DVD Rental Business Insights

This comprehensive report provides an analysis of our DVD rental business, encompassing a variety of critical metrics and insights, reflecting our operations' health, customer engagement, and overall financial performance. The contents are structured to offer a summarized breakdown of customer activity, revenue generation, inventory utilization, and popularity metrics of our rental inventory by rental category.

Customer Activity Analysis: The report begins with an in-depth exploration of customer rental patterns, highlighting trends in customer preferences and frequency of rentals. This section provides a clearer understanding of our customer base and their engagement levels throughout the year.

Revenue Overview: A detailed examination of the revenue streams from rentals. This includes analysis of peak periods, comparison of month over month performance, and identification of the employees responsible for that revenue.

Inventory Utilization And Performance: Our analysis on inventory utilization offers a look into the efficiency of our current stock management. It identifies the rates at which different categories of DVDs are rented out, helping pinpoint areas where inventory adjustments may be beneficial. This section will also look at the most and least popular DVDs based on rental frequency. By understanding customer preferences, we can tailor our purchasing and promotional strategies to better align with demand.

Procedures and Queries: To ensure transparency and reproducibility of our findings, the report concludes with a detailed description of the procedures and SQL queries used to extract data from the "dvdrental" database. This part serves as a valuable resource for those interested in diving deeper into the data used to create this report. .

This report aims to equip stakeholders with the knowledge required to make informed decisions and to continue enhancing our service offerings to meet customer demands effectively.

Customer Activity Analysis

2005 - 2006 saw a total of 599 customers that took out 16,044rentals, with the most rentals being in July ‘05 and the least rentals being February ‘06. A detailed month over month breakdown by customer is available in the database as a separate detailed report. Information on how to access this data is in the Procedures and Queries section.

	count bigint	date text
1	182	2006-02
2	5686	2005-08
3	6709	2005-07
4	2311	2005-06
5	1156	2005-05

5686 rentals in August ’05

6709 rentals in July ’05

2311 rentals in June ’05

1156 rentals in May ’05

182 rentals in February ‘06

Revenue Overview

This same period saw a total revenue of 61,312 with $30,252 coming from store 1 and $31,059 coming from store 2. Each store only has 1 employee wich is also the manager of that store making Mike Hillyer responsible for store 1, and Jon Stephens responsible for store 2.

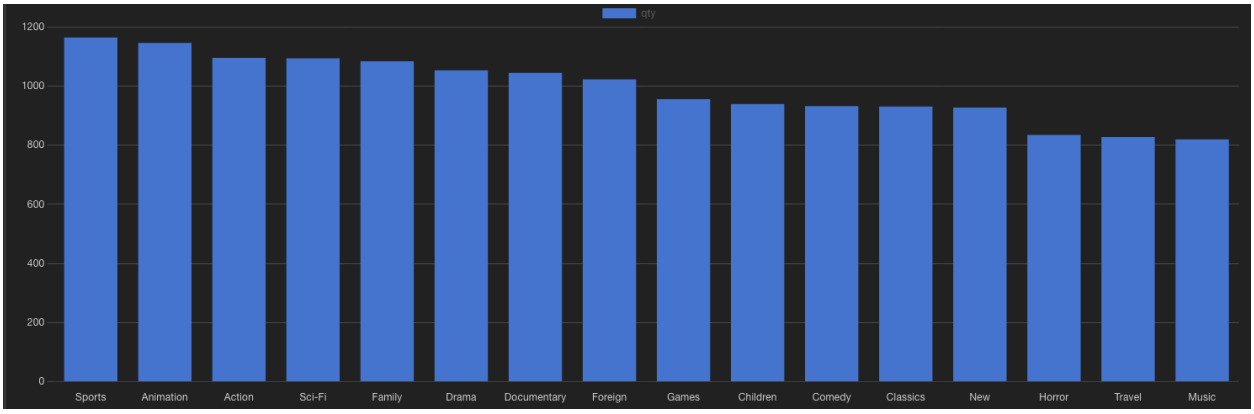
	store smallint	staff_name text	total numeric
1	1	Mike Hillyer	30252.12
2	2	Jon Stephens	31059.92

Note: store 1 total inventory is 2270, store 2 is 2311. Even accounting for inventory difference store 2 has slightly higher overall performance. Store 1 ultimately had generated $13.32 per dvd in their inventory while store 2 generated $13.43

Inventory Utilization

Company wide, 99.97% of the dvd inventory was rented out at least once, with the only non-utilized inventory being a single copy of ‘Academy Dinosaur’. I do believe this to be random happenstance but would also like to take the opportunity to point out the importance of ensuring a proper rotation of inventory at all times to maximize the longevity of our inventory. I would consider anything above 95% to be excellent.

