Lesson 2B Objectives:

As we dive into the intricacies of JOINS and UNIONS in SQL, this lesson aims to equip you with the knowledge and skills necessary to navigate and manipulate data across multiple tables effectively. By the end of Lesson 2B, you will achieve the following objectives:

1. **Understand the Concept and Application of JOINS:**
    a. Gain a comprehensive understanding of how JOINS operates within SQL to combine rows from two or more tables based on a related column between them.
    b. Explore the four main types of JOINS: INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN, understanding their differences and use cases.
2. **Master the Use of UNIONS:**
    a. Learn the purpose of the UNION operator in SQL and how it differs from UNION ALL.
    b. Understand when and why to use UNION to combine the result sets of multiple SELECT statements into a single, unified result set.
3. **Apply JOINS and UNIONS in Real-World Scenarios:**
    a. Through the context of a hospital database, apply JOINS to extract interconnected data across patient, doctor, and appointment tables.
    b. Utilize UNIONS to consolidate data from different sources, such as aggregating procedure lists from various departments.
4. **Develop Problem-Solving Skills with JOINS and UNIONS:**
    a. Engage in hands-on exercises and challenges that encourage critical thinking and problem-solving using JOINS and UNIONS.

    b. Tackle realistic scenarios that mimic the data analysis needs of a hospital administration, enhancing your ability to make data-driven decisions.
5. **Best Practices and Optimization Techniques:**
    a. Learn best practices for structuring queries with JOINS and UNIONS to improve readability and performance.
    b. Explore advanced techniques for optimizing queries involving JOINS and UNIONS, ensuring efficient data retrieval and analysis.

By achieving these objectives, you will be well-prepared to leverage the power of JOINS and UNIONS in your SQL queries, enhancing your ability to analyze and interpret complex datasets across various industries and sectors. Let's begin this learning journey with enthusiasm and a keen interest in uncovering the full potential of relational databases.

Lesson 2B
Using the context of a hospital database, we will explore practical examples to fully understand how these operations can be applied to real-world scenarios.

1) **Understanding JOINS**

JOINS are used to combine rows from two or more tables based on a related column between them. This operation is fundamental when working with relational databases, as data related to a single entity or transaction is often spread across tables.

Types of JOINS:
A. INNER JOIN - Retrieves records that have matching values in both tables. It's the most common type of join.
    Example: Finding all patients who have appointments. This would involve joining the `Patients` and `Appointments` tables on the patient ID.

B. LEFT JOIN (or LEFT OUTER JOIN) - Returns all records from the left table, and the matched records from the right table. If there is no match, the result is NULL on the right side.
    Example: Listing all patients, including those without appointments.

C. RIGHT JOIN (or RIGHT OUTER JOIN) - Similar to LEFT JOIN, but returns all records from the right table, and the matched records from the left table.
    Example: Identifying all doctors who have scheduled appointments.

D. FULL JOIN (or FULL OUTER JOIN) - Combines LEFT JOIN and RIGHT JOIN, returning records when there is a match in either the left or right table.
    Example: Creating a comprehensive list of all appointments, including patients without appointments and appointments without a registered patient.

2) **Understanding JOINS**

UNIONS are used to combine the result sets of two or more SELECT statements. Each SELECT statement within the UNION must have the same number of columns, and those columns must have similar data types.

A. **UNION:** Selects only distinct values by default, eliminating duplicate rows between the various SELECT statements.

    Example: Aggregating a list of all services provided by the hospital, including both medical procedures and consultations, without duplicates.

B. **UNION ALL:** Includes all duplicate rows, combining all results from the SELECT statements without filtering for uniqueness.

    Example: Compiling a complete log of patient visits and procedures, where duplicates indicate multiple visits or recurring procedures.

DVD Rental Database Examples:

Let's apply the concepts of JOINS and UNIONS using the DVD Rental database for some unique examples that mirror the complexity and types of queries you might encounter in a real-world scenario like a hospital database.

