L5 Data Engineer

Module 2 – Databases and Data Lakes – EPA Preparation (Applied Exercise)

This is a long exercise that overlaps with Topic 2 (Schemas and Integration) and Topic 3 (Advanced SQL) so the learners have multiple weeks to complete this.

This Applied Exercise is expertly crafted to prepare you for your endpoint assessment, particularly the professional discussion as facilitated by BCS (assessment method 2 – see EPA guidance in the programme handbook for more information). The professional discussion EPA session will be a formal two-way conversation where you will demonstrate your proficiency across specified criteria. This exercise addresses the "Data Quality and Performance" and "Problem Solving" themes of the professional discussion.

In this exercise, you will be engaging in the creation, population, and querying of a star schema using advanced SQL techniques. This involves designing database tables, populating them with sample data, and utilising SQL joins to extract meaningful insights from the interconnected data. This hands-on approach is aimed at strengthening your practical SQL skills and your understanding of relational database management systems.

Objectives of the Applied Exercise:

Data Quality and Performance: *By actively participating in the building and manipulation of a star schema, you will gain practical insights into the intricacies of* ***data warehousing*** *principles (K15) and the critical role they play in maintaining high* ***data quality and system performance****. This exercise also aligns with K24, focusing on the evaluation of data management* ***prototypes*** *and their transition into* ***production*** *environments.*

Problem Solving*: Throughout this exercise, you will face challenges related to* ***data integration*** *(K17) and the management of a data solution's lifecycle (K25), from initial scoping and prototyping through to development, production, and continuous improvement. These challenges will test and enhance your* ***problem-solving skills, helping you to provide innovative solutions*** *(B5) and effectively manage and manipulate data.*

As you progress through this exercise, you will create a piece of so-called "portfolio work" that serves as tangible evidence of your skills and learning reflections. **Document each step of your exercise by capturing screenshots** of your SQL code and the results of your queries. This will demonstrate your capability in handling various data operations—from designing schemas to implementing complex SQL queries for data integrity checks (S7, S9).

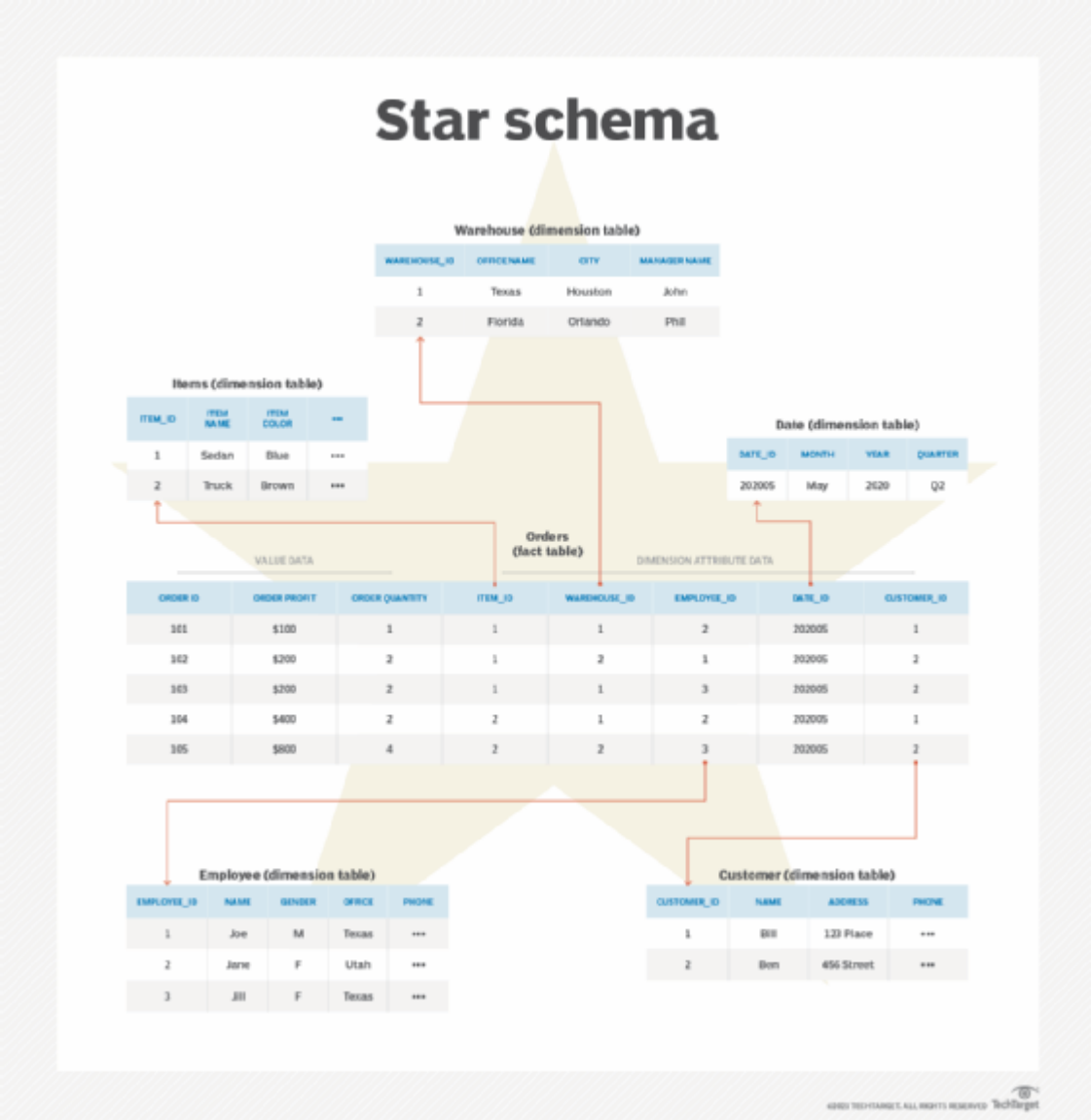
> *Reflective Learning: Your portfolio should also include a personal reflection discussing the lessons learned throughout this exercise. Emphasize how these experiences have solidified your understanding of the importance of data quality, performance, and effective problem-*

