

SUMMARY: BASIC DATABASE OPERATIONS-II

SESSION OVERVIEW:

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

- Understand how to use logical and comparison operators in the WHERE clause.
- Understand the WILDCARDS in the WHERE clause.
- Understand the DISTINCT clause.
- Understand the concepts of ORDER BY.
- Understand the concepts of LIMIT and OFFSET.
- Understand commenting for documentation in SQL.
- Understanding the column alias.

KEY TOPICS AND EXAMPLES:

NOTE:

- The outputs attached below are just sample outputs and do not indicate that the query returns only these many rows.
- All the queries mentioned in this document represent the Customers table. The dataset has been provided in the session 2 document. Please refer to session 2 to get the dataset. Import the dataset in your MySQL Workbench to get efficient exposure to the session.

Understanding the logical and relational operators in WHERE clause:

Comparison Operators:

Comparison operators in SQL are used to compare two values in a query, enabling filtering and specific data retrieval based on conditions. They are essential in the WHERE clause and other parts of SQL statements like JOIN conditions and HAVING clauses. (JOINs and HAVING clauses will be discussed in the upcoming sessions)

OPERATOR	DESCRIPTION		
=	Equal to		
>	Greater Than		
<	Less than		
>=	Greater than or equal to		
<=	Less than or equal to		
<> or !=	Not equal to		

NOTE: In the previous session we understood the WHERE clause using the "=" comparison operator. In this session, we will continue with the remaining comparison and logical operators.

Recalling the syntax of WHERE Clause: Syntax:

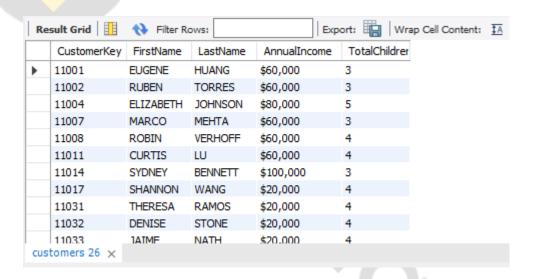


```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example 1:

```
# Query to find the customers who have more than 2 children.
Select CustomerKey, FirstName, LastName, AnnualIncome, TotalChildren
From customers
Where TotalChildren > 2;
```

Output:

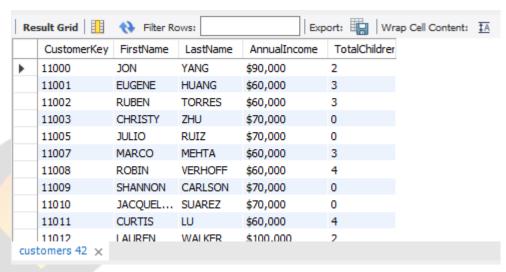


Example 2:

```
# Query to find the customers with the total number of children not
equal to 5.

SELECT CustomerKey, FirstName, LastName, AnnualIncome, TotalChildren
FROM customers
WHERE CAST(TotalChildren AS INT) <> 5;
```





NOTE: The above query returns the specific columns mentioned along with the filter applied using the WHERE clause. The returned table will consist of columns like CustomerKey, FirstName, LastName, AnnualIncome, and TotalChildren where the Total number of children is not equal to 5.

Logical Operators:

Logical operators in SQL are used to combine two or more conditions in a WHERE clause, allowing you to perform more complex queries by linking conditions on database columns. These operators play a crucial role in filtering data based on specific criteria.

OPERATORS	DESCRIPTION		
AND	TRUE if all the conditions separated by AND are TRUE.		
OR	TRUE if any of the conditions separated by OR is TRUE.		
NOT	Displays a record if the condition(s) is NOT TRUE.		
LIKE	TRUE if the operand matches a pattern		
IN	TRUE if the operand is equal to one of a list of expressions		
BETWEEN	TRUE if the operand is within the range of comparisons		

Syntax 1: (AND operator)

```
SELECT column1, column2, ...
FROM tableName
WHERE condition1 AND condition2;
```

NOTE:

- In the above syntax first, we specify the columns which we desire to return, then we specify the table name from which we want to return and lastly, we specify the conditions using the desired operator in between.
- We can add multiple conditions in the WHERE clause.

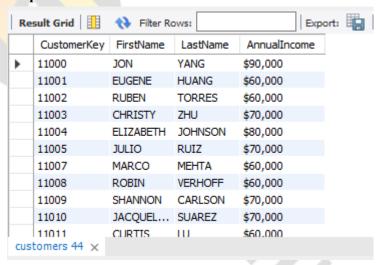


Example 1:

```
# Query to find the customers whose education level is Bachelors and
Occupation is Professional.