Example 1: Find All Actors Who Have Starred in a Comedy

Imagine you want to highlight actors who have starred in comedies for a special feature in your rental store. This requires a JOIN across the `film`, `film_actor`, and `category` tables.

```
SELECT DISTINCT a.first_name, a.last_name
FROM actor a
JOIN film_actor fa ON a.actor_id = fa.actor_id
JOIN film f ON fa.film_id = f.film_id
JOIN film_category fc ON f.film_id = fc.film_id
JOIN category c ON fc.category_id = c.category_id
WHERE c.name = 'Comedy';
```

This query uses multiple JOINS to navigate from actors to the films they've starred in and filters the results for comedies. By breaking down the query into these components, we can see how each JOIN operation builds upon the last to progressively filter and refine the data until we have a list of actors who have starred in comedies, demonstrating the power and flexibility of SQL JOIN operations in querying relational databases. Let's break this down a bit!

1. SELECT DISTINCT a.first_name, a.last_name:

- SELECT DISTINCT is used to select the columns `first_name` and `last_name` from the result set. The `DISTINCT` keyword ensures that duplicate rows are removed from the results, so each actor is listed only once, even if they have starred in multiple comedy films.
- a.first_name, a.last_name specifies which columns to retrieve, prefixed with `a`, which is an alias for the `actor` table defined in the `FROM` clause.

2. FROM actor a:

- FROM specifies the primary table from which to retrieve the data. In this case, it's the `actor` table.
- a is an alias for the `actor` table, allowing us to refer to it more succinctly in other parts of the query.

3. JOIN film_actor fa ON a.actor_id = fa.actor_id:

- This line performs an INNER JOIN between the `actor` table and the `film_actor` table (aliased as `fa`).

- The join condition, ON a.actor_id = fa.actor_id, specifies that the query should only include rows where the `actor_id` value in the `actor` table matches an `actor_id` value in the `film_actor` table. This effectively filters for actors who have appeared in at least one film.

4. JOIN film f ON fa.film_id = f.film_id:
- Another INNER JOIN is performed, this time between the `film_actor` table and the `film` table (aliased as `f`).
- The join condition, ON fa.film_id = f.film_id, matches films in the `film` table to their corresponding entries in the `film_actor` table, filtering for the specific films that the actors have appeared in.

5. JOIN film_category fc ON f.film_id = fc.film_id:
- This line joins the `film` table to the `film_category` table (aliased as `fc`), based on matching `film_id` values.
- This join links each film to its assigned categories, preparing to filter these by the comedy genre.

6. JOIN category c ON fc.category_id = c.category_id:
- Here, the `film_category` table is joined to the `category` table (aliased as `c`), using the `category_id`.
- This final join allows us to access the name of each category associated with the films.

7. WHERE c.name = 'Comedy';
- The WHERE clause filters the results to only include rows where the category name is 'Comedy'.
- This is how the query ensures that only actors who have starred in films categorized as comedies are included in the results.

Example 2: Listing Movies and Their Language

To assist customers who are looking for movies in a specific language, you need a list that shows each movie and its corresponding language.

```
SELECT f.title, l.name AS language
FROM film f
JOIN language l ON f.language_id = l.language_id
ORDER BY f.title;
```

This query performs an INNER JOIN between the `film` and `language` tables, providing a clear list for customer queries.

Example 3: Combining All Staff and Customer Names

For a community event, you want to create nametags for both staff and customers. To prepare, you need a comprehensive list of all names. The UNION ensures any staff who are also registered as customers are not listed twice, providing a unique list of names.

```
SELECT first_name, last_name FROM customer
UNION
SELECT first_name, last_name FROM staff;
```

Example 4: Aggregate Film Categories and Store Inventory Categories

Suppose you want to analyze the types of categories in your film catalog and compare them to the categories tagged in your physical store inventory for a consistency check.

```
SELECT name AS category FROM category
UNION
SELECT category AS category FROM inventory_categories;
```

This UNION combines category names from your film catalog with those from the store's inventory system, removing duplicates for a clean comparison.

