M.K. Institute of Computer Studies, Bharuch
F.Y.B.C.A. (SEM – 1)
105: Data Manipulation and Analysis (DMA)
UNIT-5
NOTES

UNIT-5: Queries (Single Table only)

5.1 Using where clause and operators with where clause:

- **5.1.1 In, between , like, not in, =, !=, >, =, <=, wildcard operators**
- **5.1.2 Order by, Group by, Distinct**
- **5.1.3 AND, OR operators, Exists and not Exists**
- **5.1.4 Use of Alias**

5.2 Constraints (Table level and Attribute Level)

- **5.2.1 NOT NULL, CHECK, DEFAULT**
- **5.2.2 UNIQUE, Primary Key, Foreign Key**
- **5.2.3 On Delete Cascade**

5.3 SQL Functions :

- **5.3.1 Aggregate Functions: avg(), max(), min(), sum(), count(), first(), last().**
- **5.3.2 Scalar Functions: ucase(), lcase(), round(), mid().**

5.4 Creating sequence

5.5 Views :

- **5.5.1 Creating simple view, updating view, dropping view.**
- **5.5.2 Difference between View and Table.**

INTRODUCTION TO QUERY

- Database language which is used to create, maintain and retrieve the relational database.
- **Query** is a way of requesting information from the database. A database query can be either a select query or an action query.
- Query Processing is the activity performed in extracting data from the database. In query processing, it takes various steps for fetching the data from the database.
- For example, a manager can perform a query to select the employees who were hired 5 months ago. The results could be the basis for creating performance evaluations.
- One of several different query languages may be used to perform a range of simple to complex database queries.
- SQL, the most well-known and widely-used query language, is familiar to most database administrators (DBAs).

5.1 Using where clause and operators with where clause:

- The WHERE clause is used to filter records.
- **Where clause** is fetch a particular row or set of rows from a table.
- This clause filters records based on given conditions and only those row(s) comes out as result that satisfies the condition defined in WHERE clause of the SQL query.
- The SQL WHERE clause is used to filter the results and apply conditions in a SELECT, INSERT, UPDATE, or DELETE statement
- SQL Where Clause Syntax

```
SELECT Column_name1, Column_name2, ....  [OR ]
FROM Table_name
WHERE Condition;
```

1. Example - One Condition in the WHERE Clause

In this example, we have a table called *suppliers* with the following data:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

```
SELECT * FROM suppliers WHERE state = 'California';
```

There will be 4 records selected. These are the results that you should see:

supplier_id	supplier_name	city	state
200	Google	Mountain View	California
300	Oracle	Redwood City	California
700	Dole Food Company	Westlake Village	California
900	Electronic Arts	Redwood City	California

2. Example - Two Conditions in the WHERE Clause (AND Condition)

we can use the AND condition in the WHERE clause to specify more than 1 condition that must be met for the record to be selected.
In this example, we have a table called *customers* with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

```
SELECT * FROM customers
WHERE  favorite_website = 'techonthenet.com' AND customer_id > 6000;
```

There will be 1 record selected. These are the results that you should see:

customer_id	last_name	first_name	favorite_website
9000	Johnson	Derek	techonthenet.com

3. Example - Two Conditions in the WHERE Clause (OR Condition)

You can use the OR condition in the WHERE clause to test multiple conditions where the record is returned if any one of the conditions are met.

In this example, we have a table called *products* with the following data:

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

```
SELECT * FROM products
WHERE product_name = 'Pear' OR product_name = 'Apple';
```

There will be 2 records selected. These are the results that you should see:

product_id	product_name	category_id
1	Pear	50
4	Apple	50

4. Example - Combining AND & OR conditions

You can also combine the AND condition with the OR condition to test more complex conditions.

Let's use the *products* table again for this example.

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

```
SELECT * FROM products
WHERE (product_id > 3 AND category_id = 75) OR (product_name = 'Pear');
```

There will be 2 records selected. These are the results that you should see:

product_id	product_name	category_id
1	Pear	50
5	Bread	75

Comparison Operators

Comparison operators are used in the WHERE clause to determine which records to select.

Comparison Operator	Description
=	Equal
<>	Not Equal

Comparison Operator	Description
!=	Not Equal
>	Greater Than
>=	Greater Than or Equal
<	Less Than
<=	Less Than or Equal
IN ( )	Matches a value in a list
NOT	Negates a condition
BETWEEN	Within a range (inclusive)
IS NULL	NULL value
IS NOT NULL	Non-NULL value
LIKE	Pattern matching with % and _
EXISTS	Condition is met if subquery returns at least one row

Example - Equality Operator

In SQL, you can use the = operator to test for equality in a query.
In this example, we have a table called *suppliers* with the following data:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

```
SELECT * FROM suppliers    WHERE    supplier_name = 'Microsoft';
```

There will be 1 record selected. These are the results that you should see:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington

Example - Inequality Operator

In SQL, there are two ways to test for inequality in a query.
You can use either the <> or != operator. Both will return the same results.
Let's use the same *suppliers* table as the previous example.

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
300	Oracle	Redwood City	California

supplier_id	supplier_name	city	state
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

```
SELECT * FROM suppliers    WHERE   supplier_name <> 'Microsoft';
```

OR

```
SELECT * FROM suppliers    WHERE   supplier_name != 'Microsoft';
```

There will be 8 records selected. These are the results you should see with either one of the SQL statements:

supplier_id	supplier_name	city	state
200	Google	Mountain View	California
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

Example - Greater Than Operator

You can use the > operator in SQL to test for an expression greater than.
In this example, we have a table called *customers* with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

```
SELECT * FROM customers   WHERE    customer_id > 6000;
```

There will be 3 records selected. These are the results that you should see:

customer_id	last_name	first_name	favorite_website
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

Example - Greater Than or Equal Operator

In SQL, you can use the >= operator to test for an expression greater than or equal to.
Let's use the same *customers* table as the previous example.

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

```
SELECT * FROM customers   WHERE    customer_id >= 6000;
```

There will be 4 records selected. These are the results that you should see:

customer_id	last_name	first_name	favorite_website
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

Example - Less Than Operator

You can use the < operator in SQL to test for an expression less than.
In this example, we have a table called *products* with the following data:

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

```
SELECT * FROM products     WHERE    product_id < 5;
```

There will be 4 records selected. These are the results that you should see:

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50

Example - Less Than or Equal Operator

In SQL, you can use the <= operator to test for an expression less than or equal to.
Let's use the same *products* table as the previous example.

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

```
SELECT * FROM products     WHERE     product_id <= 5;
```

There will be 5 records selected. These are the results that you should see:

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75

IN Condition

The IN condition (sometimes called the IN operator) allows you to easily test if an expression matches any value in a list of values.
It is used to help reduce the need for multiple OR conditions in a **SELECT, INSERT, UPDATE**, or **DELETE** statement.

