## Agenda

- CRUD
  - Read
    - BETWEEN Operator
    - LIKE Operator
    - IS NULL Operator
    - ORDER BY Clause revisited
    - LIMIT Clause
  - Update
  - Delete
    - Delete vs Truncate vs Drop
    - Truncate
    - Drop

---

## BETWEEN Operator

Now, we are going to start the discussion about another important keyword in SQL, `BETWEEN` .

Let's say we want to get all the films from the `film` table which have a release year >= `2005` and <= `2010` . We can do this by ORing 2 conditions. We can also use the `BETWEEN` operator to do that. Example:

```sql
SELECT * FROM film WHERE release_year BETWEEN 2005 AND 2010;
```

BETWEEN operator is inclusive of the values specified. So, the above query will return all the films which have a release year >= `2005` and <= `2010` . So that is something to be mindful of.

Between Operator also works for strings. Let's assume that there is a country table with a "name" column of type varchar. If we execute this query:

```sql
Select * from country where name between 'a' and 'b';
```

We will get this result:

```
Argentina
.
.
.
Argelia.

-- The above query will give us all country names starting with A/a till B/b.
-- The above query willl limit answers till letter b only. Ex: 'Bolivia' will not be included since it have more letters than just b.
-- Therefore above query gives all countries between a till b. Regardless of case sensitivity.
```

Between works with other data-types as well such as dates. Let's say there is an orders table and we want all orders between dates '2023-07-01' AND '2024-01-01'.

```sql
SELECT * FROM Orders
WHERE OrderDate BETWEEN '2023-07-01' AND '2024-01-01';
```

> *Try this above query with your own variations.*

---

## LIKE Operator

LIKE operator is one of the most important and frequently used operator in SQL. Whenever there is a column storing strings, there comes a requirement to do some kind of pattern matching. Example, assume Scaler's database where we have a `batches` table with a column called `name` . Let's say we want to get the list of `Academy` batches and the rule is that an Academy batch shall have `Academy` somewhere within the name it can be at starting, at end or anywhere in the name of batch. How do we find those? We can use the `LIKE` operator for this purpose.

Let's talk about how the `LIKE` operator works. The `LIKE` operator works with the help of 2 wildcards in our queries, `%` and `_`. The `%` wildcard matches any number of characters (>= 0 occurrences of any set of characters). The `_` wildcard matches exactly one character (any character). Example:

1. LIKE 'cat%' will match "cat", "caterpillar", "category", etc. but not "wildcat" or "dog".
2. LIKE '%cat' will match "cat", "wildcat", "domesticcat", etc. but not "cattle" or "dog".
3. LIKE '%cat%' will match "cat", "wildcat", "cattle", "domesticcat", "caterpillar", "category", etc. but not "dog" or "bat".
4. LIKE '_at' will match "cat", "bat", "hat", etc. but not "wildcat" or "domesticcat".
5. LIKE 'c_t' will match "cat", "cot", "cut", etc. but not "chat" or "domesticcat".
6. LIKE 'c%t' will match "cat", "chart", "connect", "cult", etc. but not "wildcat", "domesticcat", "caterpillar", "category".

**Example:**

```
SELECT * FROM batches WHERE name LIKE '%Academy%';
```

Similarly, let's say in our Sakila database, we want to get all the films which have `LOVE` in their title. We can use the `LIKE` operator. Example:

```
SELECT * FROM film WHERE title LIKE '%LOVE%';

-- These pattern strings are case insensitive as well.
-- Hence below query will give same results as above.

SELECT * FROM film WHERE title LIKE '%LovE%';
```

> *Conclusion*

Some of the key points to remember are:

- A significant tool for pattern-based data searching is the LIKE operator in MySQL. The underscore (_) wildcard character is used to match a single character, whereas the percentage (%) wildcard character is used to match any number of characters (zero or more) in a string.
- To verify if you have understood the LIKE operator, let us have few quizzes.
- These pattern strings will be considered as case insensitive as well.
- Extra Resource for Like operator: https://www.scaler.com/topics/like-in-mysql/

---

## Quiz 1

If you want to find all customers from a 'Customers' table whose names end with 'son', which SQL query would you use?

