DVD RENTAL

DATA REPORTS

# Data Analysis with SQL using ETL

**(Extraction, Transformation, and Loading)**

Celeste M Catala

Western Govenor's University

D191: Advanced Data Management

Mindy Crowder

9/2/2024

## Section A: Introduction

Businesses thrive when they deep dive into their data and review what is going well and generating revenue and what is not beneficial for the company, in which case these processes should be either reduced or suspended. As with this DVD rental company, my goal as a data analyst hired by this business is to analyze the data and see what is helping to generate the most revenue and how we can use this information to not just maintain our current sales but also look for opportunities to further increase revenue. It is important when analyzing data to start with a specific question in mind and then search through the data and find specific answers to those specific questions. Whereas just reviewing the data as it is can be tedious and inconclusive for business analyst/reporting. Narrowing the results can be done by using ETL (extraction, transformation, and loading) which is a process used to move and prepare data from various sources to a data warehouse where the data is accessible for analysis, reporting, and decision-making.

The business question that I will be exploring and finding answers to is "Which customers demonstrate the highest loyalty to the DVD rental store, as indicated by their frequency of visits based upon their rental history?" The two tables I will create to answer this question are a detailed and summary table. The detailed table is granular data pertaining to an individual customer and their rental transactions. With the detailed table we can learn about a customer's activity such as how many times they appear on the table based on their rental history that is linked to their customer_id. The summary table aggregates data from the detailed table, summarizing customer activity. With the structure of the summary table including the customer's full name, email, and customer_count (how many times they appear in the detailed table) we can easily identify which customers have rented the most DVDs because the table is sorted in descending order according to their customer_count. With the summary table, we can easily identify frequent renters and implement a reward system that rewards them monthly. This combination of detailed and summarized data will allow the DVD rental store to make data-driven decisions, identify customer segments, and tailor marketing strategies effectively.

The business report successfully identified the 100 most frequent customers in descending order of rental activity. This analysis was conducted by evaluating and selecting from 16,044 rows of data provided by the detailed table. These results provide valuable insights that can be leveraged to optimize loyalty programs, tailor marketing strategies, and improve inventory management. By understanding which customers are most engaged, the DVD rental store can enhance customer satisfaction, increase operational efficiency, and ultimately drive business growth.

### A1. Specific Fields needed for the Detailed Table and Summary Table

***Detailed Tables Fields***:

- **customer_id** (customer table)
- **first_name** (customer table)
- **last_name** (customer table)
- **email** (customer table)
- **rental_id** (rental table)

- **rental_date** (rental table)
- **return_date** (rental table)

***Summary Table Fields (****aggregating data from the 'detailed' table****):***

- **full_name** (first name and last name combined)
- **email**
- **customer_count**

## A2. Describe the Types of Data Fields used for the Report

The data types for the fields of the '***detailed***' table include:

- **customer_id** ('INT'): This field is used to store the unique identifier for each customer. The integer data type is used because customer IDs are typically numeric and do not require decimal places.

- **first_name** ('VARCHAR (50)'): This field will store the first name of the customers. The varchar(50) data type is appropriate because names are text fields and typically do not exceed 50 characters.

- **last_name** ('VARCHAR (50)'): This field will store the last name of the customers. The varchar(50) data type is appropriate because names are text fields and typically do not exceed 50 characters.

- **email** ('VARCHAR (100)'): This field stores the email addresses of customers. The varchar(100) data type allows for varying lengths of email addresses, which usually do not exceed 100 characters.

- **rental_id** ('INT'): This field stores the unique identifier for each rental transaction. The integer data type is appropriate for storing numeric values.

- **rental_date** ('TIMESTAMP'): This field stores the date and time when a rental occurred. The timestamp data type is used to capture both the date and time.

- **return_date** ('TIMESTAMP'): This field stores the date and time when a return occurred. The timestamp data type is used to capture both the date and time.

The data types for the fields of the '***summary***' table include:

- **full_name** ('VARCHAR(220)'): This field stores the full name of the customer by combining the first and last names. The varchar(220) data type is used to ensure enough space for the combined name.

- **email** ('VARCHAR(100)'): This field stores the customer's email address. The varchar(100) data type allows for email addresses up to 100 characters long.

- **customer_count** ('INT'): This field stores the total number of rentals a customer has made. The integer data type is appropriate because it represents a count, which is a whole number.

