Lesson 5B: CREATE, INSERT, UPDATE, DELETE

Lesson 5B will explore the practical aspects of SQL operations. This lesson is engineered to bridge the gap between theoretical knowledge and real-world application, providing you with hands-on experience in creating tables, inserting data, and maintaining your database through updates and deletions. Through a simulated project, you'll apply these skills in a context that mirrors actual database management scenarios.

What's Included in This Lesson:

1. Table Creation: Explore the process of designing and creating tables for a hypothetical database project. You'll learn to define table schemas, choose appropriate data types, establish primary keys, and set up relationships between tables.
2. Data Insertion Techniques: Explore various methods for inserting data into your newly created tables. From single-row inserts to bulk data loading, understand the nuances of populating tables efficiently and accurately.
3. Updating and Deleting Records: Gain proficiency in modifying existing data within your tables through updates and deletions. Learn how to pinpoint specific records for updates or removal, ensuring data integrity and relevance.
4. Simulated Database Project: Engage in a simulated project that involves setting up a database for a school's extracurricular activities and student participation. This scenario will guide you through creating tables for students, activities, and enrollments, followed by populating and maintaining this database.
5. Best Practices: Throughout the lesson, best practices for database design, data entry, and data maintenance will be highlighted to help you avoid common pitfalls and ensure your database is robust and reliable.

Scenario Introduction:
Imagine a high school looking to better organize and analyze its extracurricular activities and student participation. Your task is to create a database that captures information about students, the various activities offered, and student enrollments in these activities.

Part 1: Table Creation

1. Students Table: Holds information about students.

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    grade_level VARCHAR(4)
);
```

Explanation: This SQL statement is a command to create a new table called `students` in a database.

- **`CREATE TABLE students`:** This part of the statement initiates the creation of a new table within the database and names it `students`. The `CREATE TABLE` command is used to define a new table along with its structure.

`student_id SERIAL PRIMARY KEY`:

- `student_id` is a column name intended to uniquely identify each student record in the table.
- `SERIAL` is a data type that auto-increments. Whenever a new record is inserted into the table, PostgreSQL automatically generates a unique value for this column, starting at 1 and incrementing by 1 for each new record. This feature makes it ideal for primary keys.
- `PRIMARY KEY` designates this column as the primary key of the table. A primary key constraint enforces uniqueness across the column's values, ensuring that no two rows have the same `student_id` and that no `student_id` value is `NULL`. It uniquely identifies each row in the table.

`first_name VARCHAR(50)`: `first_name` is a column designed to store the first name of each student. `VARCHAR(50)` specifies the data type of the `first_name` column. `VARCHAR` stands for character varying, allowing for variable-length strings up to the maximum length specified—in this case, 50 characters. It means this column can store strings (text data) up to 50 characters long.

`last_name VARCHAR(50)`: Similar to `first_name`, `last_name` is a column for storing the student's last name. `VARCHAR(50)` here again specifies that this column can contain text data up to 50 characters in length.

**grade_level VARCHAR(4)`:** `grade_level` is intended to store the grade level of each student, such as "9th", "10th", and so on. `VARCHAR(4)` indicates this column can hold strings of up to 4 characters, accommodating grade levels typically represented by 2 to 4 characters (including possible suffixes like "th").

2. Activities Table: Lists extracurricular activities.

```
CREATE TABLE activities (
    activity_id SERIAL PRIMARY KEY,
    activity_name VARCHAR(100),
    sponsor_teacher VARCHAR(100)
);
```

Explanation: This SQL command creates a new table named `activities` with three columns.

1. `activity_id SERIAL PRIMARY KEY`: A unique identifier for each activity that auto-increments with each new record, serving as the primary key.

2. `activity_name VARCHAR(100)`: A column for the name of the activity, allowing text up to 100 characters.

3. `sponsor_teacher VARCHAR(100)`: Stores the name of the teacher sponsoring the activity, with a maximum length of 100 characters.

3. Enrollments Table: Tracks student participation in activities.

```
CREATE TABLE enrollments (
    enrollment_id SERIAL PRIMARY KEY,
    student_id INTEGER REFERENCES students(student_id),
    activity_id INTEGER REFERENCES activities(activity_id),
    enrollment_date DATE
```

```
);
```

Explanation:
This SQL command creates a new table called `enrollments` designed to record student enrollments in various activities. It includes four columns.
1. `enrollment_id SERIAL PRIMARY KEY`: Assigns a unique auto-incrementing identifier to each enrollment record, serving as the primary key.

2. `student_id INTEGER REFERENCES students(student_id)`: This column stores integer values representing the IDs of students. `REFERENCES students(student_id)` establishes a foreign key relationship between this `student_id` in the `enrollments` table and the `student_id` in the `students` table. It ensures that every `student_id` in `enrollments` matches a valid `student_id` in `students`, maintaining referential integrity and linking each enrollment to a specific student.

3. `activity_id INTEGER REFERENCES activities(activity_id)`: Similar to `student_id`, this column holds integer IDs for activities. `REFERENCES activities(activity_id)` creates a foreign key relationship to the `activity_id` in the `activities` table. This linkage guarantees that each `activity_id` in `enrollments` corresponds to an existing activity, connecting each enrollment record to a particular activity.

4. `enrollment_date DATE`: Records the date of enrollment using the `DATE` data type, which stores calendar dates (year, month, day).

Part 2: Data Insertion
Now, let's insert sample data into `students` and `activities` tables, and then create enrollment records.