Syntax:

```
                    expression IN (value1, value2, .... value_n);
```

OR

```
                    expression IN (subquery);
```

Here, **expression : This is a value to test.**
value1, value2 ...,value_n: These are the values to test against *expression*. If any of these values matches *expression*, then the IN condition will evaluate to true.

subquery: This is a SELECT statement whose result set will be tested against *expression*. If any of these values matches *expression*, then the IN condition will evaluate to true.

Example - Using the IN Condition with Character Values
In this example, we have a table called *suppliers* with the following data:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas

supplier_id	supplier_name	city	state
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

```
SELECT * FROM suppliers    WHERE supplier_name IN ('Microsoft', 'Oracle', 'Flowers Foods');
```

There will be 3 records selected. These are the results that you should see:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
300	Oracle	Redwood City	California
800	Flowers Foods	Thomasville	Georgia

It is equivalent to the following SQL statement:

```
SELECT * FROM suppliers    WHERE
supplier_name = 'Microsoft'          OR       supplier_name = 'Oracle'
OR       supplier_name = 'Flowers Foods';
```

As you can see, using the IN condition makes the statement easier to read and more efficient than using multiple OR conditions.

Example - Using the IN Condition with Numeric Values

In this example, we have a table called *customers* with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

```
SELECT * FROM customers   WHERE customer_id IN (5000, 7000, 8000, 9000);
```

There will be 4 records selected. These are the results that you should see:

customer_id	last_name	first_name	favorite_website
5000	Smith	Jane	digminecraft.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

It is equivalent to the following SQL statement:

```
SELECT * FROM customers          WHERE   customer_id = 5000          OR
          customer_id = 7000          OR       customer_id = 8000          OR
customer_id = 9000;
```

Example - Using the IN Condition with the NOT Operator

In this example, we have a table called *products* with the following data:

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

```
SELECT * FROM products     WHERE    product_name NOT IN ('Pear', 'Banana', 'Bread');
```

There will be 4 records selected. These are the results that you should see:

product_id	product_name	category_id
3	Orange	50
4	Apple	50
6	Sliced Ham	25
7	Kleenex	NULL

It is equivalent to the following SQL statement:

```
SELECT * FROM products     WHERE product_name <> 'Pear' AND product_name <> 'Banana'
AND product_name <> 'Bread';
```

As you can see, the equivalent statement is written using AND conditions instead of OR conditions because the IN condition is negated.

BETWEEN Condition

The SQL BETWEEN condition allows you to easily test if an expression is within a range of values (inclusive).

It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

Syntax:

```
expression BETWEEN value1 AND value2;
```

here, expression :A column or calculation.

value1 and value2 :These values create an inclusive range that *expression* is compared to.

Note

- The SQL BETWEEN Condition will return the records where *expression* is within the range of *value1* and *value2* (inclusive).

Example

In this example, we have a table called *suppliers* with the following data:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
300	Oracle	Redwood City	California

supplier_id	supplier_name	city	state
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

```
SELECT * FROM suppliers     WHERE supplier_id BETWEEN 300 AND 600;
```

There will be 4 records selected. These are the results that you should see:

supplier_id	supplier_name	city	state
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin

OR

It is equivalent to the following SELECT statement:

```
SELECT * FROM suppliers     WHERE supplier_id >= 300   AND supplier_id <= 600;
```

Example - Using BETWEEN Condition with Date Values

In this example, we have a table called *orders* with the following data:

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

```
SELECT  * FROM orders       WHERE order_date
 BETWEEN '2016/04/19' AND '2016/05/01';
```

There will be 3 records selected. These are the results that you should see:

order_id	customer_id	order_date
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

This example would return all records from the *orders* table where the *order_date* is between Apr 19, 2016 and May 1, 2016 (inclusive).

Example - Using NOT Operator with the BETWEEN Condition

In this example, we have a table called *customers* with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

```
SELECT * FROM customers   WHERE customer_id          NOT BETWEEN 5000 AND 8000;
```

There will be 2 records selected. These are the results that you should see:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
9000	Johnson	Derek	techonthenet.com

This would return all rows where the *customer_id* was **NOT** between 5000 and 8000, inclusive.
OR
It would be equivalent to the following SELECT statement:

```
SELECT * FROM customers WHERE customer_id < 5000  OR customer_id > 8000;
```

LIKE Condition

The LIKE condition allows you to use wildcards to perform pattern matching in a query.
The LIKE condition is used in the WHERE clause of a SELECT, INSERT, UPDATE, or DELETE statement.

Syntax

```
expression LIKE pattern [ ESCAPE 'escape_character' ]
```

Here, **expression: A character expression such as a column or field.**

pattern: **A character expression that contains pattern matching. The wildcards that you can choose from are:**

Wildcard	Explanation
%	Allows you to match any string of any length (including zero length)
_	Allows you to match on a single character

ESCAPE 'escape_character' :Optional. It allows you to pattern match on literal instances of a wildcard character such as % or _.

Example - Using % Wildcard in the LIKE Condition

Let's explain how the % wildcard works in the LIKE condition.
Remember that the % wildcard matches any string of any length (including zero length).

In this first example, we want to find all of the records in the *customers* table where the customer's *last_name* begins with 'J'.

In this example, we have a table called *customers* with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

```
SELECT * FROM customers   WHERE last_name LIKE 'J%'  ORDER BY last_name;
```

There will be 2 records selected. These are the results that you should see:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
9000	Johnson	Derek	techonthenet.com

This example returns the records in the *customers* table where the *last_name* starts with 'J'. As you can see, the records for the last names Jackson and Johnson have been returned.

Using Multiple **%** Wildcards in the LIKE Condition

Using the same *customers* table with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

Let's try to find all *last_name* values from the *customers* table where the *last_name* contains the letter 'e'. Enter the following SQL statement:

```
SELECT last_name FROM customers WHERE last_name LIKE '%e%' ORDER BY last_name;
```

There will be 3 records selected. These are the results that you should see:

last_name
Anderson
Ferguson
Reynolds

Example - Using _ Wildcard in the LIKE Condition

Remember that _ wildcard is looking for exactly one character, unlike the % wildcard.

Using the *categories* table with the following data:

category_id	category_name
25	Deli
50	Produce

category_id	category_name
75	Bakery
100	General Merchandise
125	Technology

Let's try to find all records from the *categories* table where the *category_id* is 2-digits long and ends with '5'.