solving within professional environments. Showcase how this exercise has prepared you to take proactive steps and assume accountability in your role (B1). Reflect on your readiness to meet deadlines, adapt to changing priorities, and your contribution to sustainability and environmental strategies (S27). Feel free to adjust aspects of this exercise (such as data types or column names) to better reflect the nature of the data at your workplace.

By immersing yourself in this Applied Exercise, you will be thoroughly prepared to discuss your technical competencies and their practical applications during the professional discussion

Let's get started!

Consider the Star Schema from your Webinar:



You will be given SQL instructions to create a fact table named orders and the associated dimension tables for a star schema database structure. The schema includes primary and foreign key constraints that establish the relationships necessary for a star schema.

Before you do any steps with SQL, read the following re-cap of SQL Components and document any additional learning that you may have done (such as googling, reviewing e-learning or your notes).

> *Primary Key: A primary key is a unique identifier for a record in a table. It cannot accept null values and must contain unique values. A table can have only one primary key, which can consist of single or multiple fields. In the context of this schema, each dimension table has a primary key on their ID fields (e.g., item_id, warehouse_id).*
>
> *Foreign Key: A foreign key is a field (or collection of fields) in one table that uniquely identifies a row of another table. In this schema, the foreign keys in the orders table reference the primary keys of the dimension tables. These keys help maintain referential integrity by ensuring that the key field only contains values that are present in the referenced primary key field of the linked table.*
>
> *This setup of primary and foreign keys is typical for a star schema in a data warehousing environment, where the fact table (orders) sits at the center of the schema, and the dimension tables (items, warehouse, date, employee, customer) are linked to it through foreign keys. This structure supports efficient data analysis and querying within an OLAP (Online Analytical Processing) system.*

The SQL you will be given will introduce something known as **constraints.**

> These constraints enhance the robustness and reliability of the database schema by ensuring that data stored in the database adheres to specified rules and standards, thereby preventing errors during data entry and retrieval:
>
> Item Color Constraint: The item_color field in the items table needs a constraint that restricts the values to 'blue', 'brown', 'yellow', or 'red'.
>
> Order ID Constraint: The order_id field in the orders table is constrained to be unique and non-negative. This is achieved by setting it as a primary key and adding a check constraint to ensure the value is greater than or equal to zero.
>
> Gender Constraint: The gender field in the employee table now accepts only three values: 'male', 'female', or 'other'. This is enforced using a check constraint.
>
> Furthermore, the SERIAL constraint. In PostgreSQL, you can use the SERIAL data type for automatically incrementing integer fields. For a non-negative unique primary key, the SERIAL type starts at 1 and increments automatically. This approach is common in PostgreSQL for defining primary keys that automatically generate unique values upon new record insertion. This approach simplifies the insertion process, as there is no need to manually input a value for the primary key—PostgreSQL handles it automatically, ensuring data integrity and reducing the risk of duplication or entry errors.
>
> **To further refine the database schema, you will add constraints to ensure that all fields in the customer, employee, and warehouse tables are non-empty. You'll also create a view to filter orders with blue items and write a simple stored procedure that performs a data validation check upon adding a new record.**

Non-Empty Constraints: By adding NOT NULL to all fields in the customer, employee, and warehouse tables, this ensures that all entries must have values, thus preventing any incomplete records.