Walk-Through Simulations
Simulation 1: List of Films and Their Actors

Retrieve a list that shows each film along with the actors who starred in it.

```
SELECT f.title, STRING_AGG(a.first_name || ' ' || a.last_name, ', ') AS actors
FROM film f
JOIN film_actor fa ON f.film_id = fa.film_id
JOIN actor a ON fa.actor_id = a.actor_id
GROUP BY f.title;
```

RETURNS:

Explanation: This query joins the `film`, `film_actor`, and `actor` tables to correlate films with their actors. It uses `STRING_AGG` to concatenate actor names per film, showing a comprehensive cast list for each title.

Simulation 2: Finding Customers Who Rented a Specific Film

Identify customers who rented the film "ACADEMY DINOSAUR."

```
SELECT DISTINCT c.first_name, c.last_name
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
WHERE f.title = 'ACADEMY DINOSAUR';
```

RETURNS:

Explanation: By joining the `customer`, `rental`, `inventory`, and `film` tables, this query pinpoints customers who have rented "ACADEMY DINOSAUR," ensuring no duplicated names with `DISTINCT`.

Simulation 3: Combine Comedy and Action Films in One List

Create a unified list of all Comedy and Action films, excluding duplicates.

```
SELECT 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 = 'Comedy'
UNION
SELECT 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';
```

RETURNS:

Explanation: This simulation leverages `UNION` to amalgamate two select statements: one fetching Comedy and the other Action films, filtering out any duplicates between the two categories.

Simulation 4: Actors Not Featured in Any Film

List all actors who, due to a database error, haven't been correctly linked to any film in the `film_actor` table.

```
SELECT a.first_name, a.last_name FROM actor a
LEFT JOIN film_actor fa ON a.actor_id = fa.actor_id
WHERE fa.film_id IS NULL;
```

RETURNS:

Explanation: Utilizing a `LEFT JOIN` and filtering where `film_id IS NULL` in the `film_actor` table, this query finds actors without any film associations, highlighting potential data integrity issues.

Simulation 5: Number of Films by Language

Count how many films there are for each language in the inventory.

```
SELECT l.name AS language, COUNT(f.film_id) AS number_of_films
FROM film f
JOIN language l ON f.language_id = l.language_id
```

```
GROUP BY l.name;
```

RETURNS:

Explanation: This query joins the `film` and `language` tables to group films by their language, using `COUNT` to tally the number of films for each language, providing insights into the linguistic diversity of the film inventory.

Key Terms

1. JOIN: A SQL operation used to combine rows from two or more tables based on a related column between them. It's instrumental in querying relational databases where data is distributed across multiple tables.

2. INNER JOIN: The most common type of JOIN, which returns rows when there is at least one match in both tables being joined. If there is no match, the rows are not returned.

3. LEFT JOIN (or LEFT OUTER JOIN): Returns all rows from the left table, and the matched rows from the right table. For rows in the left table with no match in the right table, the result set will contain NULL values for the columns from the right table.

4. RIGHT JOIN (or RIGHT OUTER JOIN): Opposite to LEFT JOIN, it returns all rows from the right table and the matched rows from the left table. Rows in the right table that do not have a match in the left table will have NULL values for the columns from the left table in the result set.

5. FULL JOIN (or FULL OUTER JOIN): Combines the results of both LEFT JOIN and RIGHT JOIN. It returns rows when there is a match in one of the tables. It effectively includes all rows from both tables, filling in NULL values for matches that do not exist in the opposing table.

6. UNION: A SQL operation used to combine the result sets of two or more SELECT statements. It removes duplicate rows between the various SELECT statements.

7. UNION ALL: Similar to UNION, but it does not remove duplicate rows. It's used when you want to include all rows from the result sets of the SELECT statements, including duplicates.

8. DISTINCT: Often used with SELECT statements to remove duplicate rows from the result set, ensuring that only unique rows are returned.

9. Aggregate Functions: Functions used to perform calculations on a set of values, returning a single value. Common aggregate functions include `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`. They are often used in conjunction with GROUP BY.

10. GROUP BY: A clause used in conjunction with aggregate functions to group the result set by one or more columns. It's essential for performing aggregate calculations on subsets of data.