Enter the following SQL statement:

```
SELECT * FROM categories   WHERE category_id LIKE '_5';
```

There will be 2 records selected. These are the results that you should see:

category_id	category_name
25	Deli
75	Bakery

In this example, there are 2 records that will pattern match - the *category_id values* 25 and 75. Notice that the *category_id* of 125 was not selected because, the _ wilcard matches only on a single character.

Using Multiple Wildcards in the LIKE Condition

If you wanted to match on a 3-digit value that ended with '5', you would need to use the _ wildcard two times. You could modify your query as follows:

```
SELECT * FROM categories WHERE category_id LIKE '__5';
```

Now you will return the *category_id* value of 125:

category_id	category_name
125	Technology

Example - Using the NOT Operator with the LIKE Condition

Next, let's look at an example of how to use the NOT Operator with the LIKE condition.

In this example, we have a table called *suppliers* with the following data:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

Let's look for all records in the *suppliers* table where the *supplier_name* does **not** contain the letter 'o'.

Enter the following SQL statement:

```
SELECT * FROM suppliers WHERE supplier_name NOT LIKE '%o%';
```

There will be 1 record selected. These are the results that you should see:

supplier_id	supplier_name	city	state
400	Kimberly-Clark	Irving	Texas

In this example, there is only one record in the *suppliers* table where the *supplier_name* does not contain the letter 'o'.

Example - Using Escape Characters with the LIKE Condition

It is important to understand how to "Escape Characters" when pattern matching.
You can escape % or _ and search for the literal versions instead.

In this example, we a table called *test* with the following data:

test_id	test_value
1	10%
2	25%
3	100
4	99

We could return all records from the *test* table where the *test_value* contains the % literal.
Enter the following SQL statement:

```
SELECT * FROM test WHERE test_value LIKE '%!%%' escape '!';
```

These are the results that you should see:

test_id	test_value
1	10%
2	25%

You could further modify the above example and only return *test_values* that start with 1 and contain the % literal.
Enter the following SQL statement:

```
SELECT * FROM test WHERE test_value LIKE '1%!%%' escape '!';
```

These are the results that you should see:

test_id	test_value
1	10%

SQL: EXISTS Condition

The SQL EXISTS condition is used in combination with a subquery and is considered to be met, if the subquery returns at least one row. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

Syntax

```
WHERE EXISTS ( subquery );
```

here, **subquery :**

- The *subquery* is a SELECT statement.
- If the *subquery* returns at least one record in its result set, the EXISTS clause will evaluate to true and the EXISTS condition will be met.
- If the *subquery* does not return any records, the EXISTS clause will evaluate to false and the EXISTS condition will not be met.

Note

- SQL statements that use the EXISTS condition are very inefficient since the sub-query is rerun for EVERY row in the outer query's table. There are more efficient ways to write most queries, that do not use the EXISTS condition.

Example - Using EXISTS Condition with the SELECT Statement

In this example, we have a *customers* table with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

And a table called *orders* with the following data:

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20

Order by

- The ORDER BY keyword is used to sort the result-set in ascending or descending order.
- The ORDER BY keyword sorts the records in ascending order by default.
- To sort the records in descending order, use the DESC keyword.

Syntax

```
SELECT column1,column2,.....
FROM table_name
ORDER BY column1,column2... ASC | DESC;
```

here, ASC :Optional.

ASC sorts the result set in ascending order by expression. This is the default behavior, if no modifier is provider.

DESC :Optional. DESC sorts the result set in descending order by expression.

Example - Sorting Results in Ascending Order

In this example, we have a table called *customers* with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

```
SELECT * FROM customers ORDER BY last_name;
```

There will be 6 records selected. These are the results that you should see:

customer_id	last_name	first_name	favorite_website
8000	Anderson	Paige	NULL
6000	Ferguson	Samantha	bigactivities.com
4000	Jackson	Joe	techonthenet.com
9000	Johnson	Derek	techonthenet.com
7000	Reynolds	Allen	checkyourmath.com
5000	Smith	Jane	digminecraft.com

OR

```
SELECT * FROM customers ORDER BY last_name ASC;
```

Example - Sorting Results in descending order

In this example, we have a table called *suppliers* with the following data:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

```
SELECT * FROM suppliers WHERE supplier_id > 400 ORDER BY supplier_id DESC;
```

There will be 5 records selected. These are the results that you should see:

supplier_id	supplier_name	city	state
900	Electronic Arts	Redwood City	California
800	Flowers Foods	Thomasville	Georgia
700	Dole Food Company	Westlake Village	California
600	SC Johnson	Racine	Wisconsin
500	Tyson Foods	Springdale	Arkansas

Example - Using both ASC and DESC attributes

In this example, let's use the same *products* table as the previous example:

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50

product_id	product_name	category_id
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

```
SELECT * FROM products WHERE product_id <> 7 ORDER BY category_id DESC, product_name ASC;
```

There will be 6 records selected. These are the results that you should see:

product_id	product_name	category_id
5	Bread	75
4	Apple	50
2	Banana	50
3	Orange	50
1	Pear	50
6	Sliced Ham	25

Group by

- The Group By statement is used for organizing similar data into groups.
- The data is further organized with the help of equivalent function.
- Group by clause is used to group the results of a SELECT query based on one or more columns.
- It is also used with SQL functions to group the result from one or more tables.
- It means, if different rows in a precise column have the same values, it will arrange those rows in a group.
    - The SELECT statement is used with the GROUP BY clause in the query.
    - WHERE clause is placed before the GROUP BY clause.
    - ORDER BY clause is placed after the GROUP BY clause.

Syntax :

```
SELECT expression1, expression2, ... expression_n,        function_name(aggregate_expression)
FROM tables
[WHERE conditions]
GROUP BY expression1, expression2, ... expression_n
[ORDER BY expression [ ASC | DESC ]];
```

Here

expression1, expression2, ... expression_n :Expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY Clause at the end of the SQL statement.

aggregate_function: This is an aggregate function such as the SUM, COUNT, MIN, MAX, or AVG functions.

aggregate_expression: This is the column or expression that the *aggregate_function* will be used on.

tables : The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.

WHERE conditions : Optional.

These are conditions that must be met for the records to be selected.

ORDER BY expression: Optional.

The expression used to sort the records in the result set. If more than one expression is provided, the values should be comma separated.

ASC |DESC : Optional.

ASC sorts the result set in ascending order by *expression*. This is the default behavior, if no modifier is provider.

DESC sorts the result set in descending order by expression.

Example of **Group by** in a Statement

Consider the following Emp table.