```
INSERT INTO students (first_name, last_name, grade_level) VALUES ('Jane', 'Doe', '11th');
INSERT INTO activities (activity_name, sponsor_teacher) VALUES ('Chess Club', 'Mr. Smith');
INSERT INTO enrollments (student_id, activity_id, enrollment_date) VALUES (1, 1, '2023-09-01');
```

Explanation: The process of inserting data into SQL tables involves adding new records to an existing table structure. The `INSERT INTO` statement is fundamental for this process, enabling you to specify the target table, the columns for which you're providing data, and the values for each specified column.

Example Inserts:

- The first statement inserts a new student named Jane Doe in the 11th grade into the `students` table.
- The second statement adds a new activity, Chess Club sponsored by Mr. Smith, into the `activities` table.
- The third statement creates a new enrollment record linking the student with ID 1 to the activity with ID 1, marking the enrollment date as September 1, 2023.

Potential Challenges When Inserting Data:
1. Data Type Mismatches: If the data you're trying to insert doesn't match the column's data type (e.g., inserting text into an INTEGER column), the database will reject the insert operation.

2. Violating Constraints: Attempting to insert a record that violates table constraints (e.g., unique constraints, foreign key constraints) will result in an error. For example, inserting a student_id into `enrollments` that doesn't exist in `students` will fail due to the foreign key constraint.

3. Incomplete Data: Not providing values for mandatory fields without default values specified in the table schema will lead to errors.

4. Incorrect NULL Handling: Inserting NULL into a column that is set as NOT NULL will cause an error.

Inserting Data from CSV Files vs. Line-by-Line:

Line-by-Line Insertion:
- Involves manually writing `INSERT INTO` statements for each record.
- Provides high control over individual records but can be time-consuming for large datasets.
- Prone to human error in typing or in adhering to constraints and data types.

Inserting Data from CSV Files:
- Databases like PostgreSQL allow for bulk data import from CSV files, which is efficient for loading large datasets.
- Tools like pgAdmin or command-line utilities (e.g., `psql`'s `\copy` command) facilitate this process.
- CSV import must be carefully prepared to match the table's schema, including the order and data type of columns.

SQL File Inserts:
- SQL files containing pre-written `INSERT INTO` statements can be executed to insert multiple records at once.
- Useful for transferring data between databases or for backups.
- Similar to line-by-line inserts but automated and less prone to manual entry errors.

Differences in Processes:
- Efficiency: Bulk importing from CSV or executing SQL files is significantly more efficient for large datasets compared to manual line-by-line insertion.
- Control: Manual insertion offers more control over each record but is less practical for large volumes of data.
- Error Handling: Bulk operations may fail if any part of the data doesn't comply with the schema or constraints, requiring careful data preparation and error checking.

Part 3: Updating and Deleting Records

a. Update an activity record to change the sponsor teacher.

```
UPDATE activities SET sponsor_teacher = 'Ms. Johnson' WHERE activity_id = 1;
```

Explanation: This SQL Statement updates a record within the `activities` table, specifically changing the `sponser_teacher` field of the activity with an `activity_id` of 1 to `Ms. Johnson`.

1. `UPDATE activities`: This part of the statement specifies that the update operation is to be performed on the `activities` table. The `UPDATE` command is used to modify existing records in a table.

2. `SET sponsor_teacher = 'Ms. Johnson'`: The `SET` clause indicates which column(s) should be updated and what new value(s) they should be set to. In this case, it directs the database to change the `sponsor_teacher` column's value to 'Ms. Johnson' for the specified record(s).

3. `WHERE activity_id = 1;`: The `WHERE` clause specifies the condition that must be met for the records to be updated. Here, it specifies that only the record with an `activity_id` of 1 should have its `sponsor_teacher` updated. This condition ensures that the update affects only the intended record.

Key Points:

- Targeted Update: This operation carefully targets a specific record using the `WHERE` clause, preventing unintended changes to other records in the table.
- Importance of the WHERE Clause: Without the `WHERE` clause, the update operation would apply to every record in the `activities` table, changing every `sponsor_teacher` to 'Ms. Johnson'. Always use a `WHERE` clause unless you intentionally want to update all records.
- Data Integrity: The update respects the integrity of the database. Assuming 'Ms. Johnson' is a valid teacher within the context of the database, this change maintains the consistency and accuracy of the `activities` table.

b. Delete a student's enrollment record.

```
DELETE FROM enrollments WHERE student_id = 1 AND activity_id = 1;
```

Explanation: This SQL statement is used to delete a specific record from the `enrollments` table where both the `student_id` and `activity_id` match the specified criteria, in this case, both being equal to 1.

1. `DELETE FROM enrollments`: This part specifies that the deletion operation targets the `enrollments` table. The `DELETE` command is used to remove records from a table based on a specified condition.

2. `WHERE student_id = 1 AND activity_id = 1;`: The `WHERE` clause defines the condition(s) that must be met for records to be deleted. In this instance, the condition specifies that the record(s) to be deleted must have a `student_id` of 1 and an `activity_id` of 1. The use of `AND` means both conditions must be true for the record to be selected for deletion.

Key Points:

- **Precise Targeting:** This operation specifically targets the deletion of records where both the student and activity IDs match the given values, ensuring only the intended record is removed. This precision is crucial to avoid accidentally deleting other records.
- **Impact on Database Integrity:** Deleting data from a database can have significant implications, especially in tables that are part of relational structures. It's important to ensure that deleting this record does not violate referential integrity constraints or leave orphaned records in related tables.
- **Data Loss:** Deletion operations are irreversible within the context of a single transaction once committed. It's vital to be certain that a record should indeed be removed before executing a `DELETE` statement. Always consider backing up data or checking constraints to prevent accidental loss of important information.

This example demonstrates how to use the `DELETE` statement in SQL to remove data that is no longer needed or relevant, maintaining the cleanliness and relevance of the data within your database.