SCALER

SQL Revision notes

Relational Database

- It is a collection of interrelated tables. The data within these tables have relationships with one another. Example: Microsoft SQL Server, MySQL, etc.
- A relational database is structured.

Non-relational Database

- It is a kind of database that doesn't use tables, fields, and columns of structured data. Examples: MongoDB, Apache Cassandra, etc.
- A non-relational database is semi-structured or unstructured.

Management system

- A set of operations that help us to manage the database.
- Some of these operations are called CRUD operations.
    - C - create
    - R - read
    - U - update
    - D - delete

Types of relationships in RDBMS

There are three types of relationships that can be present between tables:

- **One-to-one** relationship occurs when each row in Table 1 has only one related row in Table 2.
- **One-to-many** occurs when one record in Table 1 is related to one or more records in Table 2.
- **Many-to-many** occurs when multiple records in one table are related to multiple records in another table.

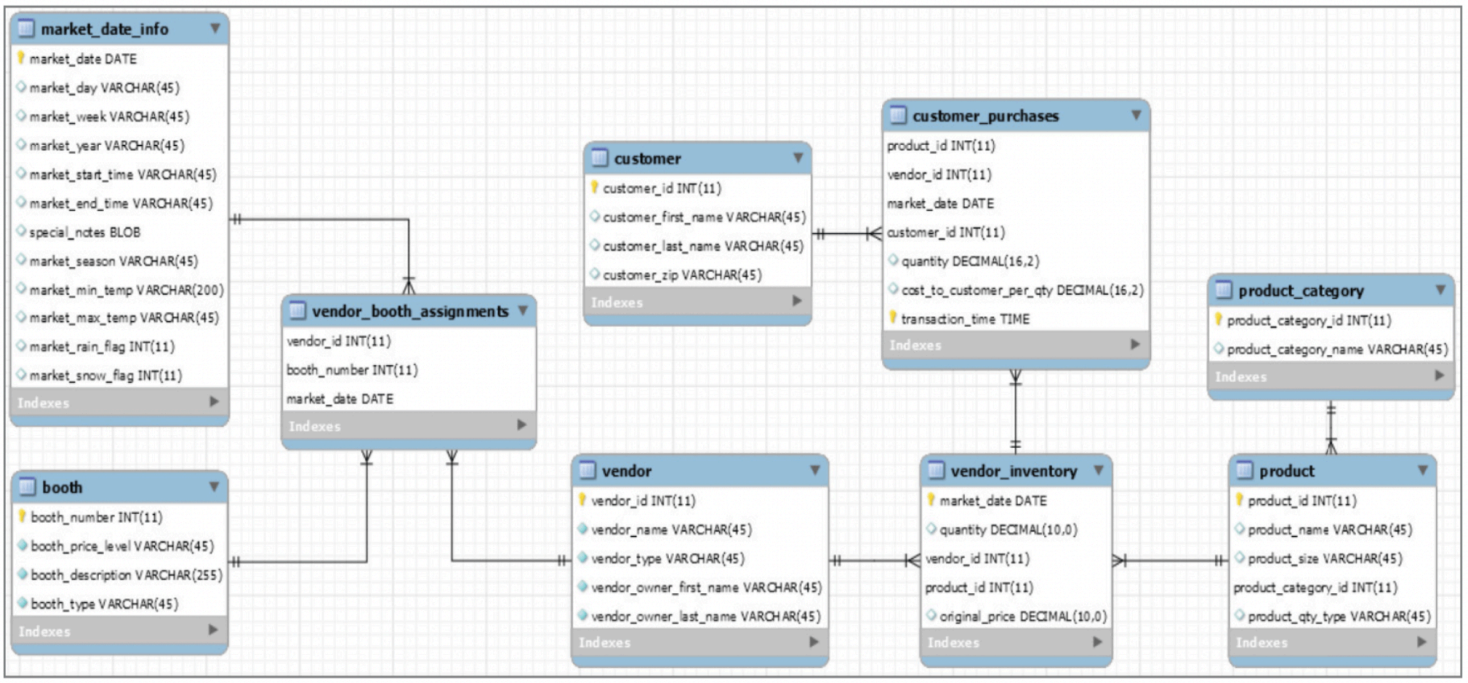
Primary key: A primary key is a field in a table that uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

Foreign Key: A foreign key is a column or a combination of columns whose values match a Primary Key in a different table. It is a column used to link two tables together.