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	9000
405	Tiger	35	8000

SQL query for the above requirement will be,

```
SELECT name, age FROM Emp GROUP BY salary
```

Result will be,

name	age
Rohan	34
Shane	29
Anu	22

Example of **Group by** in a Statement with **WHERE** clause

Consider the following Emp table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	9000
405	Tiger	35	8000

```
SELECT name, salary FROM Emp
WHERE age > 25
 GROUP BY salary
```

Result will be.

name	salary

Rohan	6000
Shane	8000
Scott	9000

You must remember that Group By clause will always come at the end of the SQL query, just like the Order by clause.

Example - Using GROUP BY with the SUM Function

In this example, we have a table called *employees* with the following data:

employee_number	last_name	first_name	salary	dept_id
1001	Smith	John	62000	500
1002	Anderson	Jane	57500	500
1003	Everest	Brad	71000	501
1004	Horvath	Jack	42000	501

```
SELECT dept_id, SUM(salary) AS total_salaries
FROM employees
GROUP BY dept_id;
```

There will be 2 records selected. These are the results that you should see:

dept_id	total_salaries
500	119500
501	113000

In this example, we've used the SUM function to add up all of the salaries for each *dept_id* and we've aliased the results of the SUM function as *total_salaries*.

Example - Using GROUP BY with the COUNT function

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

```
SELECT category_id, COUNT(*) AS total_products
FROM products
WHERE category_id IS NOT NULL
GROUP BY category_id
ORDER BY category_id;
```

There will be 3 records selected. These are the results that you should see:

category_id	total_products
25	1

category_id	total_products
50	4
75	1

Example - Using GROUP BY with the MIN function

employee_number	last_name	first_name	salary	dept_id
1001	Smith	John	62000	500
1002	Anderson	Jane	57500	500
1003	Everest	Brad	71000	501
1004	Horvath	Jack	42000	501

```
SELECT dept_id, MIN(salary) AS lowest_salary
FROM employees
GROUP BY dept_id;
```

There will be 2 records selected. These are the results that you should see:

dept_id	lowest_salary
500	57500
501	42000

Example - Using GROUP BY with the MAX function

employee_number	last_name	first_name	salary	dept_id
1001	Smith	John	62000	500
1002	Anderson	Jane	57500	500
1003	Everest	Brad	71000	501
1004	Horvath	Jack	42000	501

```
SELECT dept_id, MAX(salary) AS highest_salary
FROM employees
GROUP BY dept_id;
```

There will be 2 records selected. These are the results that you should see:

dept_id	highest_salary
500	62000
501	71000

Distinct

- The SQL DISTINCT command is used with SELECT key word to retrieve only distinct or unique data.
  OR
- The SQL DISTINCT clause is used to remove duplicates from the result set of a SELECT statement.
- In a table, there may be a chance to exist a duplicate value and sometimes we want to retrieve only unique values. In such scenarios, SQL SELECT DISTINCT statement is used.

Syntax:

```
SELECT DISTINCT column_name ,column_name
FROM table_name
[WHERE conditions];
```

HERE,
expressions : The columns or calculations that you wish to retrieve.
tables : The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.
WHERE conditions : Optional. The conditions that must be met for the records to be selected.

Note

- When only one expression is provided in the DISTINCT clause, the query will return the unique values for that expression.
- When more than one expression is provided in the DISTINCT clause, the query will retrieve unique combinations for the expressions listed.
- In SQL, the DISTINCT clause doesn't ignore NULL values. So when using the DISTINCT clause in your SQL statement, your result set will include NULL as a distinct value.

Example :

Student_Name	Gender	Mobile_Number	HOME_TOWN
Rahul Ojha	Male	7503896532	Lucknow
Disha Rai	Female	9270568893	Varanasi
Sonoo Jaiswal	Male	9990449935	Lucknow

```
SELECT DISTINCT home_town
FROM students
```

Now, it will return two rows.

HOME_TOWN
Lucknow
Varanasi

Example - Finding Unique Values in a Column

In this example, we have a table called *suppliers* with the following data:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

```
SELECT DISTINCT state FROM suppliers ORDER BY state;
```

There will be 6 records selected. These are the results that you should see:

state
Arkansas
California
Georgia
Texas
Washington
Wisconsin

Example - Finding Unique Values in Multiple Columns

Using the same *suppliers* table from the previous example, enter the following SQL statement:

```
SELECT DISTINCT city, state FROM suppliers ORDER BY city, state;
```

There will be 8 records selected. These are the results that you should see:

city	state
Irving	Texas
Mountain View	California
Racine	Wisconsin
Redmond	Washington
Redwood City	California
Springdale	Arkansas
Thomasville	Georgia
Westlake Village	California

Example - How the DISTINCT Clause handles NULL Values

In this example, we have a table called *products* with the following data:

product_id	product_name	category_id
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

Now let's select the unique values from the *category_id* field which contains a NULL value.

```
SELECT DISTINCT category_id FROM products ORDER BY category_id;
```

There will be 4 records selected. These are the results that you should see:

category_id
NULL
25

category_id
50
75

5.1.4 Use of Alias

- SELECT AS is used to assign temporary names to table or column name or both.
- This is known as creating Alias.

Why use Alias in SQL?

1. To reduce the amount of time to query by temporary replacing the complex & long table and column names with simple & short names.
2. This method is also used to protect the column names of the databases by not showing the real column names on the screen.
3. Alias are useful when we are working with JOIN operations or aggregate functions such as COUNT, SUM etc.

Alias Facts

1. An alias only temporary renames the column or table name, it lasts for the duration of select query. The changes to the names are not permanent.
2. This technique of creating alias is generally used by DBA (Database Administrators) or Database users.
3. The temporary table name is also called correlation name.

Syntax:

```
column_name [AS] alias_name
```

OR

```
table_name [AS] alias_name
```

Here, **column_name : The original name of the column that you wish to alias.**
table_name : The original name of the table that you wish to alias.
alias_name : The temporary name to assign.

Note

- If the *alias_name* contains spaces, you must enclose the *alias_name* in quotes.
- It is acceptable to use spaces when you are aliasing a column name. However, it is not generally good practice to use spaces when you are aliasing a table name.
- The *alias_name* is only valid within the scope of the SQL statement.

Alias Example

Table: STUDENT

STUDENT_ID	STUDENT_NAME	STUDENT_AGE	STUDENT_ADDRESS
1001	Negan	29	Noida
1002	Sirius	28	Delhi
1003	Ron	28	Delhi
1004	Luna	30	Agra

```
Query :
```

```
SELECT STUDENT_ID AS ID, STUDENT_NAME AS NAME, STUDENT_ADDRESS ADDRESS FROM STUDENT;
```

Result:

```
ID          NAME        ADDRESS
-----       ------      -------
1001          Negan       Noida
1002          Sirius      Delhi
1003          Ron         Delhi
1004          Luna        Agra
```

Example - How to Alias a Column Name

Generally, aliases are used to make the column headings in your result set easier to read. Most commonly, you will alias a column when using an aggregate function such as MIN, MAX, AVG, SUM or COUNT in your query.

