SUMMARY: CRUD OPERATIONS

SESSION OVERVIEW:

By the end of this session, the students will be able to:

- Understand and apply CREATE, READ, UPDATE, and DELETE operations on databases.
- Understand the concepts of EMPTY values and NULL values using the WHERE clause.
- Understand the uses and the concepts of TRUNCATE functions.
- Understand the uses and the concepts of DROP function.
- Understand the difference between DELETE, TRUNCATE, and DROP commands.

KEY TOPICS AND EXAMPLES:

Understanding and applying CREATE, READ, UPDATE, and DELETE operations on databases:

Importance of CRUD operations in SQL:
Understanding CRUD operations in SQL (Create, Read, Update, Delete) is crucial for several reasons, especially considering that these operations form the backbone of most database interactions. Whether you're developing applications, managing databases, or analyzing data, CRUD operations are fundamental in handling data stored in relational databases.

1. **CREATE Operations:**

   The CREATE TABLE statement in SQL is a fundamental part of setting up and maintaining databases. It allows database administrators and developers to define the structure of a database in a detailed and organized manner.

   a. **Database Schema Definition:** The primary use of CREATE TABLE is to define how data is structured within the database. Each table is designed to hold data in a specific format and layout, dictated by the fields (columns) defined in the CREATE TABLE statement.

   b. **Data Integrity:** Each column in a table is defined with a specific data type (e.g., integer, text, date), which ensures that only data of that type can be stored in the column. This helps prevent data entry errors.

   c. **Data Relationships:** CREATE TABLE includes the ability to define foreign keys that help maintain referential integrity between tables. This is crucial for relational databases where the relationships between different data entities are key.

Creating a table in SQL from scratch:

Syntax:

```
CREATE TABLE table_name (
    column1 datatype1 [constraint],
    column2 datatype2 [constraint],
    column3 datatype3 [constraint],
    ...
    [table_constraints]
);
```

Note:

- *table_name: The name of the table you want to create.*
- *column1, column2, column3, ...: The names of the columns in the table.*
- *datatype1, datatype2, datatype3, ...: The data types for each column (e.g., INT, VARCHAR, DATE, etc.).*
- *constraint: Optional. Constraints for each column (e.g., NOT NULL, UNIQUE, PRIMARY KEY).*
- *table_constraints: Optional. Constraints that apply to the whole table (e.g., FOREIGN KEY, CHECK).*

Example 1:

Note: Creating this table to showcase how we can create a table from scratch. This does not imply that this table will be used in the subsequent lectures.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE NOT NULL,
    Salary DECIMAL(10, 2), -- Example with decimal for salary
    BirthDate DATE NOT NULL,
    Gender CHAR(1), -- Example using CHAR for gender ('M' or 'F')
    Active BOOLEAN DEFAULT TRUE
);
```

Output:
The above query will help us create an employee table in the specified schema along with the specified columns. The columns will have the specified data types which can be changed whenever required. (The temporary changing of data type has been discussed in session 2 using CAST and the permanent changes of the data type will be taught in the upcoming sessions)

(In this session we will not use the constraints and the table_constraints as we will be discussing these in detail in the upcoming sessions.)

Creating a table in SQL from the existing table:

We will understand the creation of a table from the existing table with the help of the following uses:

- **Backup or Snapshot:** Creating a table from an existing table can be used as a way to create a backup or snapshot of the data in the original table at a specific point in time. This can be useful for archiving data or for creating a copy of the data for analysis or reporting purposes.

Syntax:

```
CREATE TABLE new_table_name AS
SELECT * FROM existing_table_name;
```

Example:

```
CREATE TABLE CustomersDetails AS
SELECT * FROM Customers;
```

The above query will create a separate table containing all the records similar to the Customers table. Just the new table name will be changed from Customers to CustomerDetails.

- **Data Transformation:** You can create a new table with transformed or aggregated data from an existing table. This can involve applying filters, aggregations, or other transformations to the data.

Syntax:

```
CREATE TABLE new_table_name AS
SELECT Distinct *
FROM existing_table_name;
```

Output:

The above syntax will create a separate table containing all the unique records from the existing table. This will help us to remove the duplicates from the table if there are any duplicates available in the table.

2. <u>**READ Operations:**</u>

In CRUD operations, 'R' is an acronym for read, which means retrieving or fetching the data from the SQL table. So, we will use the SELECT command to fetch the inserted records from the SQL table. We can retrieve all the records from a table using an asterisk (*) in a SELECT query. There is also an option of retrieving only those records that satisfy a particular condition by using the WHERE clause in a SELECT query.

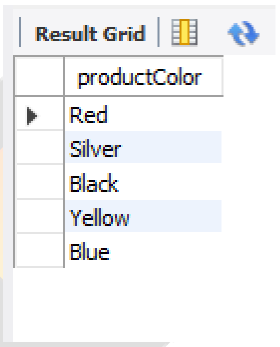
Example 1:

```
# Query to find the colors of the products from performing well
financially in the market.