Overall the company had a very predictable spread in terms of rental popularity as the most popular rental categories also had the most available inventory across all stores. There are a few exceptions to this however such as action being 0.18% more popular than science fiction even though they have the same inventory, and Drama being 2.94% more popular than Foreign.



Count of rentals by category

	category character varying (25)	qty bigint
1	Sports	1179
2	Animation	1166
3	Action	1112
4	Sci-Fi	1101
5	Family	1096
6	Drama	1060
7	Documentary	1050
8	Foreign	1033
9	Games	969
10	Children	945
11	Comedy	941
12	New	940
13	Classics	939
14	Horror	846
15	Travel	837
16	Music	830

Inventory Breakdown By Category

	category_name character varying (25)	category_count bigint	percentage_of_total_inventory numeric
1	Sports	344	7.51
2	Animation	335	7.31
3	Sci-Fi	312	6.81
4	Action	312	6.81
5	Family	310	6.77
6	Foreign	300	6.55
7	Drama	300	6.55
8	Documentary	294	6.42
9	Games	276	6.02
10	New	275	6.00
11	Classics	270	5.89
12	Children	269	5.87
13	Comedy	269	5.87
14	Horror	248	5.41
15	Travel	235	5.13
16	Music	232	5.06

Count of rentals by category

Next if we look at popularity by individual titles, the least Popular Rentals of our current inventory are Hardy Robbers, Mixed Doors, and Train Bunch, all with 4 rentals each and the overall most popular rental was Bucket Brotherhood with 34 rentals.

Top 20

	film_id smallint	count bigint	title character varying (255)
1	103	34	Bucket Brotherhood
2	738	33	Rocketeer Mother
3	767	32	Scalawag Duck
4	730	32	Ridgemont Submarine
5	331	32	Forward Temple
6	382	32	Grit Clockwork
7	489	32	Juggler Hardly
8	418	31	Hobbit Alien
9	973	31	Wife Turn
10	369	31	Goodfellas Salute
11	753	31	Rush Goodfellas
12	891	31	Timberland Sky
13	1000	31	Zorro Ark
14	31	31	Apache Divine
15	621	31	Network Peak
16	735	31	Robbers Joon
17	403	30	Harry Idaho
18	450	30	Idols Snatchers
19	374	30	Graffiti Love
20	609	30	Muscle Bright

Bottom 20

	film_id smallint	count bigint	title character varying (255)
1	904	4	Train Bunch
2	400	4	Hardly Robbers
3	584	4	Mixed Doors
4	310	5	Fever Empire
5	612	5	Mussolini Spoilers
6	335	5	Freedom Cleopatra
7	343	5	Full Flatliners
8	699	5	Private Drop
9	441	5	Hunter Alter
10	107	5	Bunch Minds
11	459	5	Informer Double
12	362	5	Glory Tracy
13	781	5	Seven Swarm
14	94	5	Braveheart Human
15	180	5	Conspiracy Spirit
16	903	5	Traffic Hobbit
17	558	5	Mannequin Worst
18	297	6	Extraordinary Conquerer
19	883	6	Tequila Past
20	808	6	Sling Luke

Lastly lets look at a performance metric for each category. This “performance” number is derived by dividing the number of rentals by the available inventory for that category to create a ratio of rentals:inventory per category. This data can be used to determine wich categories we should focus on expanding the most moving foreward as a higher number means the movies are rented out more frequently relative to their available inventory. For example horror is a low performer and a low popularity, ranking 14th by customer rental count and 16th by performance with it being 14th on our inventory percentage list. By contrast Music is a category that stands out as something we should consider increasing the total inventory of. Even though it ranks dead last on customer rental quantity, this is likely due to low availability as it is the 2nd highest performing category and the smallest percentage of the overall inventory, meaning it would be worth looking into increasing the overall inventory of music related films.

