Lesson 4B: Queries and Sub-Queries

This lesson is crafted to navigate you through the intricacies of crafting precise queries and leveraging the power of sub-queries for advanced data analysis. With a focus on enhancing your ability to request and manipulate data, we examine the foundational and nuanced aspects of SQL, ensuring you gain a comprehensive understanding and hands-on experience.

Objectives:

1. **Understand Basic and Complex Queries:** Familiarize yourself with constructing SQL queries to retrieve data tailored to specific requirements, from simple data fetches to more complex, condition-based retrievals.
2. **Master Sub-Queries:** Learn how to embed queries within other queries to perform complex analyses, filter results, and manipulate data in ways that single-layer queries cannot achieve.
3. **Apply Sub-Queries Across Contexts:** Explore the use of sub-queries in different parts of a SQL statement, including in `SELECT` clauses, `FROM` clauses, `WHERE` clauses, and even in `HAVING` clauses for filtering aggregated data.
4. **Enhance Data Retrieval Precision:** Gain insights into how sub-queries can refine your data extraction process, allowing for more precise and relevant outcomes that support informed decision-making.
5. **Practical Application and Examples:** Through hands-on examples and guided exercises, apply your learning in real-world scenarios, reinforcing your understanding and proficiency in using queries and sub-queries effectively.

1. Defining Queries and Sub-Queries

A *query* is a request for data or information from a database. At its core, a query is written in SQL (Structured Query Language) and allows users to retrieve, insert, update, or delete data within the database tables. Queries are the primary means through which we interact with and extract data from databases. They can range from simple commands fetching specific columns from a single table to more complex instructions involving multiple tables, conditional logic, and data manipulation.

```
SELECT first_name, last_name FROM actor;
```

In this example, the query retrieves the `first_name` and `last_name` columns from the `actor` table.

A *sub-query*, also known as an inner query or nested query, is a query within another SQL query. Sub-queries allow for more complex data retrieval operations by enabling the result of one query to be used in the condition or component of another. Essentially, sub-queries can perform an additional layer of filtering, data aggregation, or manipulation that supports the outer query's goals.

```
SELECT title, release_year
FROM film
WHERE film_id IN (SELECT film_id FROM film_actor
WHERE actor_id = 1);
```

In this example, the sub-query selects all `film_id` values associated with an `actor_id` of 1. The outer query then uses these `film_id` values to fetch titles and release years of those specific films.

2. **Examples of Good and Bad Queries**

In this section, we'll explore examples of both well-constructed (good) and poorly constructed (bad) queries within the context of the `dvdrental` database. These examples aim to highlight best practices in SQL querying and common pitfalls to avoid, enhancing your ability to write efficient, clear, and effective SQL code.

Example 1:

```
SELECT title, release_year FROM film;
```

Why It's Good: This query explicitly requests only the columns needed (`title` and `release_year`), which is more efficient than requesting all columns, especially in tables with many fields.

Example 2:

```
SELECT film.title, actor.first_name, actor.last_name
FROM film_actor
INNER JOIN film ON film_actor.film_id = film.film_id
INNER JOIN actor ON film_actor.actor_id = actor.actor_id
WHERE film.title = 'Matrix, The';
```

Why It's Good: This query correctly joins the `film` and `actor` tables through the `film_actor` table to retrieve actor names for a specific film. It uses explicit JOIN conditions and a precise WHERE clause, making it clear and efficient.

Example 3:

```
SELECT * FROM film;
```

Why It's Bad: Using `SELECT *` retrieves all columns from the `film` table, which can be inefficient, especially if only a few columns are actually needed for the analysis or output.

Example 4:

```
SELECT film.title, actor.first_name, actor.last_name
FROM film, actor;
```

Why It's Bad: This query lacks a proper JOIN condition, leading to a Cartesian product between the `film` and `actor` tables. It results in a massive dataset where every film is matched with every actor, which is rarely useful and can be highly inefficient.

Example 5:

```
SELECT title FROM film
WHERE film_id IN (SELECT film_id FROM film WHERE length > 120);
```

Why It's Bad: The sub-query is unnecessary here since the filtering condition (`length > 120`) could be applied directly in the main query's WHERE clause, simplifying the query and improving readability.

Tips:

1. Good queries are clear, and concise, and ask only for the data needed. They efficiently leverage JOINs to combine tables in meaningful ways and apply conditions directly where possible.

2. Bad queries often involve unnecessary complexity, such as overusing `SELECT *`, lacking proper JOIN conditions or employing redundant sub-queries. These practices can lead to inefficient data retrieval and confusion.

3. **Writing Good Sub-Queries (Practice)**

Example 6:

Scenario: Find all films that have a higher replacement cost than the average replacement cost of all films.

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

Why It's Good: This sub-query efficiently calculates the average replacement cost of all films and then uses that value to filter films in the outer query. It's a straightforward example of using a sub-query to apply a dynamic condition based on aggregate data.