Let's look at an example of how to use to alias a column name in SQL.
In this example, we have a table called *employees* with the following data:

employee_number	last_name	first_name	salary	dept_id
1001	Smith	John	62000	500
1002	Anderson	Jane	57500	500
1003	Everest	Brad	71000	501
1004	Horvath	Jack	42000	501

```
SELECT dept_id, COUNT(*) AS total
FROM employees
GROUP BY dept_id;
```

There will be 2 records selected. These are the results that you should see:

dept_id	total
500	2
501	2

In this example, we've aliased the COUNT(*) field as total. As a result, total will display as the heading for the second column when the result set is returned. Because our *alias_name* did not include any spaces, we are not required to enclose the *alias_name* in quotes.

5.2 Constraints (Table level and Attribute Level)

- Constraints are the rules that we can apply on the type of data in a table.
- SQL Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

- Constraints can be divided into the following two types,
    1. Column level constraints: Limits only column data.
    2. Table level constraints: Limits whole table data.

- Constraints are used to make sure that the integrity of data is maintained in the database.
- Following are the most used constraints that can be applied to a table.
    - NOT NULL : Ensures that a column cannot have a NULL value
    - UNIQUE : Ensures that all values in a column are different

- PRIMARY KEY : A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY : Uniquely identifies a row/record in another table
- CHECK: Ensures that all values in a column satisfies a specific condition
- DEFAULT : Sets a default value for a column when no value is specified

NOT NULL:

- NOT NULL constraint makes sure that a column does not hold NULL value.
- When we don't provide value for a particular column while inserting a record into a table, it takes NULL value by default.
- By specifying NULL constraint, we can be sure that a particular column(s) cannot have NULL values.

Example 1:

```
CREATE TABLE STUDENT(ROLL_NO INT NOT NULL,STU_NAME VARCHAR (35) NOT NULL,
STU_AGE INT NOT NULL,
STU_ADDRESS VARCHAR (235),
PRIMARY KEY (ROLL_NO)
);
```

Example 2:

```
CREATE TABLE Persons (    ID int NOT NULL,    LastName varchar(255) NOT NULL,
   FirstName varchar(255) NOT NULL,    Age int);
```

CHECK:

- Using the CHECK constraint we can specify a condition for a field, which should be satisfied at the time of entering values for this field.
- For example, the below query creates a table Student and specifies the condition for the field AGE as (AGE >= 18).

Example 1:

```
CREATE TABLE Student(ID int(6) NOT NULL,NAME varchar(10) NOT NULL,AGE int NOT NULL
CHECK (AGE >= 18)              );
```

Example 2:

```
CREATE table Student( s_id int NOT NULL CHECK(s_id > 0),
Name varchar(60) NOT NULL, Age int );
```

The above query will restrict the s_id value to be greater than zero.

DEFAULT

- The DEFAULT constraint is used to provide a default value for a column while inserting a record into a table.
- The default value will be added to all new records IF no other value is specified.
- That is, if at the time of entering new records in the table if the user does not specify any value for these fields then the default value will be assigned to them.
- For example, the below query will create a table named Student and specify the default value for the field AGE as 18.

EXAMPLE 1:

```
CREATE TABLE Student
(
ID int(6) NOT NULL,
NAME varchar(10) NOT NULL,
AGE int DEFAULT 18
);
```

EXAMPLE 2:

```
CREATE TABLE STUDENT(ROLL_NO   INT  NOT NULL,
STU_NAME VARCHAR (35) NOT NULL,
STU_AGE INT NOT NULL,
EXAM_FEE INT  DEFAULT 10000,
STU_ADDRESS VARCHAR (35) ,
PRIMARY KEY (ROLL_NO)      );
```

UNIQUE

- UNIQUE constraint ensures that a field or column will only have unique values that is columns are different.
- A UNIQUE constraint field will not have duplicate data. This constraint can be applied at column level or table level.
- This constraint helps to uniquely identify each row in the table. i.e. for a particular column, all the rows should have unique values.
- We can have more than one UNIQUE columns in a table.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.
- For example, the below query creates a tale Student where the field ID is specified as UNIQUE. i.e, no two students can have the same ID. Unique constraint in detail.

EXAMPLE 1:

```
CREATE TABLE Student (ID int(6) NOT NULL UNIQUE,NAME varchar(10),ADDRESS varchar(20));
```

Using **UNIQUE** constraint after Table is created (Column Level)

```
ALTER TABLE Student ADD UNIQUE(s_id);
```

The above query specifies that s_id field of Student table will only have unique value.

Example:

Here we are setting up the UNIQUE Constraint for two columns: STU_NAME & STU_ADDRESS. which means these two columns cannot have duplicate values.

Note: STU_NAME column has two constraints (NOT NULL and UNIQUE both) setup.

```
CREATE TABLE STUDENTS(      ROLL_NO   INT          NOT NULL,
```

```
    STU_NAME VARCHAR (35)   NOT NULL UNIQUE,
    STU_AGE  INT          NOT NULL,
    STU_ADDRESS  VARCHAR (35) UNIQUE,
    PRIMARY KEY (ROLL_NO)             );
```

Primary Key

- Primary Key is a field which uniquely identifies each row in the table.
- If a field in a table as primary key, then the field will not be able to contain NULL values as well as all the rows should have unique values for this field.
- So, in other words we can say that this is combination of NOT NULL and UNIQUE constraints.

EXAMPLE 1:

- A table can have only one field as primary key.Below query will create a table named Student and specifies the field ID as primary key.

```
CREATE TABLE Student (      ID int(6) NOT NULL UNIQUE,          NAME varchar(10),
                            ADDRESS varchar(20),                PRIMARY KEY(ID)
                        );
```

Using PRIMARY KEY constraint at Table Level

```
CREATE table Student (s_id int PRIMARY KEY, Name varchar(60) NOT NULL, Age int);
```

The above command will creates a PRIMARY KEY on the s_id.

Using PRIMARY KEY constraint at Column Level

```
ALTER table Student ADD PRIMARY KEY (s_id);
```

The above command will creates a PRIMARY KEY on the s_id.

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

Foreign Key

- Foreign Key is a field in a table which uniquely identifies each row of a another table.
- That is, this field points to primary key of another table.
- This usually creates a kind of link between the tables.