You'll also witness adding a view, a stored procedure and a trigger.

View for Orders with Blue Items: The blue_item_orders view filters orders that involve items with a blue color. It joins the orders table with the items table to check the item color, providing an easy way to query relevant data.

Stored Procedure and Trigger: The check_order_quantity() function will be a simple data validation check that ensures the quantity of an order is not unreasonably high (set arbitrarily here as 1000 units). The trigger order_quantity_check calls this function before inserting a new record into the orders table, providing an automated way to prevent data entry errors.

These additions make the database more robust and user-friendly by ensuring data integrity and simplifying data queries based on specific criteria.

Special considerations

Furthermore, Renaming the date table to date_quarter in your PostgreSQL schema is a prudent move because "date" is a reserved keyword in SQL. Reserved keywords have specific meanings or functions in SQL languages and are part of the SQL standard. Using a reserved keyword as a table or column name can lead to conflicts during query execution, potentially causing errors unless properly quoted. By renaming the table to date_quarter, you avoid potential syntax issues and improve the readability of your SQL scripts. The new name also provides a clearer description of what the table contains, specifically focusing on quarterly data time frames.

Reserved Keyword: In SQL, "date" is commonly used to define data types that store dates, and as such, it is a reserved keyword. This means the SQL engine could misinterpret the table name "date" as a data type, leading to confusion or errors during query processing.

Future Proofing: Using descriptive and non-reserved names helps ensure that your database schema remains compatible with future updates to the SQL standard and different SQL database systems, which might adjust or expand their list of reserved keywords.

You are now ready to execute the following code in your chosen SQL environment!

```
CREATE TABLE items (
    item_id SERIAL PRIMARY KEY,
    item_name VARCHAR(255) NOT NULL,
    item_color VARCHAR(50) NOT NULL CHECK (item_color IN ('blue', 'brown', 'yellow', 'red'))
);

-- Create dimension table for Warehouse
CREATE TABLE warehouse (
    warehouse_id SERIAL PRIMARY KEY,
    office_name VARCHAR(255) NOT NULL,
    city VARCHAR(100) NOT NULL,
    manager_name VARCHAR(255) NOT NULL
);

-- Create dimension table for Date
CREATE TABLE date_quarter (
    date_id SERIAL PRIMARY KEY,
    month INT NOT NULL,
    year INT NOT NULL,
    quarter INT NOT NULL
);

-- Create dimension table for Employee
CREATE TABLE employee (
    employee_id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    gender VARCHAR(10) NOT NULL CHECK (gender IN ('male', 'female', 'other')),
    office VARCHAR(100) NOT NULL,
    phone VARCHAR(20) NOT NULL
);

-- Create dimension table for Customer
CREATE TABLE customer (
    customer_id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address VARCHAR(255) NOT NULL,
    phone VARCHAR(20) NOT NULL
```

```
);

-- Create the fact table for Orders
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    order_profit DECIMAL(10, 2) NOT NULL,
    order_quantity INT NOT NULL,
    item_id INT NOT NULL,
    warehouse_id INT NOT NULL,
    employee_id INT NOT NULL,
    date_id INT NOT NULL,
    customer_id INT NOT NULL,
    FOREIGN KEY (item_id) REFERENCES items(item_id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouse(warehouse_id),
    FOREIGN KEY (employee_id) REFERENCES employee(employee_id),
    FOREIGN KEY (date_id) REFERENCES date_quarter(date_id),  -- foreign key avoids reserved
keyword
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);

-- Create a view for orders with blue items
CREATE VIEW blue_item_orders AS
SELECT o.*
FROM orders o
JOIN items i ON o.item_id = i.item_id
WHERE i.item_color = 'blue';

-- Stored procedure to check order quantity is sensible (e.g., not excessively high)
CREATE OR REPLACE FUNCTION check_order_quantity()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.order_quantity > 1000 THEN
        RAISE EXCEPTION 'Order quantity too high: %', NEW.order_quantity;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger to invoke the check on order quantity before inserting a new order
CREATE TRIGGER order_quantity_check
BEFORE INSERT ON orders
```

```
FOR EACH ROW

EXECUTE FUNCTION check_order_quantity();
```

