This report's purpose is to address the question of "what film category is rented from most at our stores".

1. In the report, there is an analysis of the number of times each specific item of inventory was rented so that the data could be joined to the specific films the inventory refers to, and then joined with the film categories those films are in. The amount each item of inventory was rented is analyzed by counting the amount each specific item of inventory appeared in the rental table. Each inventory id number is linked to the film id number it belonged to so the films that were rented can be identified and the amount each film was rented could be totaled. The film id number is linked to the category id number so the amount each film was rented could be accumulated and counted alongside other films in their respective categories, making it possible to identify the amount rented in each film category.

2. The data tables used for the report include the category table, the rental table, the inventory table, and the film_category table.

    - **The category table** provides the names of every film category.
    - **The rental table** allows for an analysis of every time a specific item of inventory was rented.
    - **The inventory table** allows the inventory id numbers to be joined with the film id number so the films are identified and each time a specific film was rented could be totaled.
    - **The film_category table** allows for each film id number to be joined with the film_category id number and allows for a count of the total amount rented in that specific film category.

3. In the detailed report, the fields being used include the category_id, category_name, amount_film, amount_rented.
    - **Category_id** is the specific id number assigned to each film category.
    - **Category_name** is the name of each film category.
    - **Amount_film** is the number of specific films that are in each film category.
    - **Amount_rented** is the amount of inventory rented in each film category.

    In the summary report, the fields being used are narrowed to the **category_id**, **category_name**, and **amount_rented** fields.

4. The category_name field in the detailed section will require a custom transformation to better represent each category. The category_name field contains the film category names but doesn't specify that each category refers to films. In order to better represent each film category, the category_name field will be transformed to include the word 'Films' at the end of each name.

5. The different business uses of this report that are represented by the detailed and summary sections include decision making on product placement, product advertising, and product supply. Knowing the film category that is rented from the most will give insight into where these products should be placed, for example, a film category that is popular can be placed in the back of the store which will require customers to be introduced to other films before reaching the film they want. Conversely, the product can be placed in the front of the store to make these films more accessible to the customers. The report also aids in the decision of which films should be advertised on posters in front of each store. The report also gives insight into what films should be purchased for our stores to optimize sales.

6. The report should be refreshed monthly in order to stay relevant to stakeholders. This will allow each store to order new films that are in a popular film category for that coming month. This will also allow stores to accurately target customers with advertisements on new films that are out in that specific film category.

Detailed Table

category_id [PK] integer	category_name character varying (25)	amount_film integer	amount_rented integer
1	Action Films	61	1112
2	Animation Films	64	1166
3	Children Films	58	945
4	Classics Films	54	939
5	Comedy Films	56	941
6	Documentary Films	63	1050
7	Drama Films	61	1060
8	Family Films	67	1096
9	Foreign Films	67	1033
10	Games Films	58	969
11	Horror Films	53	846
12	Music Films	51	830
13	New Films	60	940
14	Sci-Fi Films	59	1101
15	Sports Films	73	1179
16	Travel Films	53	837

Summary Table

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

Summary: The film category that was rented from the most was the sports films category with 1179 films rented. With this data in mind, we are able to make better decisions on the product placement, product advertisement, and product supply in our stores. Since more sports films are rented than any other category, we have the decision to either move the film category to the front of the store to make these films more accessible or to the back of the store in hopes of increased sales of other film categories as customers walk through our stores. We should also advertise more sports films on posters in front of our stores to draw our customers in. Finally, since we have a lot of sports fans, we should order more sports films to increase sales.

B.

```
-- Section B starts here

CREATE TABLE detailed_report (
category_id INT PRIMARY KEY,
category_name varchar (25),
amount_film INT,
amount_rented INT,
FOREIGN KEY (category_id) REFERENCES category (category_id));

CREATE TABLE summary_table(
category_id INT PRIMARY KEY,
category_name VARCHAR (25),
```

```
CREATE TABLE summary_table(
category_id INT PRIMARY KEY,
category_name VARCHAR (25),
amount_rented INT,
FOREIGN KEY (category_id) REFERENCES category (category_id));
```