	category character varying (25)	rental_qty bigint	inventory_qty bigint	performance numeric
1	Documentary	1044	294	3.55
2	Music	819	232	3.53
3	Travel	827	235	3.52
4	Drama	1053	300	3.51
5	Action	1095	312	3.51
6	Sci-Fi	1093	312	3.50
7	Family	1083	310	3.49
8	Children	939	269	3.49
9	Games	955	276	3.46
10	Comedy	932	269	3.46
11	Classics	930	270	3.44
12	Animation	1145	335	3.42
13	Foreign	1022	300	3.41
14	Sports	1164	344	3.38
15	New	927	275	3.37
16	Horror	834	248	3.36

Procedures And Queries

Analysis of the DVD Rental Database begins with adding the ‘tablefunc’ module for PostgreSQL databases to have access to the crosstab feature, allowing the use of pivot tables.

```
CREATE EXTENSION IF NOT EXISTS tablefunc;
```

Next create the pivot table.

```
CREATE OR REPLACE FUNCTION create_customer_rental_history() RETURNS void AS $$
DECLARE
    column_headers text;
    crosstab_query text;
BEGIN
    EXECUTE 'DROP TABLE IF EXISTS customer_rental_history';

    SELECT string_agg('"' || date || '" int', ', ') INTO column_headers
    FROM (SELECT DISTINCT to_char(rental_date, 'YYYY-MM') AS date FROM rental) sub;

    crosstab_query := 'CREATE TABLE customer_rental_history AS
        SELECT *
        FROM crosstab(
            $crosstab$
            SELECT
                c.customer_id AS customer_id,
                c.first_name || '' '' || c.last_name AS customer,
                to_char(r.rental_date, ''YYYY-MM'') AS date,
                COUNT(*) AS rentals
            FROM rental AS r
            LEFT JOIN customer AS c ON r.customer_id = c.customer_id
            GROUP BY c.customer_id, customer, date
            ORDER BY customer, date
            $crosstab$,
            $crosstab$
            SELECT DISTINCT to_char(rental_date, ''YYYY-MM'') AS date
            FROM rental
            ORDER BY date
            $crosstab$
        ) AS ct (
            "CustomerID" int,
            "Customer" text, ' || column_headers || ');';

    EXECUTE crosstab_query;
END $$ LANGUAGE plpgsql;
```

The query pictured above creates a pivot table from a dataset consisting of customer rental data aggregated by customer and month of activity. First column is the customers ID followed by their name, concatenated for ease of reading, and 1 column for each month in wich there is rental data with the customers rental count for that month. Now while seeing each individual customers rental history is a useful starting point for a lot of reports and does simplify reviewing account history on a customer by customer basis, this table can be aggregated further to provide a month over month report for total number of rentals per month, and if this report were to be run every year it could be used to provide important data on the habits of individual customers over time.

However if you are only interested in the total number of rentals month over month this can be achieved from the base table “public.rental” with the following query.

```sql
SELECT
    COUNT(*),
to_char(rental.rental_date, 'YYYY-MM') AS date
FROM rental
GROUP BY date
ORDER BY date DESC
```

The basis of the pivot table was made after aggregating customer data into a new table called customer_rentals_by_month wich was created with the following query:

```sql
CREATE TABLE customer_rentals_by_month AS
SELECT
    c.first_name || ' ' || c.last_name AS customer,
    to_char(r.rental_date, 'YYYY-MM') AS date,
    COUNT(*) AS rentals
FROM rental AS r
LEFT JOIN customer AS c ON r.customer_id = c.customer_id
GROUP BY customer, date
ORDER BY customer DESC;

ALTER TABLE customer_rentals_by_month
ADD CONSTRAINT unique_customer_date UNIQUE (customer, date);
```