Database schema

- A database schema serves as a blueprint for the shape and format of data within a database. This includes describing categories of data and their connections through tables, primary keys, data types, and other elements.

- For example, given below is a schema of a **Farmer's Market(Mandi)**.



- **Some observations from the above schema**
    - Each entity provides some details about the tables in the database. E.g. table name, its attributes(columns), datatype, etc.
    - There exists a many-to-many relationship between the *product* table and the *vendor* table. One vendor can bring many products to the market and one product can be sold by many vendors. A many-to-many relationship would always have a **junction table**.
    - The above schema is also called the **Entity relationship diagram**.

Datatypes

There are three main types of datatypes in RDBMS: Numeric, string, and date and time.
Some examples of these data types are:

String data type

- CHAR(n): A fixed-length string of size *n*.
- VARCHAR(n): A variable-length string with a maximum length *n*.

Numeric datatypes

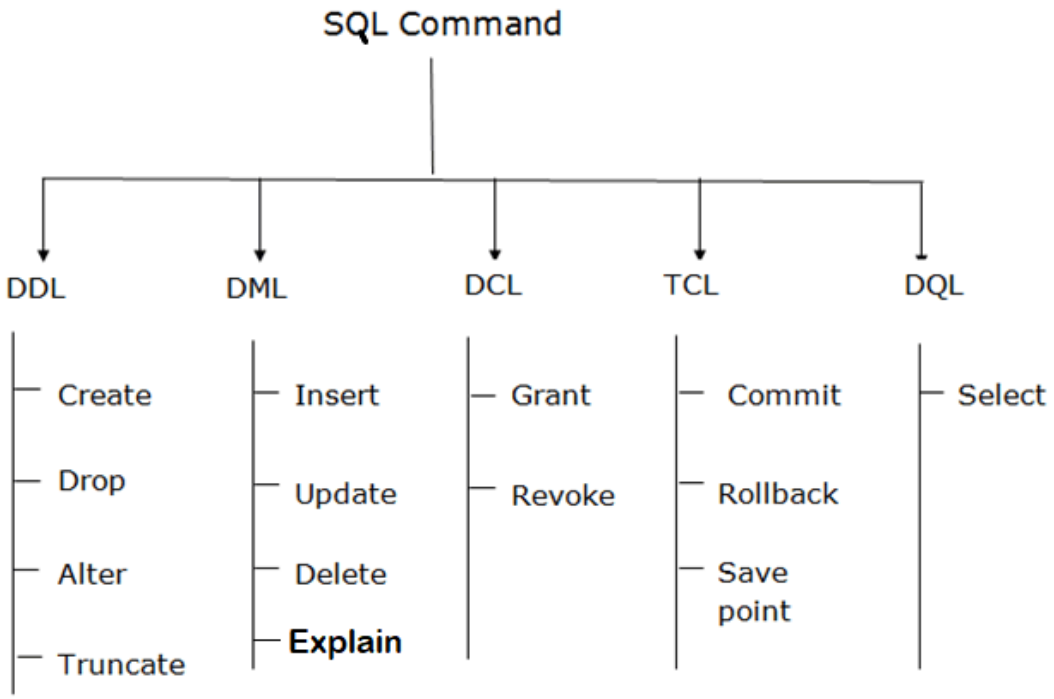
- INTEGER(size)/INT(size): A medium integer.
- FLOAT(): A floating point number.

Date and Time Data Types

- DATE: A date having the format: YYYY-MM-DD.
- DATETIME: A date and time combination having the format: YYYY-MM-DD hh:mm:ss.
- TIME: A time format: hh:mm:ss

SQL (Structured Query Language)

- SQL is a standard language that allows us to interact with relational databases in order to store, manipulate and retrieve the data in databases.
- There are different types of SQL commands:
    - DDL (Data definition language)
    - DML (Data manipulation language)
    - TCL (Transaction control language)
    - DCL (Data control language)
    - DQL (Data Query language)



SQL query syntax

```
SELECT [columns to return]
FROM [table name]
WHERE [conditional statement]
ORDER BY [columns to sort on]
LIMIT [no. of first n rows to be returned]
```

OFFSET

- OFFSET clause is used to find a starting point to display a set of rows as a final output by eliminating a set of records from a given table in order.