11. Aliases: Temporary names given to tables or columns for the duration of a SQL query. Aliases help simplify query syntax and improve readability, especially when dealing with multiple joins or complex queries.

FAQs:

Throughout our exploration of JOINS and UNIONS, several common questions arise. Let's address these queries to clarify any confusion and deepen our understanding of these powerful SQL features.

1. Why do you put an "f" at the end of some of the statements? Or a "fa"? What's up with that?

The "f", "fa", or similar letters you see in SQL statements are known as aliases. An alias is a temporary name assigned to a table or column for the duration of a query. **In the context of our examples:**

- "f" often stands for a table named `film`, serving as a shorthand that simplifies referencing the table in the query.
- "fa" might represent the `film_actor` table, providing a concise way to refer to it.

Aliases help make SQL queries more readable, especially when dealing with multiple tables or when tables are joined.

2. What does "fc" stand for? What is "c."? Like what's with all the extra letters?

Similarly, "fc" and "c" are also aliases used to simplify query syntax:
- "fc" might be used as an alias for the `film_category` table.
- "c" could represent the `category` table.

Using aliases like "fc" and "c" reduces the complexity of SQL statements, making them cleaner and easier to follow, especially in queries involving multiple JOINS.

3. Why are there so many different types of JOIN functions?

SQL provides various types of JOINS (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN) to cater to different data retrieval needs. Each type of JOIN serves a specific purpose based on how you need to combine data from multiple tables:
- INNER JOIN returns rows with matching values in both tables.
- LEFT JOIN and RIGHT JOIN include all rows from one table plus any matching rows in the other.
- FULL JOIN combines the effects of both LEFT JOIN and RIGHT JOIN.

The variety ensures flexibility in handling diverse data relationships, allowing you to tailor your queries to achieve the exact results you need.

4. Do we have to do all of this to get the data from different tables?

While it might seem complex at first, using JOINS and UNIONS is necessary for retrieving and combining data from multiple tables in a relational database. These operations are fundamental to SQL and enable you to perform comprehensive data analysis, report generation, and more. With practice, these concepts will become second nature, enhancing your ability to work efficiently with databases.

5. Is it possible to get all of the data from the whole database?

Yes, it's technically possible to retrieve all data from the entire database by executing SELECT statements on each table. However, doing so indiscriminately is generally not practical or advisable due to potential performance issues and information overload. It's more efficient to use targeted queries with specific JOINs or UNIONs to access the precise data you need for analysis or decision-making.

Review Questions
Test your understanding of the concepts covered in Lesson 2B with these multiple-choice review questions.

Question 1: What does an INNER JOIN return?

A) All rows from both tables, regardless of matches
B) Only rows that have matching values in both joined tables
C) All rows from the left table and matched rows from the right table

D) A combination of all rows from both tables where there is no match

Question 2: Which statement about LEFT JOIN is true?
A) It returns only the rows from the left table that have a match in the right table
B) It returns all rows from the right table, and the matched rows from the left table
C) It returns all rows from the left table, and the matched rows from the right table
D) It excludes all rows from the left table that have a match in the right table

Question 3: When would you use a FULL JOIN?
A) When you need all records when there is a match in either the left or right table
B) When you only want records that have matching values in both tables
C) When you want all records from the left table only
D) When you want to exclude all matching records

Question 4: What is the difference between UNION and UNION ALL?
A) UNION ALL removes duplicate records, while UNION does not
B) UNION sorts the result set, but UNION ALL does not
C) UNION removes duplicate records, while UNION ALL includes duplicates
D) There is no difference; they function identically

Question 5: What does the GROUP BY clause do?
A) Groups the result set by specified columns and excludes duplicates
B) Groups the result set by specified columns and includes duplicates
C) Summarizes or aggregates identical data into summary rows
D) Orders the entire result set by specified columns

Answers (Don’t Cheat Yourself!)

1. B) Only rows that have matching values in both joined tables
2. C) It returns all rows from the left table, and the matched rows from the right table
3. A) When you need all records when there is a match in either the left or right table
4. C) UNION removes duplicate records, while UNION ALL includes duplicates
5. C) Summarizes or aggregates identical data into summary rows