Overview:

A thriving business relies on customer support. The business exchanges goods and services for money, but will often need some other incentive to retain customers. The goal is to maximize repeat business while simultaneously bringing on new customers. To evaluate repeat business, we might ask how much has each customer spent in total. Answering this question will help develop and monitor a plan to drive repeat business.

As an example, I would like to model a rewards program where customers can earn perks based on the total amount spent at the DVD store. A rewards program can have a powerful psychological effect by offering financial incentives for repeat business. Beyond the financial aspect, this can also give the customer a sense of accomplishment or prestige if they are recognized for their contributions.

Customers can be categorized as such:

Tier Name	**Amount Spent**
Standard	Under $100
Gold	$100 - $200
Platinum	Over $200

Customers above the “Standard” tier may receive perks such as:

- BOGO rental (rent one DVD, get a second rental free)
- Free concessions or merchandise with rental
- Ability to waitlist certain popular titles
- Tour of managers office (where the magic happens!)
- Etc.

Actual perks and tier naming is beside the point of this project. Instead, separating customers into these categories would give statistics to better estimate profits and losses when considering perks. This report will also indicate if a customer qualifies for the rewards during transactions. Contact information will also be provided for targeted marketing campaigns or updates to the program.

SECTION A - Detailed Table

The detailed table will include:

Column Name	**From which table?**	**Type**
customer_id	customer	integer
first_name	customer	varchar
last_name	customer	varchar
email	customer	varchar
total_revenue	payment	numeric
tier	payment	varchar

The columns *customer_id*, *first_name*, *last_name* and *email* are all pretty straightforward. Information will come directly from the *customer* table and used as context for the remaining columns. Since the types are known from the *customer* table, the same types will be used here.

The column *total_revenue* will be the sum of all existing purchases per customer account.. This information can be calculated by using the *amount* column in the *payment* table since each transaction is also linked to a *customer_id*.

The result of the *total_revenue* calculation will influence the *tier* column. The descriptions are assigned “Standard”, “Gold’, or “Platinum” based on the defined ranges of each tier with a user-defined function.

Information from this table would be used to cross reference what perks and rewards a customer qualifies for during transactions. Because the perks provided often come at a loss of revenue, it is important that every transaction is consistent and fits with the promise made to the customer. Furthermore, all information provided by this table would be useful for generating reminders and targeted marketing campaigns via email. The assigned tier would determine an email template, first and last name in the body of the email to greet the customer, and lastly the email address for the destination. This type of contact can help drive business, but it can also be used to show gratitude toward the customer.

Since this information will most likely be referenced during daily transactions, it should be refreshed daily. The exact time would preferably be a few hours after the store closes to ensure everything is updated before the next morning.

SECTION A - Summary Table

The summary table will include:

Column Name	**From which table?**	**Type**
tier	detailed_table	text
number_of_customers	detailed_table	numeric

The summary table will be two columns. The first column will contain each distinct tier name and the second will be the count of customers in that tier. This information will be calculated and populated whenever changes are made to the detailed table.

Information in the summary table can be used to track growth of repeat customers. Ideally, there should be growth in all three categories of customer. However, if the business is in a small town with a limited population, there should be a growth in “gold” and “platinum” over time. This information might also be used to estimate potential profit and loss when considering perks or incentives for the rewards program since a specific number of customers in each category is known.

Section B - Function Code

Information in this report relies on categorizing customers based on the total amount spent. Customers who have spent less than $100 are categorized as “standard”, customers who have spent between $100 and $200 are “gold”, and customers who have spent more than $200 are “platinum”. The following is a function to complete this task:

```
CREATE OR REPLACE FUNCTION tier_name(total_spent numeric)
RETURNS text
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN CASE
    WHEN total_spent >= 100 AND total_spent < 200
        THEN 'gold'
    WHEN total_spent >= 200
        THEN 'platinum'
    ELSE
        THEN 'standard'
    END;
END;
$$;
```

As a slight deviation from the specific ranges, only “gold” and platinum are defined in code. Everything else will result in the return value of “standard”. This is done to ensure that any unexpected values passed in, such as a negative number or null, will return an expected result.

Section C - Table Creation

A table will be used to store detailed customer information necessary for this report. This table will be called *revenue_per_customer* and created with the following code:

```
CREATE TABLE revenue_per_customer (
    customer_id int,
    first_name  varchar(45),
    last_name   varchar(45),
    email       varchar(50),
    total_spent numeric,
    tier        text
);
```

The first four columns will have identical types as the corresponding columns from the *customer* table. The calculated value *total_spent* will be stored as numeric as this value will come from the SUM function later. The value inserted in the *tier* column will be the return value of the *tier_name* function described previously. As such the types will match.

A table summarizing the *revenue_per_customer* table will be named *tier_totals* and is defined using the following code:

```
CREATE TABLE tier_totals (
    tier text,
    number_of_customers bigint
);
```

This table will be small and straightforward. The type for *tier* corresponds to the same column in *revenue_per_customer*. Values stored in *number_of_customers* will be generated with the COUNT function resulting in the bigint type.