This table has a stored procedure to update it over time although it is ultimately non-essential to the pivoted table.

```
CREATE OR REPLACE PROCEDURE update_customer_rentals_by_month()
LANGUAGE plpgsql
AS $$
BEGIN
    -- Delete existing data
    TRUNCATE TABLE customer_rentals_by_month;

    -- Insert updated data
    INSERT INTO customer_rentals_by_month
    SELECT
        c.first_name || ' ' || c.last_name AS customer,
        to_char(r.rental_date, 'YYYY-MM') AS date,
        COUNT(*) AS rentals
    FROM rental AS r
    LEFT JOIN customer AS c ON r.customer_id = c.customer_id
    GROUP BY customer, date
    ORDER BY customer DESC;

    -- Additional operations like logging can be added here
    RAISE NOTICE 'Update completed successfully.';
END;
$$;
```

The customer_rentals_by_month table is intended to provide long term year over year tend analysis of customer rentals on a per customer basis wich would be most useful for creating some kind of loyalty rewards points system based on number of rentals over time.

For the category breakdowns listed in the report summary there are a few queries that will be useful.

Rental Count By Category:

```
SELECT
    category.name AS category,
    COUNT(*) AS qty
FROM rental
    LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
    LEFT JOIN film ON inventory.film_id = film.film_id
    LEFT JOIN film_category ON film.film_id = film_category.film_id
    LEFT JOIN category ON film_category.category_id = category.category_id
GROUP BY category
ORDER BY qty DESC
```

Performance by category:

```sql
SELECT
    c.name AS category_name,
    COUNT(i.inventory_id) AS category_count,
    ROUND((COUNT(i.inventory_id) * 100.0 / total.total_inventory), 2) AS percentage_of_total_inventory
FROM
    inventory i
    JOIN film_category fc ON i.film_id = fc.film_id
    JOIN category c ON fc.category_id = c.category_id,
    (
        SELECT
            COUNT(inventory_id) AS total_inventory
        FROM
            inventory
    ) AS total
GROUP BY
    c.name,
    total.total_inventory
ORDER BY
    percentage_of_total_inventory DESC
```

And the 2 of those combined is what I would use to interpret categorical performance in the future:

```sql
WITH
    inventory_count AS (
        SELECT
            c.name AS category_name,
            COUNT(i.inventory_id) AS category_count,
            ROUND(
                (
                    COUNT(i.inventory_id) * 100.0 / total.total_inventory
                ),
                2
            ) AS percentage_of_total_inventory
        FROM
            inventory i
            JOIN film_category fc ON i.film_id = fc.film_id
            JOIN category c ON fc.category_id = c.category_id,
            (
                SELECT
                    COUNT(inventory_id) AS total_inventory
                FROM
                    inventory
            ) AS total
        GROUP BY
            c.name,
            total.total_inventory
        ORDER BY
            percentage_of_total_inventory DESC
    ),
    rental_count AS (
        SELECT
            category.name AS category,
            COUNT(*) AS qty
        FROM
            rental
            LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
            LEFT JOIN film ON inventory.film_id = film.film_id
            LEFT JOIN film_category ON film.film_id = film_category.film_id
            LEFT JOIN category ON film_category.category_id = category.category_id
        GROUP BY
            category
        ORDER BY
            qty DESC
    )
SELECT
    rc.category,
    rc.qty AS rental_qty,
    ic.category_count AS inventory_qty,
    ROUND(CAST(rc.qty AS NUMERIC) / ic.category_count, 2) AS performance
FROM
    rental_count AS rc
    LEFT JOIN inventory_count AS ic ON rc.category = ic.category_name
ORDER BY
    performance DESC
```

Financial Performance By Employee:

```
SELECT
    staff.store_id AS store,
    staff.first_name || ' ' || staff.last_name AS staff_name,
    SUM(payment.amount) AS total
    FROM payment
LEFT JOIN staff ON payment.staff_id = staff.staff_id
    LEFT JOIN rental ON payment.rental_id = rental.rental_id
GROUP BY store, staff_name
```

Count of rentals by film:

```
SELECT
    inventory.film_id,
    COUNT(rental.inventory_id),
    film.title,
    film.release_year
FROM inventory
    LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
    LEFT JOIN film ON inventory.film_id = film.film_id
GROUP BY inventory.film_id, film.title, film.release_year
ORDER BY count DESC
```

Note: there appears to be a major incongruity in the database in that every film in the film table has a release year of 2006, even the films that were rented in 2005. Also the payment dates for rentals in 2005 are all registered in 2007 wich again raises some concerns about the integrity of the data. I had intended to do a more in-depth financial breakdown for 2005, but as the transaction data dates do not line up with rental dates that was not possible to accurately compose.

