Samy Asadi
Advanced Data Management - D326
September 25, 2025

CSN - 1 Task 1: Data Analysis

Section A. Business Report

Summary of Report

This business report answers the question: **Which film categories generate the most revenue for the DVD rental business, by month?** The report delivers two tables at different levels of detail. The **detailed table** captures each rental event (one row per rental) with its film, customer, category and per-rental revenue data. Any totals in this table (such as `amount_usd`, `payment_count`) are aggregated only per `rental_id` not across categories or months.
The **summary table** aggregates by `category_name` and `rental_month`, reporting the number of rentals, total revenue and average revenue per rental within each category–month. Together, these outputs support decisions about promotions, inventory and marketing by identifying high-performing categories on a monthly cadence.

A1. Specific Fields in the Detailed Table & the Summary Table

Detailed Table **`reporting.rental_detail`**

```
rental_id
rental_date
return_date
billed_days
customer_id
customer_name
store_id
staff_id
film_id
film_title
category_name
rental_rate
price_tier
amount_usd (revenue)
payment_count
rental_month
```

Summary Table **`reporting.rental_summary`**

`category_name`
`rental_month`
`total_rentals` – COUNT of rentals
`total_revenue` – SUM of rental revenue

A2. Types of Data Fields Used for The Report

Integer:

`rental_id, customer_id, store_id, staff_id, film_id, billed_days, payment_count, total_rentals`

Text (String):

`film_title, category_name, price_tier, customer_name`

Timestamp

`rental_date, return_date`

Numeric (Decimal)

`rental_rate, amount_usd, total_revenue`

Date

`rental_month`

A3. Source Tables from the Dataset

- **`public.rental`** (base tables for rental events with dates, customer and inventory links)
- **`public.inventory`** (connects rental to film)
- **`public.film`** (title, rental_rate for price_tier UDF)
- **`public.film_category`** (link film to categories)
- **`public.category`** (category name)
- **`public.customer`** (customer name fields)
- **`public.store (joined via inventory.store_id)`**
- **`public.payment`** (revenue per rental via SUM and COUNT)

A4. Custom Transformation

Field: **`price_tier`** derived from **`film.rental_rate.`**
Transformation: A user-defined function **`(reporting.price_tier_for_rate)`** maps numeric rates into labeled tiers [Budget, Standard, Premium].
Why: Non-technical stakeholders will understand labeled price tiers faster than raw numbers. This makes the report easier to read and interpret.

A5. Explanation of Different Business Uses

Detailed Table: `reporting.rental_detail`

The detailed table is a rental event log at one row per rental. This table is useful to analysts and operations staff because they can see every transaction line by line. Which customer rented which film, the film's category, when it was returned and how revenue that rental generated. This fine-grain detail supports auditing (verifying that payments align with rental duration and rate), detecting anomalies (unusually long rentals) and mainly tracing through lines from individual events to the summary results.

Summary Table - `reporting.rental_summary`

The summary table serves those who need a broader picture, like store managers or executives. It aggregates by category name and rental month together, consolidating many rental events down to two key metrics: total number of rentals and total revenue. This view quickly shows which categories (Horror, Action, Comedy, etc.) are generating the most revenue each month and supports high-level decisions such as which genres to promote or discount and how to design future inventory purchasing to maximize revenue.

A6. Report Refresh Frequency

This report should be refreshed monthly, within the first three days of the new month in order to cover the previous month’s rentals. A monthly cadence is optimal, as quarterly or annual refreshes do not keep pace with the fast-paced consumer environment. A weekly or biweekly refresh is too small a sample size to provide actionable data, as it could be skewed by one-off events such as holidays or cultural events like the Super Bowl. A monthly refresh is the ideal snapshot, balancing timeliness and practicality. This ensures stakeholders always have current and relevant information to make broad business decisions such as promotions, pricing and inventory.

Operationally, the refresh is executed with CALL reporting.refresh_rental_reports(), and the AFTER INSERT trigger ensures the summary remains consistent during the bulk load.

B. Provide original code for function(s) in text format that perform the transformation(s) you identified in part A4.

```
DROP FUNCTION IF EXISTS reporting.price_tier_for_rate(NUMERIC);

CREATE OR REPLACE FUNCTION reporting.price_tier_for_rate(rate NUMERIC)
RETURNS TEXT
LANGUAGE plpgsql
IMMUTABLE
STRICT
AS $$
BEGIN
  IF rate < 1.99 THEN
    RETURN 'Budget';
  ELSIF rate < 4.00 THEN
    RETURN 'Standard';
  ELSE
    RETURN 'Premium';
  END IF;
END;
$$;
```

C. Provide original SQL code in a text format that creates the detailed and summary tables to hold your report table sections.