Select CustomerKey, FirstName, LastName, AnnualIncome
From customers
Where EducationLevel="Bachelors" AND Occupation="Professional";
```

Output:



Note: The above query returns the specific columns mentioned along with the filter applied using the WHERE clause. The returned table will consist of columns like CustomerKey, FirstName, LastName, AnnualIncome where the Education level of the customers is Bachelors and the occupation is professional.

Example 2:

```
# Query to find the customers who are married and do not have any kids.
Select CustomerKey, FirstName, LastName, AnnualIncome, TotalChildren
From customers
Where MaritalStatus="M" and TotalChildren=0;
```





NOTE: The above query returns the specific columns mentioned along with the filter applied using the WHERE clause. The returned table will consist of columns like CustomerKey, FirstName, LastName, AnnualIncome, and TotalChildren where the customer is married and has 0 children.

Syntax 2: (OR)

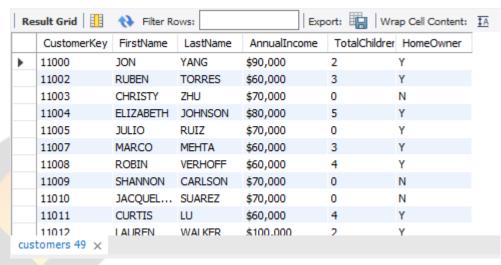
```
SELECT column1, column2, ...
FROM tableName
WHERE condition1 OR condition2;
```

NOTE:

- In the above syntax first, we specify the columns which we desire to return, then we specify the table name from which we want to return and lastly, we specify the conditions using the desired operator in between.
- We can add multiple conditions in the WHERE clause using the OR operator.

Example:

```
# Query to find the customers with 0 children and are homeowners.
Select CustomerKey, FirstName, LastName, AnnualIncome, TotalChildren,
HomeOwner
From customers
Where TotalChildren="0" OR HomeOwner="Y";
```



NOTE: The above query returns the specific columns mentioned along with the filter applied using the WHERE clause. The returned table will consist of columns like CustomerKey, FirstName, LastName, AnnualIncome, and TotalChildren where the customer is married and has 0 children.

Syntax 3: (NOT)

```
SELECT column1, column2, ...
FROM tableName
WHERE NOT condition;
```

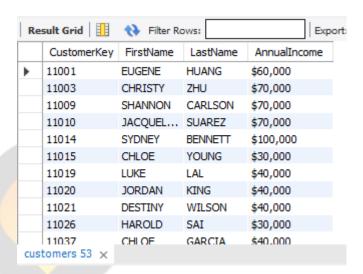
NOTE:

- In the above syntax first, we specify the columns which we desire to return, then we specify the table name from which we want to return and lastly, we specify the conditions using the desired operator.
- NOT defines the condition that must not be met for the rows to be included in the result set. This negates whatever condition follows the NOT.

Example:

```
# Query to find
Select CustomerKey, FirstName, LastName, AnnualIncome
From customers
Where NOT HomeOwner="Y";
```





Understanding WILDCARDS in SQL

NOTE: WILDCARDS are majorly used in corresponding to the LIKE logical operators.

SYMBOL	DESCRIPTION		
%	Represents zero or more characters		
_	Represents a single character		
[]	Represents any single character within the brackets		

LIKE:

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used in conjunction with the LIKE operator:

- The percent sign % represents zero, one, or multiple characters
- The underscore sign represents one, single character

Syntax 4:

```
SELECT column1, column2, ...