### Choices

- ☐ SELECT * FROM Customers WHERE Name LIKE 'son%'
- ☐ SELECT * FROM Customers WHERE Name LIKE '%son'
- ☐ SELECT * FROM Customers WHERE Name LIKE 'son'
- ☐ SELECT * FROM Customers WHERE Name LIKE 'son'

---

## Quiz 2

In a 'Books' table, you want to select all books whose titles contain the word 'moon'. Which of the following queries should you use?

### Choices

- ☐ SELECT * FROM Books WHERE Title LIKE 'moon%'
- ☐ SELECT * FROM Books WHERE Title LIKE '%moon'

- ☐ `SELECT * FROM Books WHERE Title LIKE '%moon%'`
- ☐ `SELECT * FROM Books WHERE Title LIKE 'moon_'`

---

## Quiz 3

Suppose you have an 'Orders' table and you want to find all orders whose 'OrderNumber' has '123' at the exact middle. Assume 'OrderNumber' is a five-character string. What query should you use?

### Choices

- ☐ `SELECT * FROM Orders WHERE OrderNumber LIKE '%123%'`
- ☐ `SELECT * FROM Orders WHERE OrderNumber LIKE '123%'`
- ☐ `SELECT * FROM Orders WHERE OrderNumber LIKE '_123_'`
- ☐ `SELECT * FROM Orders WHERE OrderNumber LIKE '%123'`

---

## IS NULL Operator

Now, we are almost at the end of the discussion about different operators. Do you all remember how we store empties, that is, no value for a particular column for a particular row? We store it as `NULL` / `None` . Interestingly working with NULLs is a bit tricky. We cannot use the `=` operator to compare a column with `NULL` .

> *An Empty box and empty brain aren't same things. Similarly an empty number and an empty string are considered different objects.*

---

> *Pic credits: anonymous*

---

**Example:**

```sql
SELECT * FROM film WHERE description = NULL;
```

The above query will not return any rows. Why? Because `NULL` is not equal to `NULL`. Infact, `NULL` is not equal to anything. Nor is it not equal to anything. It is just `NULL`.

Example:

```sql
SELECT NULL = NULL;
```

The above query will return `NULL`. Similarly, `3 = NULL` , `3 <> NULL` , `NULL <> NULL` will also return `NULL`. So, how do we compare a column with `NULL`? We use the `IS NULL` operator. Example:

```sql
SELECT * FROM film WHERE description IS NULL;
```

Similarly, we can use the `IS NOT NULL` operator to find all the rows where a particular column is not `NULL`. Example:

```sql
SELECT * FROM film WHERE description IS NOT NULL;
```

In many assignments, you will find that you will have to use the `IS NULL` and `IS NOT NULL` operators. Without them you will miss out on rows that had NULL values in them and get the wrong answer. Example: Find customers with id other than 2. If you use `=` operator, you will miss out on the customer with id `NULL`.

```sql
SELECT * FROM customers WHERE id != 2;
```

The above query will not return the customer with id `NULL`. So, you will get the wrong answer. Instead, you should use the `IS NOT NULL` operator. Example:

```sql
SELECT * FROM customers WHERE id IS NOT NULL AND id != 2;
```

---

## ORDER BY clause continued:

Now let's discuss another important clause. ORDER BY clause allows to return values in a sorted order. Example:

```sql
SELECT * FROM film ORDER BY title;
```

The above query will return all the rows from the `film` table in ascending order of the `title` column. If you want to return the rows in descending order, you can use the `DESC` keyword. Example:

```sql
SELECT * FROM film ORDER BY title DESC;
```

You can also sort by multiple columns. Example:

```sql
SELECT * FROM film ORDER BY title, release_year;
```

The above query will return all the rows from the `film` table in ascending order of the `title` column and then in ascending order of the `release_year` column. Consider the second column as tie breaker. If 2 rows have same value of title, release year will be used to break tie between them. Example:

```sql
SELECT * FROM film ORDER BY title DESC, release_year DESC;
```

Above query will return all the rows from the `film` table in descending order of the `title` column and if tie on `title`, in descending order of the `release_year` column.