Detailed table (one row per rental event):

`reporting.rental_detail`

```
DROP TABLE IF EXISTS reporting.rental_detail CASCADE;

CREATE TABLE reporting.rental_detail (
  rental_id      INTEGER       PRIMARY KEY,
  rental_date    TIMESTAMP     NOT NULL,
  return_date    TIMESTAMP,
  billed_days    INTEGER       NOT NULL,
  customer_id    INTEGER       NOT NULL,
  customer_name  TEXT          NOT NULL,
  store_id       INTEGER       NOT NULL,
  staff_id       INTEGER       NOT NULL,
  film_id        INTEGER       NOT NULL,
  film_title     TEXT          NOT NULL,
  category_name  TEXT          NOT NULL,
  rental_rate    NUMERIC(5,2)  NOT NULL,
  price_tier     TEXT          NOT NULL,              -- via UDF at load
  amount_usd     NUMERIC(10,2) NOT NULL DEFAULT 0,    -- per-rental payments sum
  payment_count  INTEGER       NOT NULL DEFAULT 0,    -- count of payment rows
  rental_month   DATE          NOT NULL               -- date_trunc('month',
rental_date)::date
);
```

Summary Table

`reporting.rental_summary:`

```
DROP TABLE IF EXISTS reporting.rental_summary CASCADE;

CREATE TABLE reporting.rental_summary (
  category_name  TEXT          NOT NULL,
  rental_month   DATE          NOT NULL,
  total_rentals  INTEGER       NOT NULL,
  total_revenue  NUMERIC(10,2) NOT NULL,
  PRIMARY KEY (category_name, rental_month)
);
```

D. Provide an original SQL query in a text format that will extract the raw data needed for the detailed section of your report from the source database.

```
SELECT
  r.rental_id,
  r.rental_date,
  r.return_date,
  GREATEST(1, (r.return_date::date - r.rental_date::date)) AS billed_days,
  c.customer_id,
  (c.first_name || ' ' || c.last_name) AS customer_name,
  st.store_id,
  r.staff_id,
  f.film_id,
  f.title AS film_title,
  cat.name AS category_name,
  f.rental_rate,
  reporting.price_tier_for_rate(f.rental_rate) AS price_tier,
  ROUND(COALESCE(SUM(p.amount), 0)::numeric, 2) AS amount_usd,
  COUNT(p.payment_id) AS payment_count,
  date_trunc('month', r.rental_date)::date AS rental_month
FROM public.rental r
JOIN public.inventory      i   ON i.inventory_id   = r.inventory_id
JOIN public.film           f   ON f.film_id        = i.film_id
JOIN public.film_category  fc  ON fc.film_id       = f.film_id
JOIN public.category       cat ON cat.category_id  = fc.category_id
JOIN public.customer       c   ON c.customer_id    = r.customer_id
JOIN public.store          st  ON st.store_id      = i.store_id
LEFT JOIN public.payment   p   ON p.rental_id      = r.rental_id
WHERE r.return_date IS NOT NULL
GROUP BY
  r.rental_id, r.rental_date, r.return_date,
  c.customer_id, c.first_name, c.last_name,
  st.store_id, r.staff_id,
  f.film_id, f.title, cat.name, f.rental_rate,
  date_trunc('month', r.rental_date)::date;
```

E. Provide original SQL code in a text format that creates a trigger on the detailed table of the report that will continually update the summary table as data is added to the detailed table.

Trigger: keep reporting.rental_summary in sync when rows are added to reporting.rental_detail

```
DROP TRIGGER IF EXISTS trg_rental_detail_after_insert ON reporting.rental_detail;
DROP FUNCTION IF EXISTS reporting.trgfn_rental_detail_after_insert();

CREATE OR REPLACE FUNCTION reporting.trgfn_rental_detail_after_insert()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
  INSERT INTO reporting.rental_summary (
      category_name, rental_month, total_rentals, total_revenue
  )
  VALUES (
      NEW.category_name,
      NEW.rental_month,
      1,
      NEW.amount_usd
  )
  ON CONFLICT (category_name, rental_month)
  DO UPDATE
     SET total_rentals = reporting.rental_summary.total_rentals + 1,
         total_revenue = reporting.rental_summary.total_revenue +
EXCLUDED.total_revenue;

  RETURN NEW;
END;
$$;

-- Attach the AFTER INSERT trigger to the detail table
CREATE TRIGGER trg_rental_detail_after_insert
AFTER INSERT ON reporting.rental_detail
FOR EACH ROW
EXECUTE FUNCTION reporting.trgfn_rental_detail_after_insert();
```