Query Appendix

—Count of rentals by film

```
SELECT
	inventory.film_id,
	COUNT(rental.inventory_id),
	film.title,
	film.release_year
FROM inventory
	LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
	LEFT JOIN film ON inventory.film_id = film.film_id
GROUP BY inventory.film_id, film.title, film.release_year
ORDER BY count DESC
```

—Inventory Performance By Category

```
SELECT
	c.name AS category_name,
	COUNT(i.inventory_id) AS category_count,
	ROUND((COUNT(i.inventory_id) * 100.0 / total.total_inventory), 2) AS
percentage_of_total_inventory
FROM
	inventory i
	JOIN film_category fc ON i.film_id = fc.film_id
	JOIN category c ON fc.category_id = c.category_id,
	(
		SELECT
			COUNT(inventory_id) AS total_inventory
		FROM
			inventory
	) AS total
GROUP BY
	c.name,
	total.total_inventory
ORDER BY
	percentage_of_total_inventory DESC
```

```
--Count rentals by month
SELECT
        to_char(r.rental_date, 'YYYY-MM') AS date,
        COUNT(*) AS rentals
FROM rental AS r
GROUP BY date
ORDER BY date DESC
```

```
--Count rentals by category
SELECT
        category.name AS category,
        COUNT(*) AS qty
FROM rental
        LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
        LEFT JOIN film ON inventory.film_id = film.film_id
        LEFT JOIN film_category ON film.film_id = film_category.film_id
        LEFT JOIN category ON film_category.category_id = category.category_id
GROUP BY category
ORDER BY qty DESC
```

```
--Count rentals by film
SELECT
        inventory.film_id,
        COUNT(rental.inventory_id),
        film.title
FROM inventory
        LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
        LEFT JOIN film ON inventory.film_id = film.film_id
GROUP BY inventory.film_id, film.title
ORDER BY count DESC
```

```
--Revenue per employee
SELECT
        staff.store_id AS store,
        staff.first_name || ' ' || staff.last_name AS staff_name,
        SUM(payment.amount) AS total
        FROM payment
LEFT JOIN staff ON payment.staff_id = staff.staff_id
        LEFT JOIN rental ON payment.rental_id = rental.rental_id
GROUP BY store, staff_name
```

—Create Category/Inventory Performance Table

*note: not really needed since the procedure already does this but its not a bad idea to create the table first or to have a manual backup for reference in case someone edits the procedure incorrectly.

```
CREATE TABLE inventory_performance AS
WITH
    inventory_count AS (
        SELECT
            c.name AS category_name,
            COUNT(i.inventory_id) AS category_count,
            ROUND(
                (COUNT(i.inventory_id) * 100.0 / total.total_inventory), 2
            ) AS percentage_of_total_inventory
        FROM
            inventory i
            JOIN film_category fc ON i.film_id = fc.film_id
            JOIN category c ON fc.category_id = c.category_id,
            (
                SELECT
                    COUNT(inventory_id) AS total_inventory
                FROM
                    inventory
            ) AS total
        GROUP BY
            c.name,
            total.total_inventory
        ORDER BY
            percentage_of_total_inventory DESC
    ),
    rental_count AS (
        SELECT
            category.name AS category,
            COUNT(*) AS qty
        FROM
            rental
            LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
            LEFT JOIN film ON inventory.film_id = film.film_id
            LEFT JOIN film_category ON film.film_id = film_category.film_id
            LEFT JOIN category ON film_category.category_id = category.category_id
        GROUP BY
            category
        ORDER BY
            qty DESC)
SELECT
    rc.category,
    rc.qty AS rental_qty,
    ic.category_count AS inventory_qty,
    ROUND(CAST(rc.qty AS NUMERIC) / ic.category_count, 2) AS performance
FROM
    rental_count AS rc
    LEFT JOIN inventory_count AS ic ON rc.category = ic.category_name
ORDER BY performance DESC;
```

```
— Create update_inventory_performance Stored Procedure
CREATE OR REPLACE PROCEDURE update_inventory_performance()
LANGUAGE plpgsql
AS $$
BEGIN
    EXECUTE 'DROP TABLE IF EXISTS inventory_performance';
    EXECUTE '
    CREATE TABLE inventory_performance AS
    WITH
        inventory_count AS (
            SELECT
                c.name AS category_name,
                COUNT(i.inventory_id) AS category_count,
                ROUND((COUNT(i.inventory_id) * 100.0 / total.total_inventory), 2
) AS percentage_of_total_inventory
            FROM
                inventory i
                JOIN film_category fc ON i.film_id = fc.film_id
                JOIN category c ON fc.category_id = c.category_id,
                ( SELECT
                        COUNT(inventory_id) AS total_inventory
                    FROM
                        inventory
                ) AS total
            GROUP BY
                c.name,
                total.total_inventory
            ORDER BY
                percentage_of_total_inventory DESC ),
        rental_count AS (
            SELECT
                category.name AS category,
                COUNT(*) AS qty
            FROM
                rental
                LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
                LEFT JOIN film ON inventory.film_id = film.film_id
                LEFT JOIN film_category ON film.film_id = film_category.film_id
                LEFT JOIN category ON film_category.category_id = category.category_id
            GROUP BY
                category
            ORDER BY
                qty DESC)
    SELECT
        rc.category,
        rc.qty AS rental_qty,
        ic.category_count AS inventory_qty,
        ROUND(CAST(rc.qty AS NUMERIC) / ic.category_count, 2) AS performance
    FROM
        rental_count AS rc
        LEFT JOIN inventory_count AS ic ON rc.category = ic.category_name
    ORDER BY
        performance DESC;
    ';
END;
$$;
```