- It can be used with LIMIT as:

```
LIMIT [no_of_first_n_rows_to_be_returned] OFFSET [no_of_rows_to_skip]
```

- **Example**: Extract the third-highest and fourth-highest salaries from the following employee's table.

emp_id	emp_name	job_name	manager_id	hire_date	salary	commission	dep_id
68319	KAYLING	PRESIDENT		1991-11-18	6000.00		1001
66928	BLAZE	MANAGER	68319	1991-05-01	2750.00		3001
67832	CLARE	MANAGER	68319	1991-06-09	2550.00		1001
65646	JONAS	MANAGER	68319	1991-04-02	2957.00		2001
67858	SCARLET	ANALYST	65646	1997-04-19	3100.00		2001
69062	FRANK	ANALYST	65646	1991-12-03	3100.00		2001
63679	SANDRINE	CLERK	69062	1990-12-18	900.00		2001
64989	ADELYN	SALESMAN	66928	1991-02-20	1700.00	400.00	3001
65271	WADE	SALESMAN	66928	1991-02-22	1350.00	600.00	3001

  Here, we first sort the data by the *salary* column and then skip the first two salaries using OFFSET to get the third-highest and fourth-highest salaries.

  Solution query:

```
SELECT salary
FROM employees
ORDER BY salary desc
LIMIT 2 OFFSET 2
```

  Output:

salary
3100.00
2957.00

ROUND() function

- This function rounds a number to a specified number of decimal places.
- Syntax: `ROUND(number, decimals)`

Aliases

- SQL aliases are used to give a table, or a column in a table, a temporary name. It is done using AS clause.

- Syntax to give a temporary name to a **table** is :

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

- Syntax to give a temporary name to a **column** is :

```
SELECT column_name AS alias_name
```

```
FROM table_name
WHERE [condition];
```

- **Example**: In the given *customer_purchases* table, create a new column 'total_amt' containing the total amount paid by each customer rounded off to two decimal places.

product_id	vendor_id	market_date	customer_id	quantity	cost_to_customer_per_qty	transaction_time
1	7	2019-07-03	14	0.99	6.99	17:32:00
1	7	2019-07-03	14	2.18	6.99	18:23:00
1	7	2019-07-03	15	1.53	6.99	18:41:00
1	7	2019-07-03	16	2.02	6.99	18:18:00

The new column *total_amt* is created using the AS clause after rounding the multiplication of *quantity* and *cost_to_customer_per_qty* columns to two decimal places using the ROUND() function.

Solution query:

```
SELECT
     ROUND(quantity * cost_to_customer_per_qty, 2) AS
     total_amt
from customer_purchases
```

Output:

total_amt
6.92
15.24
10.69
14.12
4.61
1.89

UPPER()

- This function takes a string as an argument and converts it into upper-case.
- Syntax: `UPPER(string)`

Note: similar to the UPPER() function, LOWER() converts a string into lower-case.

CONCAT()

- It is used to concatenate two or more strings together.
- Syntax: `CONCAT(string1, string2, string3, ….)`

- **Example:** Given the customer table, merge each customer's name into a single column that contains the first name, then a space, and then the last name in upper case.

customer_id	customer_first_name	customer_last_name	customer_zip
1	Jane	Connor	22801
2	Manuel	Diaz	22821
3	Bob	Wilson	22821
4	Deanna	Washington	22801
5	Abigail	Harris	22801
6	Betty	Bullard	22801

customer table

Solution query:

```
SELECT
     UPPER(CONCAT(customer_first_name," ",
customer_last_name)) AS full_name
FROM customer
```

Output:

full_name
JANE CONNOR
MANUEL DIAZ
BOB WILSON
DEANNA WASHINGTON
ABIGAIL HARRIS
BETTY BULLARD
JESSICA ARMENTA
NORMA VALENZUELA
JANET FORBES

WHERE

- The WHERE clause is used to filter the data based on one or more conditions.
- Multiple conditions are given for filtering using **Boolean operators** (AND, OR, etc).
- Syntax :

```
SELECT column1, column2, …
FROM table_name
WHERE condition
```

- **Example:** Given the following *product* table, get all the product info for products with ID between 3 and 8(inclusive) and for products with ID 10.