Example 7:

Scenario: List actors and the total number of films they've appeared in, ordering by the total number of films in descending order.

```
SELECT actor_info.first_name, actor_info.last_name, actor_info.total_films
FROM (
   SELECT actor.first_name, actor.last_name, COUNT(film_actor.film_id) AS total_films
   FROM actor
   JOIN film_actor ON actor.actor_id = film_actor.actor_id
   GROUP BY actor.first_name, actor.last_name
) AS actor_info
ORDER BY actor_info.total_films DESC;
```

Why It's Good: This sub-query (aliased as `actor_info`) aggregates the total number of films for each actor, and the outer query then orders these actors by their film count. It demonstrates how a sub-query can serve as a temporary table for further analysis or sorting in the outer query.

Example 8:

Scenario: Retrieve a list of customers and the title of the last film they rented.

```
SELECT customer.first_name, customer.last_name,
(
   SELECT film.title
   FROM rental
   INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
   INNER JOIN film ON inventory.film_id = film.film_id
   WHERE rental.customer_id = customer.customer_id
   ORDER BY rental.rental_date DESC
   LIMIT 1
) AS last_rented_film
FROM customer;
```

Why It's Good: This is a correlated sub-query because it references the `customer.customer_id` from the outer query. It fetches the title of the most recently rented film for each customer, demonstrating how sub-queries can efficiently pull related data based on each row's context from the outer query.

Key Terms:

1. **Sub-Query (Nested Query):** A query placed within another SQL query. Sub-queries allow for more complex data retrieval operations by using the result of one query as a condition or component in another. They can appear in various parts of the main query, including `SELECT`, `FROM`, `WHERE`, and `HAVING` clauses, enabling detailed and conditional data analysis.
2. **Relational Database:** A type of database that stores and provides access to data points that are related to one another. Relational databases are based on the relational model, an intuitive way to represent data in tables (relations), consisting of rows and columns. Each table represents a different entity type, and relationships between entities are defined through foreign keys.
3. **Abstraction in SQL:** The process of hiding the complexity of database operations from the user by using higher-level features such as views, stored procedures, and functions. Abstraction allows users to interact with the data without needing to know the intricate details of how data is stored or maintained, making queries simpler and boosting database security.
4. **JOIN Clause:** A clause used in SQL to combine rows from two or more tables, based on a related column between them. JOINs are fundamental in relational databases to query data from multiple tables simultaneously. Types of JOINs include INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN, each serving different data retrieval purposes.
5. **Aggregate Functions:** SQL functions that perform a calculation on a set of values and return a single value. Common aggregate functions include `AVG` (calculates the average of a set of values), `COUNT` (counts the number of values), `SUM` (adds up all values), `MIN` (finds the minimum value), and `MAX` (finds the maximum value). These functions are often used in conjunction with `GROUP BY` to summarize data.
6. **HAVING Clause:** An SQL clause used to filter records that result from a `GROUP BY` operation, based on a specified condition. Unlike the `WHERE` clause, which filters rows on individual record criteria before any groupings are made, the `HAVING` clause applies conditions to groups after data is aggregated. This makes the `HAVING` clause particularly useful for filtering aggregated data, allowing for more precise control over the results of queries that involve `GROUP BY`.

Practice (Multiple Choice):

Question 1 - What is a sub-query in SQL?

A) A query used to update data in a database
B) A query placed within another SQL query to perform complex data retrieval
C) A special type of JOIN operation
D) A query used to delete data based on specific conditions

Correct Answer: B) A query placed within another SQL query to perform complex data retrieval

Question 2 - Which clause is used in conjunction with aggregate functions to filter the results of a `GROUP BY` operation?

A) WHERE
B) ORDER BY
C) HAVING
D) LIMIT

Correct Answer: C) HAVING

Question 3 - In a relational database, what does a JOIN clause primarily facilitate?

A) Deleting rows from multiple tables simultaneously
B) Combining rows from two or more tables based on a related column between them
C) Summarizing data across multiple tables
D) Creating new tables based on existing ones

Correct Answer: B) Combining rows from two or more tables based on a related column between them

Question 4 - What is the purpose of using the `EXTRACT` function in SQL queries?

A) To remove unwanted rows from the result set
B) To calculate the sum of numerical values in a column
C) To extract specific parts from a date or timestamp, such as the year or month
D) To identify unique values in a dataset

Correct Answer: C) To extract specific parts from a date or timestamp, such as the year or month

Question 5 - Which of the following best describes the concept of "Abstraction in SQL"?

A) The process of simplifying SQL queries by removing unnecessary conditions
B) The technique of hiding the complexity of database operations from the user
C) A method of optimizing SQL queries for faster execution
D) The practice of manually managing database transactions for reliability

Correct Answer: B) The technique of hiding the complexity of database operations from the user