—Create customer_rentals_by_month table

* *Note: this was not needed by anything in the end and I only left it here because it was already done, and to serve as a backup example of stored procedures that is simpler than the inventory_performance.*

```
CREATE TABLE customer_rentals_by_month AS
SELECT
        c.first_name || ' ' || c.last_name AS customer,
        to_char(r.rental_date, 'YYYY-MM') AS date,
        COUNT(*) AS rentals
FROM rental AS r
LEFT JOIN customer AS c ON r.customer_id = c.customer_id
GROUP BY customer, date
ORDER BY customer DESC;

ALTER TABLE customer_rentals_by_month
ADD CONSTRAINT unique_customer_date UNIQUE (customer, date);
```

—Update customer rentals by month - Stored Procedure

```
CREATE OR REPLACE PROCEDURE update_customer_rentals_by_month(new_customer_id int)
LANGUAGE plpgsql
AS $$
BEGIN
   INSERT INTO customer_rentals_by_month (customer, date, rentals)
   SELECT
      c.first_name || ' ' || c.last_name AS customer,
      to_char(NOW(), 'YYYY-MM') AS date, -- Assume we're updating for the current month
      COUNT(*) AS rentals
   FROM rental AS r
   LEFT JOIN customer AS c ON r.customer_id = c.customer_id
   WHERE c.customer_id = new_customer_id
   GROUP BY customer, date
   ON CONFLICT (customer, date) DO UPDATE
   SET rentals = EXCLUDED.rentals;

   RAISE NOTICE 'Update completed successfully.';
END;
$$;
```

—Add in crosstab functionality

```
CREATE EXTENSION IF NOT EXISTS tablefunc;
```

— Add in pg_cron for task scheduling on Linux and MacOS

* *Note: pg_cron works on Mac an linux but I was unable to make it work in windows, so instead I added the procedure to the trigger function that executes on insert into the rental table for this procedure.*

```
CREATE EXTENSION IF NOT EXISTS pg_cron;

SELECT cron.schedule('daily_update', '0 0 * * *', $$
BEGIN
   PERFORM update_customer_rentals_by_month();
END $$);
```

—Create scheduled task on Windows
1. Create a file called psql_scheduled_tasks.sql and add the following content to that file:
CALL update_inventory_performance();
CALL update_customer_rentals_by_month();
2. Create a batch file called execute_stored_procedures.bat with the following content:
@echo off
psql -U your_username -d your_database -f “path\to\update_inventory_performance.sql"
3. Open Windows Task Scheduler, create a new task to run at midnight evey dat, and set the “Action” to start the batch file you just created.