Now populate your database with some sample data

Include SQL INSERT statements along with comments explaining each step. This will provide a narrative that helps others understand the importance and impact of each record you are entering.

```
-- Insert sample data into the 'items' table

INSERT INTO items (item_name, item_color)

VALUES ('Laptop', 'blue'), ('Desktop', 'brown'), ('Keyboard', 'yellow'), ('Mouse', 'red');

-- These records represent various inventory items that a company might sell, with specified
colors


-- Insert sample data into the 'warehouse' table

INSERT INTO warehouse (office_name, city, manager_name)

VALUES ('Main Warehouse', 'New York', 'John Doe'), ('Secondary Warehouse', 'Los Angeles',
'Jane Smith');

-- These records reflect different storage locations for the items and include the names of
managers responsible for each warehouse


-- Insert sample data into the 'date_quarter' table

INSERT INTO date_quarter (month, year, quarter)

VALUES (1, 2022, 1), (4, 2022, 2), (7, 2022, 3), (10, 2022, 4);

-- Each record corresponds to a specific quarter of the year 2022, which will be referenced in
orders to track seasonal sales


-- Insert sample data into the 'employee' table

INSERT INTO employee (name, gender, office, phone)

VALUES ('Alice Johnson', 'female', 'Headquarters', '1234567890'), ('Bob Brown', 'male',
'Remote', '0987654321');

-- These records represent employees of the company who might be handling the orders,
including their contact information


-- Insert sample data into the 'customer' table

INSERT INTO customer (name, address, phone)

VALUES ('Charlie Corp', '123 Elm St, Somewhere', '555-0011'), ('Delta Inc', '456 Oak Rd,
Anywhere', '555-0022');

-- Customers who purchase items from the company, with contact details provided for billing
and communication


-- Insert sample data into the 'orders' table
```

```
INSERT INTO orders (order_profit, order_quantity, item_id, warehouse_id, employee_id, date_id,
customer_id)

VALUES (500.00, 10, 1, 1, 1, 1, 1), (300.00, 5, 2, 2, 2, 2, 2);

-- Orders placed by customers, linking all previously entered dimensions. Includes profit and
quantity to demonstrate financial and logistical impacts of transactions


-- The above steps comprehensively demonstrate how a simple e-commerce system might be
modelled in a relational database. Each table holds specific aspects of the business process,
and together they form a fully relational schema.
```

Explanation of View and Stored Procedure Usage

View for Orders with Blue Items:

This view simplifies querying orders that include blue items, which could be useful for specific marketing or inventory analysis tasks.

Example query to use the view:

```
SELECT * FROM blue_item_orders;
```

Stored Procedure and Trigger:

The stored procedure and trigger ensure that no order is entered with a quantity exceeding 1,000, which might indicate a data entry error or an unusual bulk order needing special handling.

This is an example of implementing business rules directly within the database to maintain data integrity and enforce company policies automatically.

Exploring SQL Joins

SQL joins are crucial for querying data from two or more tables in a relational database. They allow you to combine rows from these tables based on a related column, providing a powerful tool for data analysis and reporting.

Task 1: INNER JOIN

Goal: Retrieve all orders along with the item details but only where the item colour is 'blue'.

SQL Query

```
SELECT o.order_id, o.order_quantity, i.item_name, i.item_color
FROM orders o
INNER JOIN items i ON o.item_id = i.item_id
WHERE i.item_color = 'blue';
```

This query demonstrates an INNER JOIN, which returns rows when there is at least one match in both tables. It's used here to link orders with their respective items, filtering for only those items that are blue.

Task 2: LEFT JOIN

Goal: List all employees and any orders they may have processed, including employees who have not processed any orders.

SQL Query

```
SELECT e.name, e.office, o.order_id
FROM employee e
LEFT JOIN orders o ON e.employee_id = o.employee_id;
```

The LEFT JOIN ensures that all employees are listed, regardless of whether they have processed any orders. This is crucial for full staff audits or analyses, showing unassigned employees as well.

Task 3: RIGHT JOIN

Display all customers and any orders they have placed, including all placed orders even if the customer details are not recorded.

```
SELECT c.name, c.phone, o.order_id
FROM customer c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;
```

A RIGHT JOIN is useful when you want to ensure all entries in the joined table (orders in this case) are shown—even if there's no corresponding entry in the base table (customers).

Finally, discuss Normalisation

To consolidate your comprehension of how normalisation affects data integrity and system performance, analyse the provided schema and identify potential areas of redundancy.

Suggest normalisation improvements to the current schema design.

Consider how the orders table might duplicate data, such as customer information or item details. By normalising your database, you ensure that each piece of data is stored only once, reducing the storage space and increasing the efficiency of queries. For example, storing customer information in a separate table and referencing it via a foreign key in the orders table helps maintain data consistency and integrity.

By completing these tasks, you will have a practical understanding of how SQL joins work and why normalisation is critical in database design. Remember, a well-normalised database not only enhances performance but also simplifies maintenance and scalability. Always aim to balance normalisation with practical requirements to achieve optimal database performance and utility.

Save your portfolio piece with screenshots and reflections as a word document and upload it into your learning journal.