product_id	product_name	product_size	product_category_id	product_qty_type
1	Habanero Peppers - Organic	medium	1	lbs
2	Jalapeno Peppers - Organic	small	1	lbs
3	Poblano Peppers - Organic	large	1	unit
4	Banana Peppers - Jar	8 oz	3	unit
5	Whole Wheat Bread	1.5 lbs	3	unit
6	Cut Zinnias Bouquet	medium	5	unit
7	Apple Pie	10	3	unit

product table

Solution query:

```
SELECT *
FROM product
WHERE (product_category_id > 3 AND product_category_id <= 8) OR
(product_category_id = 10)
```

Output:

product_id	product_name	product_size	product_category_id	product_qty_type
6	Cut Zinnias Bouquet	medium	5	unit
10	Eggs	1 dozen	6	unit
11	Pork Chops	1 lb	6	lbs
19	Farmers Market Resuable Shopping Bag	medium	7	unit
20	Homemade Beeswax Candles	6	7	unit

BETWEEN

- BETWEEN keyword allows us to access the values within the specified range.
- It is a shorthand for >= AND <=.

- Syntax:

```
expression BETWEEN lower_value AND upper_value;
```

- **Example:** Given the following vendor_booth_assignment table from Farmer's market (Mandi) schema. Find the booth assignments for vendors for any market date that occurred between April 6, 2019, and April 30, 2019, including either of the two dates.

vendor_id	booth_number	market_date
1	2	2019-04-03
3	1	2019-04-03
4	7	2019-04-03
7	11	2019-04-03
8	6	2019-04-03
9	8	2019-04-03
1	2	2019-04-06

Solution query:

```
SELECT * FROM vendor_booth_assignments
WHERE market_date BETWEEN "2019-04-06" AND "2019-04-30"
```

Output:

vendor_id	booth_number	market_date
1	2	2019-04-06
3	1	2019-04-06
4	7	2019-04-06
7	11	2019-04-06
8	6	2019-04-06
9	8	2019-04-06
1	7	2019-04-10

IN operator

- IN operator is a shorthand for multiple OR conditions that allows you to specify multiple values in a WHERE clause.
- Syntax:

```
WHERE column_name IN (value1, value2, ...)
```

- **Example:** Return a list of customers with selected last names: Diaz, Edwards, and Wilson from the *customer* table.

customer_id	customer_first_name	customer_last_name	customer_zip
1	Jane	Connor	22801
2	Manuel	Diaz	22821
3	Bob	Wilson	22821
4	Deanna	Washington	22801
5	Abigail	Harris	22801
6	Betty	Bullard	22801

The following query selects all the rows where the *customer_last_name* column has the value "Diaz" or "Edwards" or "Wilson".

Solution query:

```
SELECT * from customer
WHERE customer_last_name IN ("Diaz", "Edwards", "Wilson")
```

Output:

customer_id	customer_first_name	customer_last_name	customer_zip
2	Manuel	Diaz	22821
3	Bob	Wilson	22821
10	Russell	Edwards	22801
17	Carlos	Diaz	22802

LIKE clause

- The LIKE clause is used to compare a value to similar values using wildcard operators.
- There are two kinds of wildcard operators in SQL:
  - % sign stands for any number of characters including none.
  - _ sign stands for only one character.
- For example,
  - “a%” represents a string starting with “a”.
  - “%a” represents a string that ends with “a”.
  - “a_b%” represents a string starting with a and having one character between “a” and “b”.
- **Example:** Given the *customers* table. List all the customers with a CustomerName that starts with "a" and ends with "o"

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

Query:

```
SELECT * FROM Customers
WHERE ContactName LIKE 'a%o';
```

Output:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
69	Romero y tomillo	Alejandra Camino	Gran Vía, 1	Madrid	28001	Spain

IS NULL

- The IS NULL condition is used in SQL to test for a NULL value. It returns TRUE if a NULL value is found, otherwise returns FALSE.
- Syntax: expression IS NULL
- **Example:** From the given *product* table, find all the products without any size value.

product_id	product_name	product_size	product_category_id	product_qty_type
1	Habanero Peppers - Organic	medium	1	lbs
2	Jalapeno Peppers - Organic	small	1	lbs
3	Poblano Peppers - Organic	large	1	unit
4	Banana Peppers - Jar	8 oz	3	unit
5	Whole Wheat Bread	1.5 lbs	3	unit
6	Cut Zinnias Bouquet	medium	5	unit
7	Apple Pie	10	3	unit

product table

Solution query:

```
SELECT * from products
WHERE product_size IS NULL;
```

Output:

product_id	product_name	product_size	product_category_id	product_qty_type
14	Red Potatoes	NULL	1	NULL

- Similarly, The IS NOT NULL condition is used in SQL to test for a non-NULL value.