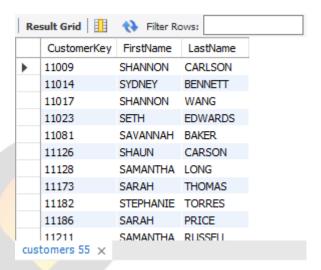
FROM tableName
WHERE columnname LIKE pattern;
```

Example 1:

```
# Query to find the customers whose names start with 'S'
SELECT CustomerKey, FirstName, LastName
From customers
where FirstName Like "S%";
```

NOTE: The above query returns the specific columns mentioned along with the filter applied using the WHERE clause. The returned table will consist of columns like CustomerKey, FirstName, and LastName where the customer's First Name starts with 'S'.



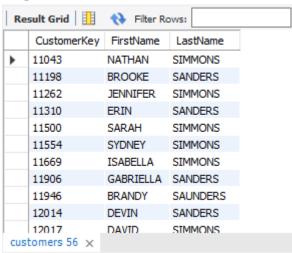


Example 2:

```
# Query to find the customers whose name starts with 'S' and ends with
'S'.
SELECT CustomerKey, FirstName, LastName
From customers
where LastName Like "S%S";
```

NOTE: The above query returns the specific columns mentioned along with the filter applied using the WHERE clause. The returned table will consist of columns like CustomerKey, FirstName, and LastName where the customer's Last Name starts with 'S' and ends with 'S' and contains one or multiple alphabets in between.

Output:



Example 3:

```
#Query to find the customers whose name consists of an 'A' in between
their names.
SELECT CustomerKey, FirstName, LastName
From customers
```



```
where FirstName Like "%A%";
```

NOTE: The above query returns the specific columns mentioned along with the filter applied using the WHERE clause. The returned table will consist of columns like CustomerKey, FirstName, and LastName where the customer's First Name has 'A' in between one or multiple alphabets.

Output:



Example 4:

Query to find the customers whose last name starts with 0 or multiple characters, have an 'A' in between followed by a single character, and end with 'G'.

```
SELECT CustomerKey, FirstName, LastName
From customers
where LastName Like "%A_G";
```

Output:



Example 5:



```
# Query to find customers whose names start and end with a vowel.

SELECT CustomerKey, FirstName, LastName
FROM customers
WHERE FirstName LIKE '[aeiou]%[aeiou]';
```

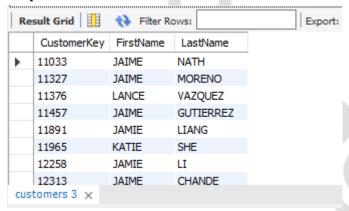
Output:

Example 6:

```
# Query to find customers whose first name has any letter followed by
'a', then any two letters, and ends with 'e'.

SELECT CustomerKey, FirstName, LastName
FROM customers
WHERE FirstName LIKE '_a_e';
```

Output:



IN:

The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

Syntax:

```
SELECT column1, column2, ...

FROM tableName

WHERE column IN (value1, value2, ...);
```

NOTE:

- In the above syntax first, we specify the columns which we desire to return, then we specify the table name from which we want to return and lastly, we specify the conditions using the desired operator.
- Using the IN operator we can in the WHERE clause we can add multiple conditions at a time.

Example 1:



```
# Query to find the customers with education level of Bachelor and
graduate degrees.

Select CustomerKey, FirstName, LastName, AnnualIncome
From customers
Where EducationLevel IN ('Bachelors', 'graduate degree');
```

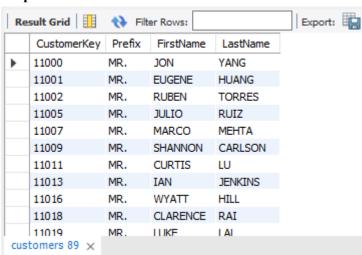
Output:



Example 2:

```
# Query to find the customers with MRS and MS prefixes.
SELECT CustomerKey, Prefix, FirstName, LastName
FROM customers
WHERE NOT prefix IN ('MRS.', 'MS.');
```

Output:



BETWEEN:



The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

Syntax:

```
SELECT column1, column2, ...
FROM tableName
WHERE column BETWEEN value1 AND value2;
```

Example:

```
# Query to find the customers with a total number of children between
0-2.
Select CustomerKey, FirstName, LastName, AnnualIncome, TotalChildren
From customers
Where TotalChildren between '0' and '2';
```

Output:

Result Grid 1					ort: Wrap Cell
	CustomerKey	FirstName	LastName	AnnualIncome	TotalChildrer
•	11000	JON	YANG	\$90,000	2
	11003	CHRISTY	ZHU	\$70,000	0
	11005	JULIO	RUIZ	\$70,000	0
	11009	SHANNON	CARLSON	\$70,000	0
	11010	JACQUEL	SUAREZ	\$70,000	0
	11012	LAUREN	WALKER	\$100,000	2
	11013	IAN	JENKINS	\$100,000	2
	11015	CHLOE	YOUNG	\$30,000	0
	11016	WYATT	HILL	\$30,000	0
	11018	CLARENCE	RAI	\$30,000	2
	11019	LUKE	I AI	\$40.000	0
cus	tomers 80 ×				

Understanding the DISTINCT clause:

The DISTINCT keyword in SQL is used to eliminate duplicate rows from the result set and return only unique instances of a particular column or set of columns. This keyword can be particularly useful when you want to count or identify different items in a database, such as counting the number of unique customer names or identifying different products sold.

Syntax:

```
SELECT DISTINCT column1, column2, ...
FROM tableName
WHERE condition;
```

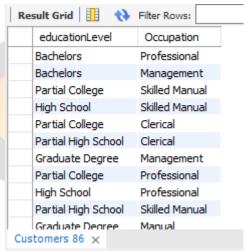
Example 1:

```
# Query to find the unique Income of the customers.
```



Select distinct EducationLevel, Occupation From Customers

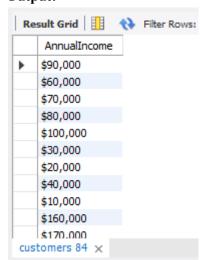
Output:



Example 2:

Query to find the unique Income of the customers.
Select distinct AnnualIncome
From customers

Output:



Example 3:

Query to find the unique annual income of the customers.
SELECT DISTINCT AnnualIncome
FROM Customers



Understanding the concepts in order by:

The ORDER BY clause in MySQL is used to sort the result set of a query by one or more columns. It helps in organizing the output in either ascending or descending order. The sorting can be based on numerical values, text values, or dates, and can involve multiple columns with different sorting directions.

Key Concepts:

- Sorting Columns: You can sort the result set by one or more columns.
- **Order Direction:** You can specify the sort direction as ascending (ASC) or descending (DESC).
- **Multiple Columns:** You can sort by multiple columns, specifying different sort directions for each.
- **Default Order:** The default order is ascending if no direction is specified.

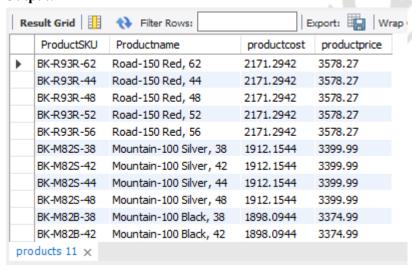
Basic syntax:

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

Example 1:

```
Select ProductSKU, Productname, productcost, productprice
from products
order by productcost desc;
```

Output:



Understanding the concepts of limit and offset:

In MySQL, the **LIMIT clause** is used to specify the number of rows to return in the result set. It is often used with the SELECT statement to control the number of rows returned from a query. This is



particularly useful for implementing pagination or for simply limiting the results of a query to a manageable size.

Basic Syntax:

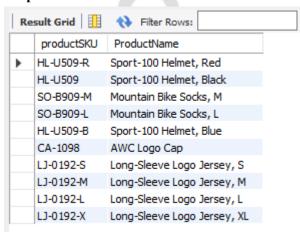
```
SELECT column1, column2, ...

FROM table_name
LIMIT number;
```

Example 1:

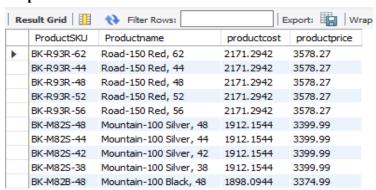
```
select productSKU, ProductName
from products
limit 10;
```

Output:



Example 2:

```
Select ProductSKU, Productname, productcost, productprice
from products
order by productprice desc
limit 10;
```





The **OFFSET clause** in MySQL is used to skip a specified number of rows in the result set before beginning to return the rows. It's particularly useful for implementing pagination, where you want to retrieve a subset of rows from a larger dataset, often in combination with the LIMIT clause.

Key Concepts of OFFSET in MySQL:

- **Skipping Rows**: OFFSET specifies how many rows to skip in the result set.
- Combining with LIMIT: Often used with LIMIT to return a specific subset of rows.
- **Zero-based Indexing:** OFFSET is zero-based, meaning OFFSET 0 starts from the first row.

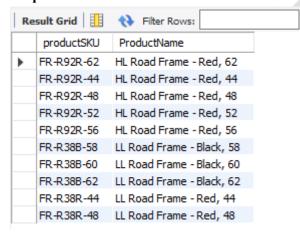
Basic Syntax:

```
SELECT column1, column2, ...
FROM table_name
LIMIT number OFFSET offset;
```

Example 1:

```
select * from products;
select productSKU, ProductName
from products
limit 10 offset 10;
```

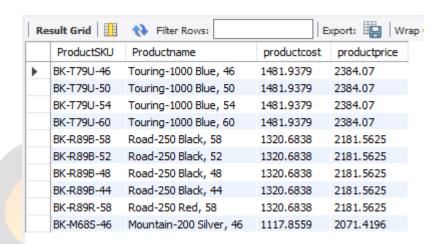
Output:



Example 2:

```
Select ProductSKU, Productname, productcost, productprice from products order by productprice desc limit 10 offset 20;
```





Understand commenting for documentation in SQL:

In SQL, comments are used to include explanatory notes or to temporarily disable parts of SQL code. Comments can help others understand your code, or remind you of what your code does, especially in complex SQL statements or scripts. SQL supports single-line and multi-line comments, and the syntax can vary slightly depending on the SQL database system (like MySQL, SQL Server, Oracle, PostgreSQL, etc.).

Single-line Comments:

Single-line comments start with two consecutive dashes (--). Everything following the -- on the same line is treated as a comment and is not executed.

Syntax and example:

```
-- This is a single-line comment

SELECT * FROM Customers; -- This is a comment after the code
```

NOTE: The comments are not executed and it is mentioned to maintain the flow of the document. It is mentioned to get a better understanding of the query and for documentation purposes.

Multi-line Comments:

Multi-line comments are enclosed between /* and */. Everything within the /* and */ markers is treated as a comment, regardless of how many lines the text spans. This type of comment is useful for temporarily disabling blocks of SQL code or for adding longer descriptions or explanations within SQL scripts.

Syntax and example:

```
/* This is a multi-line comment
  and it can span multiple lines. */
SELECT * FROM Customers;
```



```
/*
SELECT * FROM Customer WHERE CustomerKey = 11008;
This part of the code has been commented out and won't execute.
*/
```

NOTE: The comments are not executed and it is mentioned to maintain the flow of the document. It is mentioned to get a better understanding of the query and for documentation purposes.

Understanding the column alias:

In SQL, a column alias is a temporary name assigned to a column in the output of a query. Column aliases are used to make the output of SQL queries more readable or to format the column headers in a result set, which can be particularly useful when the original column names are not descriptive or too lengthy. They can also be employed when deriving new data from existing columns.

Benefits of Using Column Aliases:

- Readability and Clarity: Aliases can make SQL queries and their output more understandable, which is helpful when sharing code with others or when you need to maintain it over time.
- Necessity in Calculations: When you perform operations or calculations in a SELECT statement, the use of an alias can provide a meaningful name for the derived column, rather than leaving it unnamed.
- **Handling Ambiguity:** In queries involving multiple tables where some column names might be the same across tables, aliases help distinguish these columns within the result set.

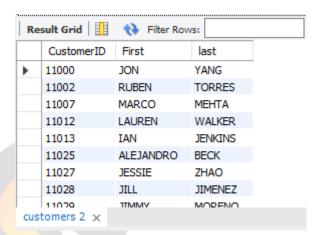
Syntax:

```
SELECT column_name AS alias_name
FROM table_name;
```

Example 1:

```
# Query to find the customers who are married and have 2 and 3 children.
(Usage of column alias).
Select CustomerKey as CustomerID, firstName as First, LastName as last
From customers
where MaritalStatus="M" and TotalChildren in (2, 3);
```





Example 2:

```
# Query that helps us to use the column alias.
select c. customerKey as customerID, c. BirthDate as DOB
from customers as c
where gender="F";
```