## A3. The two tables from the original dataset will provide the data for the detailed table and summary table.

The two tables given from the original database that will help provide data to the detail and summary tables are the ***rental table*** and ***customer table***. These tables have the fields we need to combine for both tables we will create and use for our analysis.

## A4. Custom Transformation within the Detailed Table

A transformation that will be created within the ***detailed*** table that will combine the **first_name** and **last_name** fields found originally as two separate fields within the ***customer*** table into one field called the **full_name** field. This transformation is helpful because it improves readability because it creates a more user-friendly view that is of a more intuitive format. This can make it easier for people reading or using the report to recognize customers quickly. Having a single **full name** column simplifies queries and reports when you want to display a customer's name. Without combining them, every time you query or display a customer's name, you would need to concatenate the first and last names in your SQL queries or in your reporting tool.

## A5. The Business Used of the Detailed Table and the Summary Table.

The **detailed table** is essential for operational insights and customer-specific strategies, while the **summary table** gives us a quick glance summary of the detailed table that can provide support for strategic planning and high-level decision-making. Together, they provide a comprehensive view of customer behavior and business performance, enabling the business to optimize both day-to-day operations and long-term strategies. Understanding how frequently customers visit and how often they return can help identify your most loyal customers. This data can be used to develop and optimize loyalty programs, offering special deals or rewards for frequent renters. It would be possible to create targeted marketing campaigns that appeal to customers who visit frequently or those who haven't visited in a while. The reports can also be used for predictive analysis for inventory management;

knowing when customers are most likely to return and rent again allows the business to manage inventory and ensure popular movies are available at the right times.

## A6. Refreshing Reports (Update Interval)

The update interval is the period between each report refresh. In this business case, to ensure that the data on customer visits and rental activity remains current and actionable, the reports detailing how often customers visit the store and how much they rent will be refreshed monthly.

- **Refresh Frequency:** The reports will be updated **on the first day of every month** to reflect data from the previous month.
- **Scope of Data:** Each refresh will include all customer visit and rental data collected during the preceding month, ensuring that the business has access to the most up-to-date information for analysis.
- **Purpose:** Monthly refreshes allow the business to evaluate trends, identify frequent customers, and adjust marketing or inventory strategies accordingly.

## Section B: Code for the transformation identified in Part A4. The user defined function is also recreated withing this section.

```
/*

The code below performs the transformation identified in Part A4,

which is to combine the first name and last name fields into a single field

and insert the transformed data into the summary table. It also creates the function for

section E to use.

*/


-- Drop the existing trigger that depends on the summary_refresh_function function

DROP TRIGGER IF EXISTS summary_refresh ON detailed;


-- Drop the existing function if it exists, to avoid duplicates

DROP FUNCTION IF EXISTS summary_refresh_function();


-- Create the summary_refresh_function, which is a trigger function
```

```
        CREATE FUNCTION summary_refresh_function()
        RETURNS TRIGGER  -- This function returns a trigger, meaning it is fired by certain events like INSERT or UPDATE
        LANGUAGE plpgsql  -- The function is written in PL/pgSQL
        AS $$
        BEGIN
           -- Empties the summary table by deleting all its records
           DELETE FROM summary;
        -------------------------------------------------------------------------------------------------------------------
           /*
           Insert the aggregated data into the summary table by combining
           the last name and first name into a single full name field.
           It counts how many rentals each customer has made, and groups
           the data by customer_id, last_name, first_name, and email.
           */
           INSERT INTO summary (full_name, email, customer_count)
           SELECT
              concat_ws(' ', last_name, first_name) AS full_name,  -- Concatenates first and last names into full name
              email,
              COUNT(customer_id) AS customer_count  -- Counts the number of rentals per customer
           FROM detailed
           GROUP BY customer_id, last_name, first_name, email  -- Groups the data by customer for aggregation
```

```
    --HAVING COUNT(customer_id) > 20  -- (Optional) If you want to limit it to customers with more than 20 rentals

    ORDER BY customer_count DESC  -- Orders the results by the customer_count in descending order

    LIMIT 100;  -- Limits the result to the top 100 customers with the highest rental counts


    RETURN NEW;  -- This is needed for triggers that fire on data modifications (like INSERT)

END;

$$; -- End of the function, this will refresh the summary table with the new data from the detailed table
```

