profit: RM ' || v\_total\_profit);

    DBMS\_OUTPUT.PUT\_LINE('');  -- New line for formatting

    -- Cursor to iterate through the ship modes and their counts

    DBMS\_OUTPUT.PUT\_LINE( RPAD('Shipping Mode', 18) || 'Order Count');

    FOR r IN (

        SELECT

            s.SHIPMODE,

            COUNT(\*) AS mode\_count

        FROM

            SHIPPING s

        WHERE

            (p\_date\_type = 'MONTH' AND TO\_CHAR(s.ORDERDATE, 'YYYY-MM') = p\_date\_value)

            OR

            (p\_date\_type = 'DAY' AND TO\_CHAR(s.ORDERDATE, 'YYYY-MM-DD') = p\_date\_value)

            OR

            (p\_date\_type = 'YEAR' AND TO\_CHAR(s.ORDERDATE, 'YYYY') = p\_date\_value)

        GROUP BY

            s.SHIPMODE

        ORDER BY

            mode\_count DESC

    ) LOOP

        DBMS\_OUTPUT.PUT\_LINE( RPAD(r.SHIPMODE, 18) || r.mode\_count);

    END LOOP;

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        DBMS\_OUTPUT.PUT\_LINE('No data found for the specified date.');

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

END;

/

### Sample Outputs

1. EOY Report

A screenshot of a computer screen

Description automatically generated

*Figure 7.13: EOY Report*

1. EOM Report

A screenshot of a computer screen

Description automatically generated

*Figure 7.14: EOM Report*

1. EOD Report

A screenshot of a computer

Description automatically generated

*Figure 7.15: EOD Report*

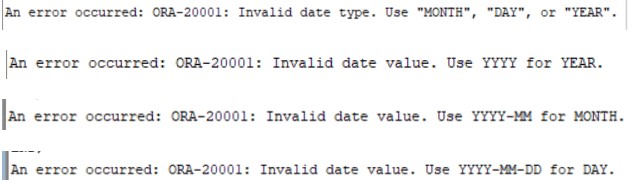
1. No Data Handling

A screenshot of a computer code

Description automatically generated

*Figure 7.16: Empty Data*

1. Invalid Input Handling



*Figure 7.17: Invalid Input*

# REFERENCES

* Global Super Store Dataset. (n.d.). Www.kaggle.com. https://www.kaggle.com/datasets/apoorvaappz/global-super-store-dataset
* Kannan, P., Morin, L., Raphaely, D., Ashdown, L., Baker, D., Carver, D., Chaliha, M., Cheng, B., Day, R., Fogel, S., Llewellyn, B., Lane, P., McDermid, D., Morales, T., Murphy, A., Murray, C., Pelski, S., Rich, K., Romero, A., . . . Chaudhry, A. (2024, July 29). UTL\_FILE. Oracle Help Center. https://docs.oracle.com/en/database/oracle/oracle-database/19/arpls/UTL\_FILE.html
* 12 using regular expressions with Oracle database. (n.d.). https://docs.oracle.com/cd/B12037\_01/appdev.101/b10795/adfns\_re.htm
* Oracle Live SQL - Script: REGEXP\_LIKE- examples. (n.d.). https://livesql.oracle.com/apex/livesql/file/content\_BCIYBOLU3HSIRATHWNFKWHXIM.html

‌