C.

```
-- Section C starts here

INSERT INTO detailed_report (category_id, category_name, amount_film, amoun
 SELECT f.category_id as category_id, c.name AS category_name, COUNT(DISTIN
    FROM film_category AS f
    INNER JOIN category as c
    ON c.category_id = f.category_id
    INNER JOIN inventory AS i
    ON i.film_id = f.film_id
    INNER JOIN  rental AS r
    ON r.inventory_id = i.inventory_id
    GROUP BY f.category_id,c.name;
```

```sql
SELECT *
FROM detailed_report;

-- Any missing values?

SELECT *
FROM film_category
WHERE category_id IS NULL;

SELECT *
FROM inventory
WHERE film_id IS NULL;
```

```sql
-- Any missing values?

SELECT *
FROM film_category
WHERE category_id IS NULL;

SELECT *
FROM inventory
WHERE film_id IS NULL;

SELECT *
FROM rental
WHERE inventory_id IS NULL;
```

```sql
-- Any duplicate values?

SELECT film_id, category_id, COUNT (*)
FROM film_category
GROUP BY film_id, category_id
HAVING COUNT(*) > 1;

SELECT inventory_id, film_id, COUNT (*)
FROM inventory
GROUP BY inventory_id, film_id
HAVING COUNT (*) > 1;
```

```sql
SELECT rental_id, inventory_id, count(*)
FROM rental
GROUP BY rental_id, inventory_id
HAVING COUNT(*) > 1;

SELECT inventory_id, rental_date, return_date, COUNT (*)
FROM rental
GROUP BY inventory_id, rental_date, return_date
HAVING COUNT(*) > 1;

--section D starts here
```

D.

```sql
--section D starts here

CREATE FUNCTION update_detailed_report()
RETURNS Table (category_id INT, category_name VARCHAR (25), amount_film INT
language plpgsql
as
$$
BEGIN
Update detailed_report as d
Set category_name= CONCAT (d.category_name, ' Films');
END;
$$
```

```sql
SELECT update_detailed_report ();

SELECT * FROM detailed_report;

-- section e starts here

CREATE FUNCTION update_summary_function()
RETURNS trigger
LANGUAGE plpgsql
AS
$$
BEGIN
```

E.

```sql
-- section e starts here

CREATE FUNCTION update_summary_function()
RETURNS trigger
LANGUAGE plpgsql
AS
$$
BEGIN
    TRUNCATE summary_table;
    INSERT INTO summary_table(
    category_id, category_name, amount_rented)
    SELECT category_id, category_name, amount_rented
    GROUP BY category_id
    ORDER BY amount_rented DESC;
    RETURN NEW;
End;$$

CREATE TRIGGER update_summary_trigger
AFTER INSERT
ON detailed_report
FOR EACH STATEMENT
EXECUTE FUNCTION update_summary_function();

INSERT INTO category (category_id, name)
VALUES (17, 'Test');
```

```sql
CREATE TRIGGER update_summary_trigger
AFTER INSERT
ON detailed_report
FOR EACH STATEMENT
EXECUTE FUNCTION update_summary_function();

INSERT INTO category (category_id, name)
VALUES (17, 'Test');

INSERT INTO detailed_report (category_id, category_name, amount_film, amoun
VALUES (17, 'Test', 1, 1);
```

```
INSERT INTO category (category_id, name)
VALUES (17, 'Test');

INSERT INTO detailed_report (category_id, category_name, amount_film, amour
VALUES (17, 'Test', 1, 1);

SELECT *
FROM summary_table;

-- section F starts here

--stored procedure should be executed monthly
```

F.

```
-- section F starts here

--stored procedure should be executed monthly
CREATE PROCEDURE update_detailed_summary ()
LANGUAGE plpgsql
AS $$
BEGIN
    TRUNCATE detailed_report;
    TRUNCATE summary_table;
    INSERT INTO detailed_report (category_id, category_name, amount_film, a
    SELECT f.category_id as category_id, c.name AS category_name, COUNT(DIS
    FROM film_category AS f
```

```
    INNER JOIN inventory AS i
    ON i.film_id = f.film_id
    INNER JOIN  rental AS r
    ON r.inventory_id = i.inventory_id
    GROUP BY f.category_id,c.name;

    INSERT INTO summary_table(
    category_id, category_name, amount_rented)
    SELECT category_id, category_name, amount_rented
    FROM detailed_report
    GROUP BY category_id
    ORDER BY amount_rented DESC;
    ORDER BY amount_rented DESC;

END;$$

CREATE FUNCTION update_detailed_summary_function()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
 CALL update_detailed_summary ();
RETURN NEW;
END;$$
```

The stored procedure can be run on a monthly schedule by using a job scheduling tool such as Linux crontab, Agent pgAgent, or extension pg_cron which are available for PostgreSQL. In order to use these tools, they have to be installed first on the server. If we are using pgAgent or pg_cron we then need to create an extension for that tool. We are able to manage the pgAgent tool using the pgAdmin interface, and we are able to manage the pg_cron tool using SQL.