- The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

Consider the two tables as shown below:

Orders Table :

O_ID	ORDER_NO	C_ID
1	2253	3
2	3325	3
3	4521	2
4	8532	1

Customers Table :

C_ID	NAME	ADDRESS
1	RAMESH	DELHI
2	SURESH	NOIDA
3	DHARMESH	GURGAON

- As we can see clearly that the field C_ID in Orders table is the primary key in Customers table, i.e. it uniquely identifies each row in the Customers table.
- Therefore, it is a Foreign Key in Orders table.

Using FOREIGN KEY constraint at Table Level

```
CREATE TABLE Orders (       Order_ID int NOT NULL,ORDER_NO int NOT NULL,
                            C_ID int,
                            PRIMARY KEY (O_ID),
                            FOREIGN KEY (C_ID) REFERENCES Customers(C_ID)
                    )
```

Using FOREIGN KEY constraint at Column Level

```
ALTER table Order  ADD FOREIGN KEY (c_id) REFERENCES Customer_Detail(c_id);
```

On Delete Cascade

- ON DELETE CASCADE clause in is used to automatically **remove** the matching records from the child table when we delete the rows from the parent table.
- It is a kind of referential action related to the **foreign key**.
- Suppose we have created two tables with a FOREIGN KEY in a foreign key relationship, making both tables a parent and child.
- Next, we define an ON DELETE CASCADE clause for one FOREIGN KEY that must be set for the other to succeed in the cascading operations.
- If the ON DELETE CASCADE is defined for one FOREIGN KEY clause only, then cascading operations will throw an error.

What is On Delete Cascade clause of foreign key?

On Delete Cascade as the name suggests deletes the dependent column entry in child table when there is any attempt of deleting the corresponding value in Parent table.

You can define foreign key with ON DELETE CASCADE clause either using create table or using alter table statement.

Example of On Delete Cascade Clause

We will again define two tables with the name as Authors and Books.
Authors will be our parent table with two columns author_id and author_name.

We will use author_id column as reference column for our foreign key constraint thus it's mandatory for us to define this column either as primary key or unique key.
Please read about Foreign Key for more information.

Let's create our parent table Authors –

```
CREATE TABLE author
(
PRIMARY KEY,    athr_aid_pk    CONSTRAINT    NUMBER(3)    author_id
VARCHAR2(30)    author_name
);
```

Read How To Define Primary Key Using Create Table

Now we will create our child table called Books. Child table books will consist of 3 columns – book_id, book_title and book_author_id. We will define foreign key on book_author_id column.

```
CREATE TABLE books
(
NUMBER(3),    book_id
VARCHAR2(30),    book_title
NUMBER(3),    book_price
 ON DELETE CASCADE          author(author_id)    REFERENCES    bok_ai_fk    CONSTRAINT
NUMBER(3)     book_author_id
);
```

Here we have our child table. In this table the column book_author_id will serve as the foreign key.
If you will see the foreign key definition of this column then you will notice that at the end of the foreign key definition we specified our clause which is "On Delete Cascade".

This is the concept behind On Delete Cascade clause in brief. You can watch my tutorial on the same for some practical examples.

Also, please do share it with your friends and help me spread the word. Thank you & have a great day!

Types of keys in DBMS

1. **Primary Key** – A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table.
2. **Super Key** – A super key is a set of one of more columns (attributes) to uniquely identify rows in a table.
3. **Candidate Key** – A super key with no redundant attribute is known as candidate key
4. **Alternate Key** – Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys.
5. **Composite Key** – A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.
6. **Foreign Key** – Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

5.3 SQL Functions :

- For doing operations on data sql has many built-in functions, they are categorised in two categories and further sub-categorised in different seven functions under each category.
- The categories are:

1. **Aggregate functions:**
   These functions are used to do operations from the values of the column and a single value is returned.

1. AVG()

- Average returns average value after calculating it from values in a numeric column.

```
Syntax :        SELECT AVG(column_name) FROM table_name
```

Consider the following Emp table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

```
QUERY :         SELECT avg(salary) from Emp;
```

Result of the above query will be,

avg(salary)
8200

. COUNT()

- Count returns the number of rows present in the table either based on some condition or without condition.

```
Syntax :        SELECT COUNT(column_name) FROM table-name
```

Consider the Emp table as in count() function :

```
SELECT COUNT(name) FROM Emp WHERE salary = 8000;
```

Result of the above query will be,

count(name)
2

. LAST()

- LAST function returns the return last value of the selected column.

```
Syntax :        SELECT LAST(column_name) FROM table-name
```

Consider the Emp table

```
SELECT LAST(salary) FROM emp;
```

Result of the above query will be,

last(salary)
8000

4. FIRST()

- First function returns first value of a selected column

```
Syntax :        SELECT FIRST(column_name) FROM table-name;
```

Consider the Emp table

```
SELECT FIRST(salary) FROM Emp;
```

and the result will be,

first(salary)
9000

5. MAX()

- MAX function returns maximum value from selected column of the table.

```
Syntax:         SELECT MAX(column_name) from table-name;
```

Consider the Emp table

```
SELECT MAX(salary) FROM emp;
```

Result of the above query will be,

MAX(salary)
10000

6. MIN()

- MIN function returns minimum value from a selected column of the table.

```
Syntax :        SELECT MIN(column_name) from table-name;
```

Consider the following Emp table,

```
SELECT MIN(salary) FROM emp;
```

Result will be,

MIN(salary)
6000

7. SUM()

- SUM function returns total sum of a selected columns numeric values.

```
Syntax:         SELECT SUM(column_name) from table-name;
```

Consider the following Emp table

```
SELECT SUM(salary) FROM emp;
```

Result of above query is,

SUM(salary)
41000

2. Scalar functions:

- Scalar functions return a single value from an input value.
- Following are some frequently used Scalar Functions in SQL.
- These functions are based on user input, these too returns single value.

1. UCASE()

- UCASE function is used to convert value of string column to Uppercase characters.

```
Syntax :        SELECT UCASE(column_name) from table-name;
```

Consider the following Emp table

eid	name	age	salary
401	anu	22	9000
402	shane	29	8000
403	rohan	34	6000
404	scott	44	10000
405	Tiger	35	8000

```
SELECT UCASE(name) FROM emp;
```

Result is,

UCASE(name)
ANU
SHANE
ROHAN
SCOTT
TIGER

2. LCASE()

- LCASE function is used to convert value of string columns to Lowecase characters.

```
Syntax  :       SELECT LCASE(column_name) FROM table-name;
```

Consider the following Emp table

```
SELECT LCASE(name) FROM emp;
```

Result will be,

LCASE(name)
anu
shane
rohan
scott
tiger

3. MID()

- MID function is used to extract substrings from column values of string type in a table.

```
Syntax:        SELECT MID(column_name, start, length) from table-name;
```