**Section C: Code that creates the detailed table and summary table.**

```
------------------- Section C. Create Detailed Table ------------------

/*

The code below creates an empty table titled 'detailed',

which will hold the data extracted in section D. It includes columns for

customer information, rental details, and return dates.

*/


-- Drop the 'detailed' table if it already exists to avoid conflicts when recreating it

DROP TABLE IF EXISTS detailed;


-- Create the 'detailed' table with the necessary columns

CREATE TABLE detailed (

    customer_id integer,        -- Holds the unique identifier for each customer
    first_name varchar(50),     -- Holds the customer's first name, up to 50 characters
    last_name varchar(50),      -- Holds the customer's last name, up to 50 characters
    email varchar(100),         -- Stores the customer's email address, up to 100 characters
    rental_id integer,          -- Stores the unique rental identifier for each rental
    rental_date timestamp,      -- Stores the date and time the rental occurred
    return_date timestamp       -- Stores the date and time the rental was returned
);

-- Detailed table was created successfully
-- Verify the structure of the detailed table by selecting all columns (no data yet)
SELECT * FROM detailed;



--------------------------- Section C. Create Summary Table ---------------------------
/*
The code below creates an empty table titled 'summary',
which will store aggregated information from the 'detailed' table,
such as the full name of customers, their email, and the count
of their rental activity (customer_count).
*/

-- Drop the 'summary' table if it already exists to avoid conflicts when recreating it
DROP TABLE IF EXISTS summary;

-- Create the 'summary' table with the necessary columns for aggregated data
```

```
    CREATE TABLE summary (

        full_name varchar(220),     -- Stores the full name of the customer, combining first and last name

        email varchar(100),         -- Stores the customer's email address, up to 100 characters

        customer_count integer      -- Stores the count of rentals made by the customer

    );


    -- Summary table was created successfully

    -- Verify the structure of the summary table by selecting all columns (no data yet)

SELECT * FROM summary;
```

**Section D: Code extracts raw data from the database and inserts it into the detailed table.**

```
            ---------------------------- Section D. Extract Raw Data -----------------------------


    /*

    The code below queries the raw data from the original database tables

    'customer' and 'rental' and inserts this data into the 'detailed' table.

    It combines the relevant customer information with the rental details

    by joining the 'customer' and 'rental' tables based on the customer_id.

    */


    -- Insert customer and rental data into the 'detailed' table

    INSERT INTO detailed (

        customer_id,     -- Customer's unique identifier

        first_name,      -- Customer's first name
        last_name,       -- Customer's last name
        email,           -- Customer's email address
        rental_id,       -- Rental's unique identifier
        rental_date,     -- Date when the rental was made
        return_date      -- Date when the rental was returned
    )
    -- Select data from the customer and rental tables for insertion
    SELECT
        c.customer_id, c.first_name, c.last_name, c.email,   -- Customer details
        r.rental_id, r.rental_date, r.return_date            -- Rental details
    FROM rental AS r
    INNER JOIN customer AS c ON c.customer_id = r.customer_id;  -- Join the 'customer' and 'rental'
tables by customer_id


    -- Display data to verify that insertion into the 'detailed' table was successful
    SELECT * FROM detailed;
```

**Section E: Create a trigger on the detailed table of the report that will continually update the summary table as data is added to the detailed table.**

```
    -------------------------------- Section E. Create a Trigger --------------------------------------------------------
----------
    /*
    The code below creates a trigger on the 'detailed' table.
    This trigger will continually update the 'summary' table as new data is added
```

```
            to the 'detailed' table. The trigger will execute the 'summary_refresh_function'

            after every insert statement on the 'detailed' table.

            */



            -- This trigger will execute the 'summary_refresh_function' after an insert on the 'detailed' table

            CREATE TRIGGER summary_refresh

            AFTER INSERT ON detailed  -- Trigger fires after each insert operation

            FOR EACH STATEMENT         -- Trigger fires once per insert statement, not per row

            EXECUTE PROCEDURE summary_refresh_function();  -- The trigger executes the
'summary_refresh_function'



            -- Verify that the trigger was created successfully

            SELECT trigger_name

            FROM information_schema.triggers

            WHERE event_object_table = 'detailed'  -- Check the 'detailed' table for triggers

              AND trigger_name = 'summary_refresh';  -- Verify that the 'summary_refresh' trigger exists



            -- Insert test data into the 'detailed' table to test if the trigger works

            --INSERT INTO detailed (customer_id, first_name, last_name, email, rental_id, rental_date,
return_date)

            -- We must specify inserting the rows 50 times so that we can view the row first in the summary
table because of the DESC order

            INSERT INTO detailed (customer_id, first_name, last_name, email, rental_id, rental_date,
return_date)

            SELECT 2, 'Celeste', 'Catala', 'cc@example.edu', 1003, '2024-01-03', '2024-01-10'

            FROM generate_series(1, 50);
```

```
	-- Verify that the data was successfully inserted into the 'detailed' table by selecting the newly inserted row where the customer_id = 2

	SELECT * FROM detailed

	WHERE customer_id = 2;  -- Check if the test row with customer_id = 2 was inserted


	-- Verify that the insertion is reflected within the summary table

	SELECT * FROM summary;
```

