Cheat Sheet for Lesson 4B: Queries and Subqueries

Subqueries are powerful SQL features that allow you to perform complex queries by nesting one query within another. Understanding the various types of subqueries and their components can significantly enhance your data querying capabilities.

Key Concepts:

- **`WHERE` Clause:** Used to filter records based on specified conditions.
- **`HAVING` Clause:** Similar to the `WHERE` clause but used for filtering groups of records in combination with the `GROUP BY` clause.
- **`JOIN` Operations:** Combine rows from two or more tables based on a related column between them.
- Aggregation Functions (`COUNT`, `SUM`, `AVG`, etc.): Perform calculations on sets of data to return single values.
- **`GROUP BY` Clause:** Groups rows sharing a property so that aggregate functions can be applied to each group.
- **`ORDER BY` Clause:** Specifies the order in which to return the rows from a query.
- **`LIMIT` Clause:** Restricts the number of rows returned by a query, useful for pagination.
- **`EXISTS`:** Logical operator used to test for the existence of any record in a subquery.
- **`IN`:** Operator that allows you to specify multiple values in a `WHERE` clause, often used with subqueries returning a list of values.
- **Common Table Expressions (CTEs):** Not subqueries per se, but CTEs allow for the creation of temporary result sets that can be easily referenced within complex queries. CTEs are defined with the `WITH` clause.

What is a CTE (e.g. `WITH`):
Common Table Expressions (CTEs) provide a way to write more readable and modular SQL queries. By using the `WITH` keyword, you can define a temporary result set that you can then reference within the main SQL query. CTEs are particularly useful for breaking down complex queries into simpler parts, improving query organization, and enhancing readability. They can be used for recursive queries, such as navigating hierarchical data structures, but are equally valuable for non-recursive tasks, like pre-aggregating data or simplifying multiple joins. CTEs make it easier to follow the logic of SQL operations by allowing you to name and thus document different parts of your query. CTEs offer a structured approach to query design, allowing for the temporary storage of result sets that can be easily referenced within a larger query. They are particularly handy when dealing with complex queries that benefit from being broken into more manageable parts. By defining these parts as CTEs, you enhance the readability and maintenance of your SQL code, making it easier for others (and your future self) to understand the query's purpose and structure. CTEs exemplify the principle of "divide and conquer" in SQL, enabling you to tackle sophisticated data retrieval tasks by simplifying the steps involved.

EX 1. Find Top 5 Customers by Total Spending

HINT: This example uses a CTE to calculate the total spending of each customer and then selects the top 5 spenders.

```
WITH CustomerSpending AS (
  SELECT p.customer_id, SUM(p.amount) AS total_spent
  FROM payment p
  GROUP BY p.customer_id
)
SELECT cs.customer_id, c.first_name || ' ' || c.last_name AS
customer_name, cs.total_spent
FROM CustomerSpending cs
JOIN customer c ON cs.customer_id = c.customer_id
ORDER BY cs.total_spent DESC
LIMIT 5;
```

EX 2. List Movies and Their Rental Counts in 'Action' Category

HINT: This example uses a CTE to identify all films in the 'Action' category, and counts how many times each was rented.

```
WITH ActionFilms AS (
  SELECT f.film_id, f.title
  FROM film f
  JOIN film_category fc ON f.film_id = fc.film_id
  JOIN category c ON fc.category_id = c.category_id
  WHERE c.name = 'Action'
)
SELECT af.title, COUNT(r.rental_id) AS rental_count
FROM ActionFilms af
JOIN inventory i ON af.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
GROUP BY af.title
ORDER BY rental_count DESC;
```

Types of Subqueries:

1. Scalar Subqueries: Return a single value (one row, one column) and can be used in the `SELECT`, `FROM`, `WHERE`, and `HAVING` clauses.

EX 1. Find Average Length of all Films

```
SELECT title, length FROM film
WHERE length > (SELECT AVG(length) FROM film);
```

EX 2. Get the maximum rental duration allowed for any film.

```
SELECT title, rental_duration FROM film
WHERE rental_duration = (SELECT MAX(rental_duration)
FROM film);
```

2. Column Subqueries: Return a single column but potentially multiple rows. Often used in the `WHERE` clause to compare a column value against a set of values returned by the subquery.

EX 1. Find customers who have rented films in the 'Action' category.

```
SELECT DISTINCT customer_id FROM rental
WHERE inventory_id IN (
  SELECT inventory_id FROM inventory
  WHERE film_id IN (
    SELECT film_id FROM film_category
    WHERE category_id = (
      SELECT category_id FROM category
      WHERE name = 'Action'
    )
  )
);
```

EX 2. List films that are more expensive than the average rental rate.

```
SELECT title FROM film
WHERE rental_rate > (
  SELECT AVG(rental_rate) FROM film
);
```

3. Row Subqueries: Return a single row but potentially multiple columns. Useful for comparisons involving more than one column.

EX 1. Find the film with a specific ID and its rental rate and

replacement cost.

```
SELECT title FROM film
WHERE (rental_rate, replacement_cost) = (
  SELECT rental_rate, replacement_cost FROM film WHERE
film_id = 1
);
```

EX 2. Identify films released on or after the earliest year in the dataset, and having a certain rating.

```
SELECT title
FROM film
WHERE release_year >= (SELECT MIN(release_year) FROM
film)
AND rating = 'PG';
```

4. Table Subqueries: Return an entire table (multiple rows and multiple columns). These are often used with the `IN` clause, `EXISTS` clause, or as a derived table in the `FROM` clause.

EX 1. Retrieve the top 5 most frequently rented films.

```
SELECT title FROM film
WHERE film_id IN (
  SELECT film_id FROM (
    SELECT inventory.film_id, COUNT(rental_id) AS
rental_count FROM rental
    JOIN inventory ON rental.inventory_id =
inventory.inventory_id
    GROUP BY inventory.film_id
    ORDER BY rental_count DESC
    LIMIT 5
  ) AS subquery
);
```

EX 2. Find customers who rented films in both 'Sci-Fi' and 'Action' categories.

```
SELECT customer_id FROM rental
WHERE inventory_id IN (
  SELECT inventory_id FROM inventory
  WHERE film_id IN (
    SELECT film_id FROM film_category
    WHERE category_id IN (
      SELECT category_id FROM category
      WHERE name IN ('Sci-Fi', 'Action')
    )
  )
)
GROUP BY customer_id
HAVING COUNT(DISTINCT inventory_id) > 1;
```

5. Correlated Subqueries: Refer to columns in the outer query, causing the subquery to be executed once for each row processed by the outer query.

EX 1. List customers and the count of their rentals that are yet to be returned.

```
SELECT customer_id,
(
  SELECT COUNT(*) FROM rental r2
  WHERE r2.customer_id = r.customer_id AND r2.return_date IS
NULL
) AS unreturned_rentals
FROM rental r
GROUP BY customer_id;
```

EX 2. Find films and their average rental rate compared to the overall average.

```
SELECT title, rental_rate,
(
  SELECT AVG(rental_rate) FROM film f2
) AS average_rental_rate
FROM film f
WHERE rental_rate > (
  SELECT AVG(rental_rate) FROM film
);
```