SELECT DISTINCT productColor
```

```
FROM products
WHERE ProductPrice> 1000 AND ProductCost<1000;
```

Output:


Result Grid

```
productColor
Red
Silver
Black
Yellow
Blue
```

3. **UPDATE Operations:**

   The UPDATE statement in SQL is used to update the data of an existing table in the database. We can update single columns as well as multiple columns using the UPDATE statement as per our requirement.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Components of the UPDATE Statement:

- **UPDATE table_name:** Specifies the table in which data is to be updated. Replace table_name with the name of the table where the changes will occur.
- **SET**: Followed by the column(s) you want to update and the new value(s) for those columns. We can update one or multiple columns in a single UPDATE statement. Each column is separated by a comma when updating multiple columns.
- **WHERE:** This clause specifies which rows should be updated. Conditions in the WHERE clause identify the specific rows that will be modified. If the WHERE clause is omitted, all rows in the table will be updated, which must be done with caution to avoid unintended data changes.

Example 1:

```
#Query to update a single record in SQL.
UPDATE Customers
SET EmailAddress = 'huang10@learnsector.com', HomeOwner = 'Y'
WHERE CustomerKey = 11001;
```

Output:

The above query will help us update the email address and homeowner of the customer with CustomerKey as 11001.

Example 2:

```
# Query to update multiple rows at a time.
UPDATE customers
SET emailaddress ='huang@learnsector.com'
WHERE lastname='huang';
```

Output:
The above query will help us update all the records that contain Huang as the last name of the customers. So, this is how we can update multiple rows at a time.
(NOTE: The above query is just a hypothetical query to show the use case of how to update multiple records using UPDATE operations.)

Error:
Whenever we use the UPDATE operations we have to make use of the WHERE clause to specify the particular record(s). If we do not use the WHERE clause then the whole table/ column will get updated. At the industry level, the amount of data that gets stored is huge thus creating haphazard. Hence, the use of the WHERE clause in the UPDATE operations needs to be assured to ensure the integrity of the data.

For example:
In the above example 2, if we hadn't used the WHERE clause, then what would have happened?
Output:
The output would update the whole email address column to huang@learnsector.com.
Thus, it is always important to use the WHERE clause.

4. **DELETE Operations:**

   SQL DELETE is a basic SQL operation used to delete data in a database. SQL DELETE is an important part of database management DELETE can be used to selectively remove records from a database table based on certain conditions. This SQL DELETE operation is important for database size management, data accuracy, and integrity.

Syntax:

```
DELETE FROM table_name
WHERE some_condition;
```

Example 1:

```
# Query to delete the rows according to the specified condition.
DELETE FROM customers
WHERE CustomerKey=12020;
```

Example 2:

```
# Query to delete all the rows.
DELETE FROM customers;
```

The above query will delete all the rows from the table. Thus, it is always advisable to mention the condition while using the DELETE clause.

Understanding the concepts of EMPTY values and NULL values:

Before jumping into NULL values, we will understand about the EMPTY values.
When we import data using Table Data Import Wizard, the NULL values aren't identified as NULL values rather they are stored as EMPTY values.

Here is an example to showcase the difference between NULL values and EMPTY values:
If you go through the customers table you will notice that the AnnualIncome and the Prefix columns have missing values. (Using the term missing values, as we do not know if it is an EMPTY value or NULL value.)
Whenever we see any missing values we tend to identify them as NULL values, but that's not the case always. Let's consider it to be a NULL value and try the query which will help us find the NULL value.

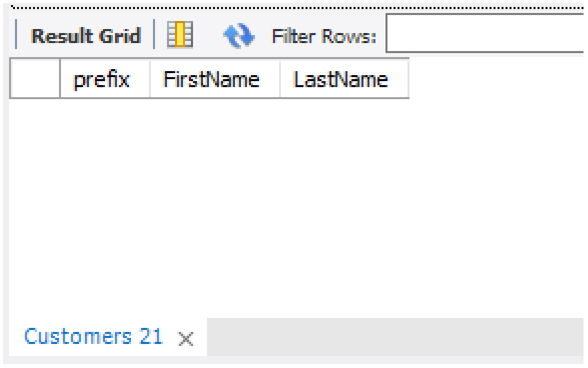
Syntax:

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

Example:

```
Select prefix, FirstName, LastName
from Customers
Where Prefix is null;
```

Output:



prefix	FirstName	LastName

The above output does not return any records which means the missing values in the prefix column does not contain NULL values.

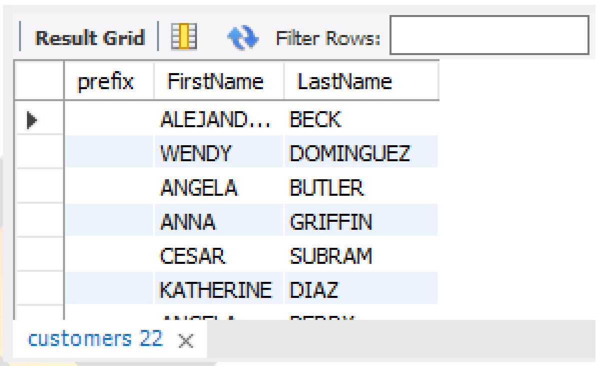
Now let's run a query that will help us understand if the missing values are EMPTY values or not.

Example:

```
Select prefix, FirstName, LastName
From customers
```

```
where Prefix = '';
```

Output:



```
Result Grid | Filter Rows:
  prefix  FirstName   LastName
▶         ALEJAND...  BECK
          WENDY       DOMINGUEZ
          ANGELA      BUTLER
          ANNA        GRIFFIN
          CESAR       SUBRAM
          KATHERINE   DIAZ
customers 22 ×
```

The above output returns all the records in which the prefix contains missing values.
Thus we can conclude that the missing values in the prefix column are not NULL values, rather these are EMPTY values.

How to convert EMPTY values to NULL values:

So, whenever we encounter empty values, here is a simple method to convert the empty value to a NULL value.

1. Using this query we will identify if it is an EMPTY value or NULL value.

```
Select prefix, FirstName, LastName
From customers
where Prefix = '';
```

2. For better integrity of the data, you want to make all the EMPTY values into NULL values. Using this query we will convert the EMPTY values into NULL values.

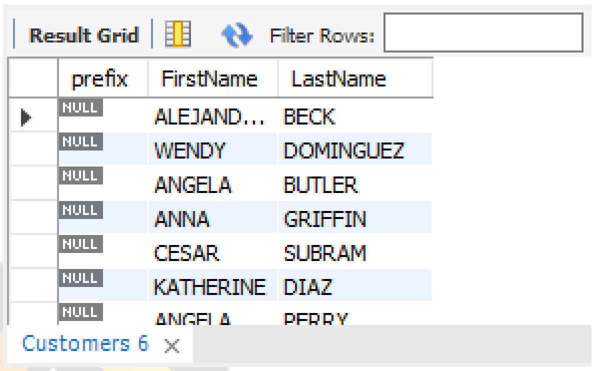
```
UPDATE customers
SET Prefix= NULL
WHERE Prefix = '';
```

The above query will update all the EMPTY values present in the Prefix column to NULL values.

3. Now, run the query to find if the EMPTY values have been converted to NULL.

```
Select prefix, FirstName, LastName
from Customers
Where Prefix is null;
```

Output:


Result Grid | Filter Rows:

prefix	FirstName	LastName
NULL	ALEJAND...	BECK
NULL	WENDY	DOMINGUEZ
NULL	ANGELA	BUTLER
NULL	ANNA	GRIFFIN
NULL	CESAR	SUBRAM
NULL	KATHERINE	DIAZ
NULL	ANGELA	PERRY

Customers 6

In the above output, we can notice that the query returns only those rows in which the Prefix column had the EMPTY values and now they have been converted to NULL values.

NOTE: You can try the same process for the AnnualIncome column yourself to understand the concept more clearly.

Using IS NULL and IS NOT NULL:

- These are the correct operators to use when you want to check if a column's value is NULL or not NULL.
- **Example Usage:**
  To find all records where a column phone_number is NULL, you would use the query:

```
SELECT *
FROM contacts
WHERE phone_number IS NULL;
```

Similarly, to find records where phone_number is not NULL:

```
SELECT *
FROM contacts
WHERE phone_number IS NOT NULL;
```

Incorrect Use of = or != with NULL:
Using = or != to compare a value with NULL does not work as one might intuitively think, because NULL is not a value. For example, the following query will not return any results because it's not possible to confirm if any value is equal to NULL:

```
SELECT *
FROM contacts
WHERE phone_number = NULL;
```

This query incorrectly attempts to find rows where phone_number is NULL, but since NULL cannot be compared using =, it returns zero rows.

Differences between the NULL values and EMPTY values:

- **Empty Values:**
  An empty value typically refers to a lack of data or a data value that contains no characters. For example, an empty string '' or a string with only spaces ' ' is considered empty. Empty values are considered to be present but contain no data.
- **NULL Values:**
  NULL represents a missing, unknown, or undefined value. It is not the same as an empty string or any specific value like zero. Columns that are not explicitly given a value when a record is created are set to NULL by default unless otherwise specified.

  The analogy to understand the NULL values and EMPTY values:

- **Empty Values:** Imagine a bookshelf where some shelves are empty. An empty shelf is like an empty value. It's there, but there are no books on it. The shelf exists, but it doesn't contain any information.
- **NULL Values:** Now, consider a missing shelf or a shelf with a sign saying "shelf missing." This is like a NULL value. There's no shelf at all, so you can't even say it's empty because the shelf itself doesn't exist to hold any books.

<u>Understanding the uses and the concepts of TRUNCATE functions:</u>

The TRUNCATE command in SQL is used to quickly delete all rows from a table and reset the table's identity seed (if it has one) to its initial value. It is similar to the DELETE statement without a WHERE clause, but TRUNCATE is typically faster and uses fewer system resources because it does not log individual row deletions. Here are some common uses of the TRUNCATE command:

- **Removing all rows from a table:** When you need to quickly remove all data from a table, TRUNCATE can be more efficient than using DELETE.
- **Resetting identity seed:** If the table has an identity column (auto-increment), TRUNCATE will reset the identity seed to its initial value.
- **Performance:** TRUNCATE is generally faster than DELETE for removing all rows from a table, especially for large tables, because it does not log individual row deletions and does not fire triggers.

Disadvantages of using the TRUNCATE command in SQL:

- **No WHERE clause:** Unlike the DELETE command, TRUNCATE does not support a WHERE clause. This means you cannot selectively delete rows based on a condition. TRUNCATE deletes all rows in the table.
- **Cannot be rolled back**: TRUNCATE is not transactional and cannot be rolled back. Once you execute the TRUNCATE command, the data is permanently deleted, and you cannot recover it using a rollback operation.
- **Resets identity column:** If the table has an identity column (auto-increment), TRUNCATE resets the identity seed to its initial value. This can lead to gaps in the sequence of values in the column, which might be undesirable in some cases.

Syntax:

```
TRUNCATE TABLE table_name;
```

Caution:
The TRUNCATE command should be used very carefully because it permanently deletes the records of the particular table and cannot be recovered. Thus, the use of the TRUNCATE command should be limited.

Understanding the use of DROP:

The DROP command in SQL is used to delete database objects such as tables, indexes, or views. It permanently removes the specified object from the database, along with all its data and associated indexes, triggers, and constraints.

Syntax 1:

```
# Syntax to drop a database.
DROP DATABASE database_name;
```

Syntax 2:

```
# Syntax to drop a table.
DROP TABLE table_name;
```

Caution:
The DROP command should be used very carefully because it permanently deletes the table/ Database as specified and cannot be recovered. Thus, the use of the DROP command should be limited.

Difference between DELETE, TRUNCATE and DROP:

DELETE	TRUNCATE	DROP
Data Manipulation Language Command (DML)	Data Definition Language Command (DDL)	Data Definition Language Command (DDL)
Used to delete content in rows of a table.	Used to delete entire content of table leaving the table structure.	Used to delete the entire content of table along with the table structure.
DELETE FROM table_name WHERE condition; (to delete the row of the table as per the condition) DELETE FROM table_name; (to delete all the records of the table)	TRUNCATE table <table_name>;	DROP table <table_name>;
Can be Rollback	Cannot be Rollback	Cannot be Rollback
Removes specific rows depending on the condition	Removes all rows	Removes the entire data immediately.

Less efficient for large tables as we have to manually specify each and every condition.	Efficiency depends on the size of the object which is being dropped.	More efficient for large tables as we are removing all the data in one step.