## Section F. Stored procedure is created to refresh the data in both the detailed and summary tables and perform the raw data extraction from Part D.

```
	-------------------------------  Section F. Stored Procedure ------------------------------------------------

	/*

	Stored procedure is created to refresh the data in both the 'detailed' and 'summary' tables.

	It clears the contents of the 'detailed' table and performs the raw data extraction

	from the 'customer' and 'rental' tables. This ensures that the 'detailed' table is

	updated with the most recent data. The 'summary' table will be automatically updated

	due to the trigger associated with the 'detailed' table.

	*/


	-- Drop the stored procedure if it already exists to avoid conflicts when creating it again

	DROP PROCEDURE IF EXISTS refresh_tables();


	-- Create the 'refresh_tables' procedure to refresh the contents of the 'detailed' and 'summary' tables

	CREATE PROCEDURE refresh_tables()

	LANGUAGE plpgsql  -- Define the procedure using the PL/pgSQL language
```

```
    AS $$
    BEGIN

        -- Empty the contents of the 'detailed' table
        DELETE FROM detailed;

        /*
        Insert new data into the 'detailed' table by selecting
        customer and rental data from the 'customer' and 'rental' tables,
        joining them based on the 'customer_id'. This refreshes the detailed table.
        */
        INSERT INTO detailed (
            customer_id,
            first_name,
            last_name,
            email,
            rental_id,
            rental_date,
            return_date
        )
        SELECT
            c.customer_id, c.first_name, c.last_name, c.email,  -- Customer details
            r.rental_id, r.rental_date, r.return_date           -- Rental details
        FROM rental AS r
        INNER JOIN customer AS c ON c.customer_id = r.customer_id;  -- Join the tables on
customer_id
```

```
END;

$$;


-- Call the stored procedure to refresh the 'detailed' table with the latest data

CALL refresh_tables();


-- View the stored procedure refreshed the detail table and removed the data we inserted

SELECT * FROM detailed
WHERE customer_id = 2;  -- Verify that the table was refreshed where customer_id = 2

--Confirm that the summary table has been refreshed and removed the data we inserted

SELECT * FROM summary;
```

## F1. Identify a relevant job scheduling tool that can be used to automate the stored procedure.

A relevant job scheduling tool that can be used to automate the stored procedure is pgAgent which is designed specifically for PostgreSQL. It allows you to schedule and automate routine tasks such as executing stored procedures, running SQL scripts, and maintaining the database. It runs as a separate service and can execute jobs at specific times or intervals, making it ideal for automating stored procedures like your refresh_tables() function. A job scheduler would be beneficial to implement because it automates repetitive tasks and saves time and increases reliability.

**How to use pgAgent for This Business Case:**

- Objective: You need to schedule your refresh_tables() stored procedure to periodically refresh the data in your detailed and summary tables.
- Using pgAgent:
  - You would set up a pgAgent job to run the refresh_tables() stored procedure automatically, for example, every night at midnight. This ensures your detailed table is always up to date with the latest data and your summary table is refreshed through the trigger.
- Job Setup:
  1. Create a new job in pgAgent, specifying a name and description for the job (e.g., "Refresh Tables Job").
  2. Set the schedule for the job to define when the procedure should run (e.g., every day at 12:00 AM).
  3. Add a step to the job that executes your stored procedure (CALL refresh_tables();).
  4. Enable the job and monitor its execution through the pgAgent interface or logs.