— Create function to create customer_rental_history crosstab/pivot table
```
CREATE OR REPLACE FUNCTION create_customer_rental_history() RETURNS void AS $$
DECLARE
    column_headers text;
    crosstab_query text;
BEGIN
    EXECUTE 'DROP TABLE IF EXISTS customer_rental_history';

    SELECT string_agg('"' || date || '" int', ', ') INTO column_headers
    FROM (SELECT DISTINCT to_char(rental_date, 'YYYY-MM') AS date FROM rental) sub;

    crosstab_query := 'CREATE TABLE customer_rental_history AS
        SELECT *
        FROM crosstab(
            $crosstab$
            SELECT
                c.customer_id AS customer_id,
                c.first_name || '' '' || c.last_name AS customer,
                to_char(r.rental_date, ''YYYY-MM'') AS date,
                COUNT(*) AS rentals
            FROM rental AS r
            LEFT JOIN customer AS c ON r.customer_id = c.customer_id
            GROUP BY c.customer_id, customer, date
            ORDER BY customer, date
            $crosstab$,
            $crosstab$
            SELECT DISTINCT to_char(rental_date, ''YYYY-MM'') AS date
            FROM rental
            ORDER BY date
            $crosstab$
        ) AS ct (
            "CustomerID" int,
            "Customer" text, ' || column_headers || ');';

    EXECUTE crosstab_query;
END $$ LANGUAGE plpgsql;
```

—Execute function to create table
```
SELECT create_customer_rental_history()
```

— Select data from newly created table
```
SELECT * FROM customer_rental_history;
```

—Create trigger function to call stored procedure and recreate the customer_rental_history

```
CREATE OR REPLACE FUNCTION call_update_customer_history() RETURNS trigger AS $$
BEGIN

    CALL update_customer_rentals_by_month(NEW.customer_id);

    PERFORM create_customer_history();

    RETURN NEW;
END $$ LANGUAGE plpgsql;
```

—Create event trigger to call trigger function on insert into rental table

```
DROP TRIGGER IF EXISTS update_customer_history_after_insert ON rental;

CREATE TRIGGER update_customer_history_after_insert
AFTER INSERT ON rental
FOR EACH ROW
EXECUTE FUNCTION call_update_customer_history();
```

— Insert new data into rental table to test if update functions work:

```
INSERT INTO rental (customer_id, rental_date, inventory_id, staff_id) VALUES (1, '2024-05-01', 1,
1);
```

Requirements Breakdown

**Note: Emphasis is placed on the Category Performance aspect of the report to meet grading requirements.*

A. Real World Written Business Report: General Insights on inventory performance and customer rental history over time.

A1. category.name will be in the final summarized table of inventory_performance as well as the temp tables used to create that table.

A2. Data types of these fields are category - varchar(25), rentaly_qty - bigint, inventory_qty - bigint, and performance - numeric

A3. The tables used in inventory_performance are: inventory, film_category, category, rental, and film.

A4. Field that required specific manipulation was rc.qtywich is field from the temp table rental_count in the query to create the report. It was bigint but needed to be cast to numeric so I could get a non-integer value when diving rc.qty by ic.category (comes from temp table inventory_count). Since if you divide 2 integers psql will only output an integer and I wanted to display a float/decimal. If I hadn't not done this every category would have had the same performance of 3 or 4 when that was not technically accurate.

A5. This report can be used to determine what rental categories are performing the best/worst and could be used to decide wich categories in the overall inventory should be expanded or refreshed the most. Also can be used to identify if any part of the inventory has lower performance than others indicating areas that would benefit the most from focussed improvement efforts.

A6. The table is set to be recreated every night at midnight using Windows Task Scheduler if your on windows, or pg_cron if your on Linux or MacOS

Note on section B: There are not permanent detailed view tables as they truly served no purpose other than to de-normalize the data wich is why I decided to create them solely as temp tables within the query to create the simplified view. If for any reason the detailed view of either of those tables is needed it is very easy to extract those queries and run them separately. This is a technique I use on a

regular basis in my work to not clutter databases with duplicate de-normalized data.

All queries used to create this report and these tables, including the stored procedures and its triggers are located in the following GitHub repository:

https://github.com/cobalt88/DVD-Rental-EOY-2005