By the way, you can ORDER BY on a column which is not present in the SELECT clause. Example:

```sql
SELECT title FROM film ORDER BY release_year;
```

Let's also build the analogy of this with a pseudocode.

```
answer = []

for each row in film:
    if row.matches(conditions in where clause) # new line from above
        answer.append(row)

answer.sort(column_names in order by clause)

filtered_answer = []

for each row in answer:
    filtered_answer.append(row['rating'], row['release_year'])

return filtered_answer
```

If you see, the `ORDER BY` clause is applied after the `WHERE` clause. So, first the rows are filtered based on the `WHERE` clause and then they are sorted based on the `ORDER BY` clause. And only after that are the columns that have to be printed taken out. And that's why you can sort based on columns not even in the `SELECT` clause.

---

## ORDER BY Clause with DISTINCT keyword

When employing the DISTINCT keyword in an SQL query, the ORDER BY clause is limited to sorting by columns explicitly specified in the SELECT clause. This restriction stems from the nature of DISTINCT, which is designed to eliminate duplicate records based on the selected columns.

Consider the scenario where you attempt to order the results by a column not included in the SELECT clause, as demonstrated in this example:

```sql
SELECT DISTINCT title FROM film ORDER BY release_year;
```

The SQL engine would generate an error in this case. The reason behind this restriction lies in the potential ambiguity introduced when sorting by a column not present in the SELECT clause.

When you use DISTINCT, the database engine identifies unique values in the specified columns and returns a distinct set of records. However, when you attempt to order these distinct records by a column that wasn't part of the selection, ambiguity arises.

Take the example query:

```sql
SELECT DISTINCT title FROM film ORDER BY release_year;
```

Here, the result set will include distinct titles from the film table, but the sorting order is unclear. Multiple films may share the same title but have different release years. Without explicitly stating which release year to consider for sorting, the database engine encounters ambiguity.

By limiting the ORDER BY clause to columns present in the SELECT clause, you provide a clear directive on how the results should be sorted. In the corrected query:

```sql
SELECT DISTINCT title FROM film ORDER BY title;
```

You instruct the database engine to sort the distinct titles alphabetically by the title column, avoiding any confusion or ambiguity in the sorting process. This ensures that the results are not only distinct but also ordered in a meaningful and unambiguous manner.

---

## LIMIT Clause

LIMIT clause allows us to limit the number of rows returned by a query. Example:

```sql
SELECT * FROM film LIMIT 10;
```

The above query will return only 10 rows from the `film` table. If you want to return 10 rows starting from the 11th row, you can use the `OFFSET` keyword. Example:

```
SELECT * FROM film LIMIT 10 OFFSET 10;
```

The above query will return 10 rows starting from the 11th row from the `film` table. You can also use the `OFFSET` keyword without the `LIMIT` keyword. Example:

```
SELECT * FROM film OFFSET 10;
```

The above query will return all the rows starting from the 11th row from the `film` table.

LIMIT clause is applied at the end. Just before printing the results. Taking the example of pseudocode, it works as follows:

```
answer = []

for each row in film:
    if row.matches(conditions in where clause) # new line from above
        answer.append(row)

answer.sort(column_names in order by clause)

filtered_answer = []

for each row in answer:
    filtered_answer.append(row['rating'], row['release_year'])

return filtered_answer[start_of_limit: end_of_limit]
```

Thus, if your query contains ORDER BY clause, then LIMIT clause will be applied after the ORDER BY clause. Example:

```
SELECT * FROM film ORDER BY title LIMIT 10;
```

The above query will return 10 rows from the `film` table in ascending order of the `title` column.