Consider the following Emp table

```
SELECT MID(name,2,2) FROM emp;
```

Result will come out to be,

MID(name,2,2)
nu
ha
oh
co
ig

4. ROUND()

- ROUND function is used to round a numeric field to number of nearest integer. It is used on Decimal point values.

```
Syntax :       SELECT ROUND(column_name, decimals) from table-name;
```

Consider the following Emp table

```
SELECT ROUND(salary) from emp;
```

Result will be,

ROUND(salary)
9001
8001
6000

10000
8000

Let's take another table

Students-Table

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

5. LEN()

- The LEN() function returns the length of the value in a text field.

```
Syntax :        SELECT LENGTH(column_name) FROM table_name;
```

Queries:

Fetching length of names of students from Students table.

```
SELECT LENGTH(NAME) FROM Students;
```

Result:

NAME
5
6
6
7
3

6. NOW()

- The NOW() function returns the current system date and time.

```
Syntax:        SELECT NOW() FROM table_name;
```

Queries:

Fetching current system time.

```
SELECT NAME, NOW() AS DateTime FROM Students;
```

Result:

NAME	DateTime
HARSH	**1/13/2017 1:30:11 PM**
SURESH	**1/13/2017 1:30:11 PM**
PRATIK	**1/13/2017 1:30:11 PM**
DHANRAJ	**1/13/2017 1:30:11 PM**
RAM	**1/13/2017 1:30:11 PM**

7. FORMAT()

- The FORMAT() function is used to format how a field is to be displayed.

```
Syntax:        SELECT FORMAT(column_name,format) FROM table_name;
```

Queries:

Formatting current date as 'YYYY-MM-DD'.

```
SELECT NAME, FORMAT(Now(),'YYYY-MM-DD') AS Date FROM Students;
```

Result:

NAME	Date
HARSH	2017-01-13
SURESH	2017-01-13
PRATIK	2017-01-13
DHANRAJ	2017-01-13
RAM	2017-01-13

5.4 Creating sequence

- In Oracle, you can create an auto number field by using sequences.
- A sequence is an object in Oracle that is used to generate a number sequence.
- This can be useful when you need to create a unique number to act as a primary key.
- The value can have maximum of 38 digits.
- A sequence can be defined to:
    - Generate numbers in ascending or descending order.
    - Provide intervals between numbers.
- Caching of sequence numbers in memory to speed up their availability.
- A sequence is an independent object and can be used with any table that requires its output:
- Creating Sequences
- Minimum information required for generating number using a sequence is:
- The starting number
- The maximum number that can be generated by a sequence.
- The increment value for generating the next number.

```
Syntax:

CREATE SEQUENCE sequence_name
[ INCREMENT BY <IntegerValue>
START WITH <IntegerValue>
MINVALUE <IntegerValue> / NOMINVALUE
MAXVALUE <IntegerValue> / NOMAXVALUE
CYCLE/NOCYCLE
CACHE <IntegerValue> / NOCACHE
ORDER/NOORDER ]
```

Here,

- **Sequence** is always given a name so that it can be referenced later when required.
- **INCREMENT BY :** Specifies interval between sequence numbers. Can be positive or negative value but not zero. If omitted the default value is 1.
- **START WITH :** Specifies first sequence number. Default for ascending sequence is 1 and descending sequence is –1.

- **MINVALUE:** Specifies the sequence minimum value.
- **NOMINVALUE**: Specifies 1 for an ascending sequence and -10^26 for a descending sequence.
- **MAXVALUE:** Specifies the sequence maximum value.
- **NOMAXVALUE:** Specifies maximum of a 10^27 for an ascending sequence and –1 for a descending sequence.
- **CYCLE:** Specifies sequence continues to generate repeat values after reaching either its maximum value.
- **NOCYCLE:** Specifies sequence cannot generate more values after reaching the maximum value.
- **CACHE:** Specifies how many values of a sequence Oracle preallocates and keeps in memory for faster access. The minimum value for this is two.
- **NOCACHE**: Specifies that values of a sequence are not pre-allocated.
- **ORDER:** Guarantees that sequence number are generated in the order of request.Only necessary if using Parallel server in parallel mode option. In exclusive mode option a sequence always generates numbers in order.
- **NOORDER :** By default. Does not guarantee that sequence number are generated in the order of request. Only necessary if using Parallel server in parallel mode option.

- Example :

```
SQL> CREATE SEQUENCE my_seq INCREMENT BY 1 START WITH 1 MINVALUE 1
MAXVALUE 999 CYCLE;
```

Sequence created.

```
SQL> SELECT my_seq.NEXTVAL FROM DUAL;
```

NEXTVAL

1
It DISPLAYS THE NEXTVALUE HELD IN CACHE.

- Following example explains how to access a sequence and its generated value in the INSERT statement.

```
SQL> INSERT INTO emp VALUES (myseq.nextval,'JACK','MANAGER',2850,40)
```

1 row created.

```
SQL> SELECT my_seq.CURRVAL FROM DUAL;
```

CURRVAL

1

- IT DISPLAYS THE CURRENT VALUE IN THE SEQUENCE.

```
SQL> SELECT my_seq.NEXTVAL FROM DUAL;
```

NEXTVAL

2

ALTERING A Sequence:

- A Sequence once created can be altered by following syntax:

```
ALTER SEQUENCE sequence_name
[ INCREMENT BY <IntegerValue>
MINVALUE <IntegerValue> / NOMINVALUE
MAXVALUE <IntegerValue> / NOMAXVALUE
CYCLE/NOCYCLE
CACHE <IntegerValue> / NOCACHE
ORDER/NOORDER ]
```

The start value of a sequence cannot be altered.

- Example:

```
SQL> ALTER SEQUENCE my_seq INCREMENT BY 2 CACHE 30;
```

- Sequence altered

```
SQL> SELECT my_seq.NEXTVAL FROM DUAL;
```

NEXTVAL

4

DROPPING A Sequence:

- A sequence can be dropped as follows:

```
DROP SEQUENCE <SequenceName>;
```

- Example:

```
SQL> DROP SEQUENCE my_seq;
```

Sequence dropped.

5.5 Views :

- In SQL, a view is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.
- The table on which view is based is described in FROM clause of SELECT statement
- and the table is called the base table.
- View is created on top of the base table. Thus redundant data will not occupy disk storage.
- It can be queried exactly like queried on a base table.
- But query fired on view will run slowly as compared to base table.
- Some views are Read-Only View used only for looking at table data.
- Others are called Updateable View can be used to Insert, Update or Delete table as
- well as view data.

Why views are created.

- when data security is required.
- when data redundancy is to be kept to the minimum while maintaining data security.