Section D - Query to Extract Data for the Detailed Table

In order to fill the *revenue_per_customer* table, data will be extracted from the *customer* table and the *payment* table. The *customer* table will fill in all contact information requested, however information from the *payment* table will need to be processed first. The following is a SELECT statement that will do just this:

```sql
WITH customer_revenue AS (
    SELECT
        customer_id,
        SUM(amount) AS total_spent
    FROM payment
    GROUP BY customer_id
)
SELECT
    customer.customer_id,
    customer.first_name,
    customer.last_name,
    customer.email,
    COALESCE(customer_revenue.total_spent, 0) AS total_spent,
    tier_name(customer_revenue.total_spent) AS tier
FROM customer
LEFT JOIN customer_revenue
    ON customer.customer_id = customer_revenue.customer_id
ORDER BY total_spent DESC;

SELECT COUNT(*) FROM customer;
SELECT COUNT(DISTINCT customer_id) FROM payment;
```

The total amount spent per customer is first processed using a Common Table Expression (CTE) represented by a WITH statement. Inside the CTE, payment amounts are totaled per unique *customer_id* with a SELECT statement. The total amount spent per customer will need to be known in order to assign the tier name via the *tier_name()* function.

When combining these two tables, LEFT JOIN is utilized to ensure the entire *customer* table is represented. Since *customer_revenue* derives its data from the *payment* table, it is not guaranteed that all customers from the *customer* table will be present. There may be customers that are in the system that haven't made any transactions, however these customers should still be accounted for in this report. Since this is a possibility, it is somewhat expected for some values in the *total_spent* column to be null.

To avoid null in the *total_spent* column, the function COALESCE is used. This function will return the first argument that is not null, thus zero will act as a default value. Similarly, the default value for the *tier_name* function is ‘standard’ avoiding a null value.

Section E - Trigger to Update Summary Table

The summary table *tier_totals* should be updated when the *revenue_per_customer* is updated. The following code will accomplish this:

```
CREATE OR REPLACE FUNCTION insert_customer()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    DELETE FROM tier_totals;
    INSERT INTO tier_totals
    SELECT
        tier,
        COUNT(customer_id)
    FROM revenue_per_customer
    GROUP BY tier;
    RETURN NEW;
END;
$$;

CREATE TRIGGER update_tier_totals
AFTER INSERT OR DELETE
ON revenue_per_customer
FOR EACH STATEMENT
EXECUTE PROCEDURE insert_customer();
```

First a function is created to refresh *tier_totals*. This function deletes all data in the table and replaces it with the totals of each tier. Then, a TRIGGER is created which will run whenever an INSERT statement is made on the *revenue_per_customer* table. This will populate the *tier_totals* table ensuring that the summary table will always be accurate.

Section F - Stored Procedure to Refresh Tables

Finally, everything can be connected together with a stored procedure. After table creation, the following code will populate the tables:

```
CREATE OR REPLACE PROCEDURE refresh_customer_revenue()
LANGUAGE plpgsql
AS $$
BEGIN
    DELETE FROM revenue_per_customer;
    DELETE FROM tier_totals;

    INSERT INTO revenue_per_customer
    WITH customer_revenue AS (
        SELECT
            customer_id,
            SUM(amount) AS total_spent
        FROM payment
        GROUP BY customer_id
    )
    SELECT
        customer.customer_id,
        customer.first_name,
        customer.last_name,
        customer.email,
        COALESCE(customer_revenue.total_spent, 0),
        tier_name(customer_revenue.total_spent)
    FROM customer
    LEFT JOIN customer_revenue
        ON customer.customer_id = customer_revenue.customer_id
    ORDER BY total_spent DESC;
RETURN;
END;
$$;
```

First, both tables are cleared. Then an INSERT statement is used with the SELECT statement described earlier. When the INSERT statement is made, the TRIGGER statement will execute, updating the summary table *tier_totals*. When run, this stored procedure will create the necessary tables to facilitate the rewards program.

As described in Section A, information from the *revenue_per_customer* will need to be up-to-date and referenced for daily transactions. Therefore *refresh_customer_revenue* will need to be called daily preferably when the store is closed and information is unlikely to change. To accomplish this, a job scheduler will need to be utilized.

The best job scheduler to accomplish this task is somewhat subjective. For a simple, small database such as this DVD rental database, a Linux based machine would sufficiently handle any database transactions. Most Linux distributions are open-source, lightweight, run on almost any hardware, and highly configurable. To handle the actual job scheduling, cron can be utilized.

Cron is useful for completing any repetitive tasks that could be automated, which is perfect for refreshing a database. A very simple bash script can be created as a crontab which will specify instructions to run *refresh_customer_revenue* at the desired time daily.

Section G - Video Link

https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=719c36e9-6843-48c0-a4a3-afc30051216c

Section H - Sources

For this project I only utilized class materials and the official PostgreSQL documentation.