F. Provide an original stored procedure in a text format that can be used to refresh the data in both the detailed table and summary table. The procedure should clear the contents of the detailed table and summary table and perform the raw data extraction from part D.

```
— Stored procedure: refresh both detailed and summary tables
-- Assumes the AFTER INSERT trigger on reporting.rental_detail maintains
reporting.rental_summary.

CREATE OR REPLACE PROCEDURE reporting.refresh_rental_reports()
LANGUAGE plpgsql
AS $$
BEGIN

  TRUNCATE TABLE reporting.rental_summary;
  TRUNCATE TABLE reporting.rental_detail;

  -- Reload DETAIL from source
  INSERT INTO reporting.rental_detail (
    rental_id,
    rental_date,
    return_date,
    billed_days,
    customer_id,
    customer_name,
    store_id,
    staff_id,
    film_id,
    film_title,
    category_name,
    rental_rate,
    price_tier,
    amount_usd,
    payment_count,
    rental_month
  )
  SELECT
    r.rental_id,
    r.rental_date,
    r.return_date,
    GREATEST(1, (r.return_date::date - r.rental_date::date)) AS billed_days,
    c.customer_id,
    (c.first_name || ' ' || c.last_name) AS customer_name,
    st.store_id,
    r.staff_id,
    f.film_id,
    f.title AS film_title,
    cat.name AS category_name,
    f.rental_rate,
    reporting.price_tier_for_rate(f.rental_rate) AS price_tier,
    ROUND(COALESCE(SUM(p.amount), 0)::numeric, 2) AS amount_usd,
    COUNT(p.payment_id) AS payment_count,
    date_trunc('month', r.rental_date)::date AS rental_month
  FROM public.rental r
  JOIN public.inventory      i   ON i.inventory_id   = r.inventory_id
  JOIN public.film           f   ON f.film_id        = i.film_id
  JOIN public.film_category  fc  ON fc.film_id       = f.film_id
  JOIN public.category       cat ON cat.category_id  = fc.category_id
  JOIN public.customer       c   ON c.customer_id    = r.customer_id
  JOIN public.store          st  ON st.store_id      = i.store_id
  LEFT JOIN public.payment   p   ON p.rental_id      = r.rental_id
  WHERE r.return_date IS NOT NULL
  GROUP BY
```

```
    r.rental_id, r.rental_date, r.return_date,
    c.customer_id, c.first_name, c.last_name,
    st.store_id, r.staff_id,
    f.film_id, f.title, cat.name, f.rental_rate,
    date_trunc('month', r.rental_date)::date;

END;
$$;

-- To run the refresh:
reporting.refresh_rental_reports();
```

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

pgAgent because it is the job scheduler for PostgresSQL that integrates with pgAdmin. User can create a job with a single SQL step and schedule it monthly.

G. Provide a Panopto video recording that includes the presenter and a vocalized demonstration of the functionality of the code used for the analysis.

https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=aca3cec1-65a4-477b-9759-b36500824977

H. Sources, Acknowledgments, and Citations

Statement of use. This submission was authored by the student with assistance from **ChatGPT** for drafting, proofreading, and validating SQL structure against PostgreSQL best practices (in-text citation: *(ChatGPT, 2025)*). General SQL reminders were cross-checked with **W3Schools**. Technical specifics for triggers, procedures, upserts, and time bucketing were verified against the **official PostgreSQL documentation**. Scheduling guidance referenced **pgAgent** docs. Dataset provenance/replication notes used the **DVD Rental** sample database guide. No third-party code was copied; all SQL in this submission is original. (W3Schools; PostgreSQL Docs; pgAdmin/pgAgent; Neon DVD Rental.) Neon+10W3Schools+10PostgreSQL+10

In-text citation examples used in this document: *(ChatGPT, 2025); (W3Schools, n.d.); (PostgreSQL Docs, 2025); (pgAdmin/pgAgent Docs, 2025); (Neon, 2024).*

References

- **OpenAI.** (2025). *ChatGPT (GPT-5 Thinking)* — conversational assistance on SQL design and project documentation.
- **W3Schools.** (n.d.). *SQL Tutorial.* (general SQL syntax reference). W3Schools
- **PostgreSQL Global Development Group.** (2025). *CREATE TRIGGER* (official syntax and privileges). PostgreSQL

- **PostgreSQL Global Development Group.** (2025). *PL/pgSQL: Trigger Functions* (writing trigger functions; RETURN NEW). PostgreSQL
- **PostgreSQL Global Development Group.** (2025). *CREATE PROCEDURE* (defining procedures). PostgreSQL
- **PostgreSQL Global Development Group.** (2025). *CALL* (invoking procedures). PostgreSQL
- **PostgreSQL Global Development Group.** (2025). *INSERT — ON CONFLICT* (official upsert behavior). PostgreSQL
- **PostgreSQL Global Development Group.** (2025). *Date/Time Functions* — date_trunc (time bucketing). PostgreSQL
- **PostgreSQL Global Development Group.** (2025). *Triggers (overview chapter)* (conceptual background). PostgreSQL
- **pgAdmin Project.** (2025). *pgAgent* (job scheduler overview and setup). pgAdmin
- **pgAdmin Project.** (2025). *Creating a pgAgent Job* (creating jobs/steps/schedules). pgAdmin
- **Neon.** (2024). *PostgreSQL Sample Database — DVD Rental* (download and ERD). Neon