5.5.1 Creating simple view

Syntax

```
CREATE [OR REPLACE] VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY <Grouping criteria> HAVING <Predicate>
```

OR REPLACE keyword use to re-create the view if it already exists.
You can use this clause to change the definition of an existing view without dropping, re-creating and regranting object privileges previously granted on it.

Note: the columns of the table are related to the view using a **one-to-one relationship.**

SQL CREATE VIEW Examples
sample database selects every product in the "Products" table with a unit price higher than the average unit price:

```
CREATE VIEW V_EMP_AVGSAL AS
SELECT EMPNO,ENAME,JOB,SAL
FROM EMP
WHERE SAL>(SELECT AVG(SAL) FROM EMP);
```

We can query the view above as follows:

```
SELECT * FROM V_EMP_AVGSAL
```

SELECTING DATA SET FROM A VIEW
Once view has been created it can be queried exactly like a base table

Syntax:

```
SELECT COL1,COL2,…. FROM <ViewName>
```

Now you can query CUSTOMERS_VIEW in similar way as you query

```
CREATE OR REPLACE VIEW V_EMP AS
SELECT EMPNO,ENAME, JOB,SAL;
```

```
SQL > SELECT * FROM V_EMP;
```

This would produce the following result:

EMPNO	ENAME	JOB	SAL
e1	rohit	MANAGER	90000
e2	jaid	EXECUTIVE	9000
e3	rohan	DIRECTOR	100000
e4	raj	MANAGER	130000
e6	krupali	MANAGER	330000

Updateable Views:

- View can be used for DML (ie the user can perform the insert,update and delete operation)
- View on which data manipulation can be done are called updateable views.
- When an updateable view name is given in an DML statement, modifications to data in the view will be immediately passed to the underlying table(base table).

View to be updateable, it should meet following criterion:

- View defined from **single table**
- If user wants to INSERT with the help of view, then PRIMARY KEY column(s)and all NOT NULL columns must be included in the view.

- User can UPDATE or DELETE records with the help of a view even if the PRIMARY KEY column and NOT NULL column(s) are excluded from view definition.

INSERT USING VIEWS created on single table

```
Sql> create table t1(name varchar2(10), id number);
```

Table created.

```
Sql> select * from t1;
```

no rows selected

```
Sql> create view t1_view as select * from t1;
```

View created.

```
Sql> insert into t1_view values('a',1);
```

1 row created.

```
Sql> select * from t1;
Or
Sql> select * from t1_view;
```

NAME	ID
a	1

UPDATE USING VIEWS created on single table

```
SQL> update t1_view set name='ajay' where id=1;
```

1 row updated.

```
Sql> select * from t1;
Or
Sql> select * from t1_view;
```

NAME	ID
ajay	1

DELETE USING VIEWS created on single table

```
SQL> delete from t1_view where id=1;
```

1 row deleted.

```
Sql> select * from t1;
Or
Sql> select * from t1_view;
```

no rows selected

FOR VIEWS CREATED FROM MULTIPLE TABLE

- A view can be created from more than one table.
- These tables will be linked by a join specified in the WHERE clause of the View definition.
- The behavior of view will vary in operations such as INSERT, UPDATE and DELETE depending on the following
  - → Whether tables were created using a Referencing clause
  - → Whether tables were created without any referencing clause and are actually standalone tables not related with one another in any way

Views Defined From Multiple Tables (Which have No Referencing Clause)

- If view is created from multiple tables which were not created using any referencing clause then though the PRIMARY Key column(s) as well as NOT NULL columns are included in the View definition the view's behavior will be as follow:

- The INSERT, UPDATE or DELETE OPERATION is not allowed.
- If attempted Oracle displays an error message.
- **For INSERT/UPDATE**
  ERROR at line 1:
  ORA-01779: cannot modify a column which maps to a non key-preserved table
- **For DELETE**
  ORA-01752: cannot delete from view without exactly one key-preserved table

Views Defined From Multiple Tables (Created With a Referencing Clause)

- If view is created from multiple tables which were created using any referencing clause then though the PRIMARY Key column(s) as well as NOT NULL columns are included in the View definition, the view will behave as follows:
  - → An **INSERT** operation is not allowed.
  - → A **DELETE or UPDATE** operations do not affect the Master table.
  - → The view can be used to **MODIFY** the columns of the detail table included in the view.

Example

Consider following tables:

Table Name :EMP

NAME	NULL?	TYPE
EID		NUMBER
ENAME		VARCHAR2(20)
JOB		VARCHAR2(20)
SAL		NUMBER
DEPT NO		NUMBER

Table Name : DEPT

NAME	NULL?	TYPE
DEPTNO	NOT NULL	NUMBER
DEPTNAME		VARCHAR2(14)
LOCATION		VARCHAR2(13)

In the above tables there is no referencing clause defined. If view is created as follows:

```
CREATE OR REPLACE VIEW MYVIEW
AS SELECT E.EID,E.ENAME,E.SAL,D.DEPTNO,D.LOC
FROM TEMPEMP E, TEMPDEPT D WHERE E.DEPTNO=D.DEPTNO;
```

- If either of following operation is performed the Oracle will give output as follows:

```
SQL> insert into myview values(7888,'daisy',1800,10,'chicago');
```

ERROR at line 1:
ORA-01776: cannot modify more than one base table through a join view

```
SQL>update myview set SAL=40000 where deptno=10;
```

4 rows updated

```
SQL> delete from myview where eid=7369;
```

1 row deleted
The above delete and modify operation also affects the base table.

Common Restrictions On Updateable Views
For a view to be updateable the view definition must not include:

- Aggregate functions.
- DISTINCT, GROUP BY or HAVING clause
- Sub-queries
- Constants, Strings or Value Expressions like SAL * 10
- UNION, INTERSECT or MINUS clause
- If a view is defined from another view, the second view should be updateable

if the user tries to perform any of INSERT, UPDATE, DELETE operation , on view , which is created from a non-updateable view oracle returns the following error message.

- **For INSERT/UPDATE/DELETE**
  ORA-01732: Data manipulation operation not legal on this view.

Dropping view.

You can delete a view with the DROP VIEW command.
Syntax

```
DROP VIEW view_name
```

Example

```
DROP VIEW v_empavgsal;
```

5.5.2 Difference between View and Table

View	Table
A view is a database object that allows generating a logical subset of data from one or more tables	table is a database object or an entity that stores the data of a database
The view is a virtual table that is extracted from a database.	Table is an actual table.
view depends on the table	Table is an independent data object.
The view is utilized to query certain information which is contained in a few distinct tables	The table holds fundamental client information and holds cases of a characterized object.
In the view, you will get frequently queried information.	In the table, changing the information in the database likewise changes the information appeared in the view which isn't the