---

## Update

Now let's move to learn U of CRUD. Update and Delete are thankfully much simple, so don't worry, we will be able to breeze through it over the coming 20 mins. As the name suggests, this is used to update rows in a table. The general syntax is as follows:

```
UPDATE table_name SET column_name = value WHERE conditions;
```

Example:

```
UPDATE film SET release_year = 2006 WHERE id = 1;
```

The above query will update the `release_year` column of the row with `id` 1 in the `film` table to 2006. You can also update multiple columns at once. Example:

```
UPDATE film SET release_year = 2006, rating = 'PG' WHERE id = 1;
```

Let's talk about how update works. It works as follows:

```
for each row in film:
    if row.matches(conditions in where clause)
        row['release_year'] = 2006
        row['rating'] = 'PG'
```

So basically update query iterates through all the rows in the table and updates the rows that match the conditions in the where clause. So, if you have a table with 1000 rows and you run an update query without a where clause, then all the 1000 rows will be updated. Example:

```sql
UPDATE film SET release_year = 2006;
-- By default MySQL works with Safe_Mode 'ON' which prevents us from doing this kind of operations.
```

The above query will result in all the rows of table having release_year as 2006, which is not desired. So, be careful while running update queries.

---

## Delete

Finally, we are at the end of CRUD. Let's talk about Delete operations. The general syntax is as follows:

```sql
DELETE FROM table_name WHERE conditions;
```

Example:

```sql
DELETE FROM film WHERE id = 1;
```

The above query will delete the row with `id` 1 from the `film` table.

Beware, If you don't specify a where clause, then all the rows from the table will be deleted. Example:

```sql
DELETE FROM film;
-- By default MySQL works with Safe_Mode 'ON' which prevents us from doing this kind of operations.
```

Let's talk about how delete works as well in terms of code.

```
for each row in film:
    if row.matches(conditions in where clause)
        delete row
```

---

## Delete vs Truncate vs Drop

There are two more commands which are used to delete rows from a table. They are `TRUNCATE` and `DROP`. Let's discuss them one by one.

#### Truncate

The command looks as follows:

```sql
TRUNCATE film;
```

The above query will delete all the rows from the `film` table. TRUNCATE command internally works by removing the complete table and then recreating it. So, it is much faster than DELETE. But it has a disadvantage. It cannot be rolled back meaning you can't get back your data. We will learn more about rollbacks in the class on Transactions. But at a high level, this is because as the complete table is deleted as an intermediate step, no log is maintained as to what all rows were deleted, and thus is not easy to revert. So, if you run a TRUNCATE query, then you cannot undo it.

> *Note: It also resets the primary key ID. For example, if the highest ID in the table before truncating was 10, then the next row inserted after truncating will have an ID of 1.*

#### Drop

The command looks as follows:

Example:

```sql
DROP TABLE film;
```

The above query will delete the `film` table. The difference between `DELETE` and `DROP` is that `DELETE` is used to delete rows from a table and `DROP` is used to delete the entire table. So, if you run a `DROP` query, then the entire table will be deleted. All the rows and the table structure will be deleted. So, be careful while running a `DROP` query. Nothing will be left of the table after running a `DROP` query. You will have to recreate the table from scratch.

Note that, DELETE:

1. Removes specified rows one-by-one from table based on a condition(may delete all rows if no condition is present in query but keeps table structure intact).
2. It is slower than TRUNCATE since we delete values one by one for each rows.
3. Doesn't reset the key. It means if there is an auto_increment key such as student_id in students table and `last student_id value is 1005` and we deleted this entry using query:

```
DELETE FROM students WHERE student_id = 1005;
```

- Now, if we insert one more entry/row in students table then student_id for this column will be 1006. Hence continuing with same sequence without resseting the value.

5. It can be rolled back. Means if we have deleted a value then we can get it back again.

TRUNCATE:

1. Removes the complete table and then recreats it with same schema (columns).
2. Faster than DELETE. Since Truncate doesn't delete values one by one rather it deletes the whole table at once by de-referencing it and then creates another table with same schema hecne Truncate is faster.
3. Resets the key. It means if there is an auto_increment key such as student_id in students table and `last student_id value is 1005` and we Truncated this whole table then in new table the fresh entry/row will start with student_id = 1.
4. It can not be rolled back because the complete table is deleted as an intermediate step meaning we can't get the same table back.

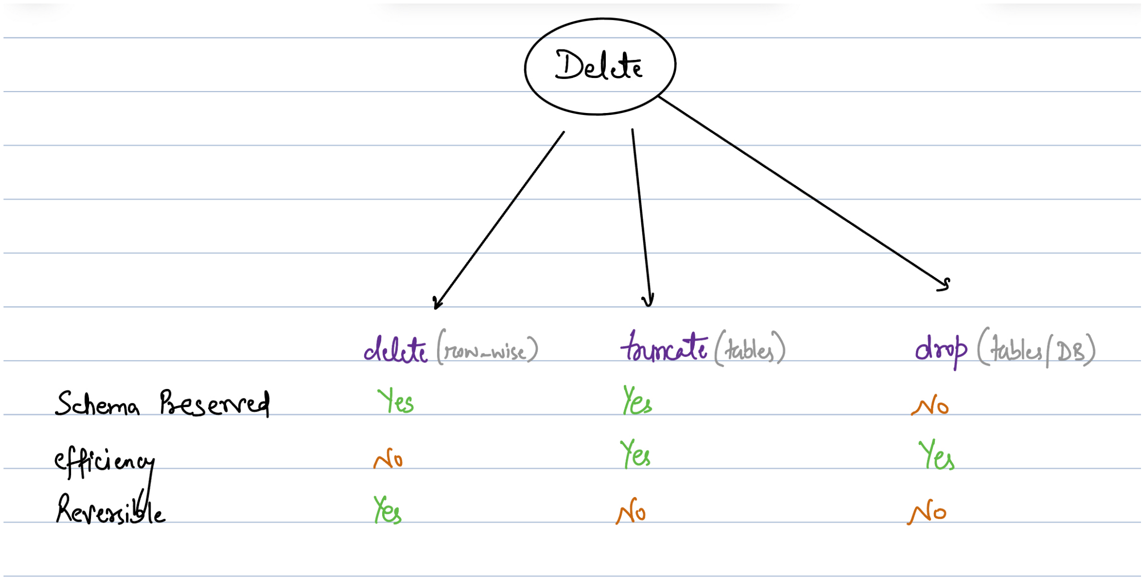
DROP:

1. Removes complete table and the table structre as well.
2. It can not be rolled back meaning that we can't get back our table or database.

---

> ***Diagram for reference:***

---



| | delete (row-wise) | truncate (tables) | drop (tables/DB) |
|---|---|---|---|
| Schema Reserved | Yes | Yes | No |
| efficiency | No | Yes | Yes |
| Reversible | Yes | No | No |

---

## Extra Reading materials

Learn more about Delte/Truncate/Drop using our Scaler Topic's article:
https://www.scaler.com/topics/difference-between-delete-drop-and-truncate/

SQL functions:
https://docs.google.com/document/d/1IFGuCvFv8CIcq_4FTIBusuARa81Oak4_snK1qJF8C54/edit#heading=h.gjdgxs

---

## Solution to Quizzes:

> *-- Quiz1: Option B (SELECT * FROM Customers WHERE Name LIKE '%son') Quiz2: Option C (SELECT * FROM Books WHERE Title LIKE '%moon%') Quiz3: Option C (SELECT * FROM Orders WHERE OrderNumber LIKE '_123_') --*