Subqueries

- A Subquery is a query within a query. It provides data to the enclosing query.
- Syntax:

```
SELECT column_name
FROM table_name
WHERE column_name expression operator
    ( SELECT COLUMN_NAME  from TABLE_NAME   WHERE ... );
```

- **Example:** List out all the purchases made at the farmer's market on the days when it rained. Use the required tables from the Farmer's Market(Mandi) database.

  Solution: Firstly, to find out the days when it rained, we use the *market_date_info* table.

```
SELECT market_date
FROM market_date_info
WHERE market_rain_flag = 1
```

  We got the following dates when it rained.

market_date
2019-03-20
2019-03-23
2019-03-30
2019-07-31
2019-09-21
2019-10-19
2019-12-04
2019-12-11

  Now, for the above list of dates, we need to get all the information in the table *customer_purchases.* Thus, we use the IN operator with the above query as a subquery.

  Query:

```
SELECT * FROM customer_purchases
WHERE market_date IN
    (
    SELECT market_date
    FROM market_date_info
    WHERE market_rain_flag = 1)
```

Output:

product_id	vendor_id	market_date	customer_id	quantity	cost_to_customer_per_qty	transaction_time
1	7	2019-07-31	3	2.64	6.99	18:36:00
1	7	2019-07-31	8	3.83	6.99	18:34:00
1	7	2019-07-31	19	3.69	6.99	18:11:00
1	7	2019-07-31	22	1.07	6.99	18:04:00
1	7	2019-09-21	6	1.99	6.99	12:17:00
1	7	2019-09-21	24	1.82	6.99	11:11:00

CASE and WHEN

- The CASE statement in SQL handles if/then logic.
- The CASE statement is followed by at least one pair of WHEN and THEN statements and finally an ELSE clause.
- The CASE expression goes through conditions and returns a value when the first condition is met. Once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.
- Syntax:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
                :
                :
    WHEN conditionN THEN resultN
    ELSE result
END AS [new_column_name];
```

- **Example:** Using the *customer_purchases* table from the Farmer's market database, bucketize the total cost to the customer into the following bins:

    i. Under $5

    ii. $5 - $9.99

    iii. $10 - $19.99

    iv. $20 and over.

Query:

```
SELECT
     market_date,
     customer_id,
     ROUND(quantity * cost_to_customer_per_qty, 2) AS
     total_amt,
     CASE
          WHEN quantity * cost_to_customer_per_qty < 5.00
          THEN " under $5 "
          WHEN quantity * cost_to_customer_per_qty < 10.00
          THEN "$5 - $9.99"
          WHEN quantity * cost_to_customer_per_qty < 20.00
          THEN "$10 - $19.99"
          WHEN quantity * cost_to_customer_per_qty >= 20.00
          THEN "$20 and up"
     END AS price_bin
FROM customer_purchases
```

Output:

market_date	customer_id	total_amt	price_bin
2019-07-03	14	6.92	$5 - $9.99
2019-07-03	14	15.24	$10 - $19.99
2019-07-03	15	10.69	$10 - $19.99
2019-07-03	16	14.12	$10 - $19.99
2019-07-03	22	4.61	under $5
2019-07-06	4	1.89	under $5
2019-07-06	12	25.16	$20 and up

IF:

- The IF() function returns a value if the condition is TRUE and another value if the condition is FALSE.
- The IF() function can return values that can be either numeric or strings depending upon the context in which the function is used.
- The IF() function accepts one parameter which is the condition to be evaluated.

Syntax:

```
IF(condition, true_value, false_value)
```

Parameters used:

- **condition –** It is used to specify the condition to be evaluated.
- **true_value –** It is an optional parameter that is used to specify the value to be returned if the condition evaluates to be true.

- **false_value –** It is an optional parameter that is used to specify the value to be returned if the condition evaluates to be false.

Example: Use *vendor_inventory* table from Farmer's market database,
Create a new column 'Product_quantity' based on the *quantity* column and label it as **'High'** if quantity>20 else **'Low'.**

Query:

```
SELECT *, IF(quantity > 20, 'High','Low') as 'Product_quantity'
FROM vendor_inventory;
```

Output:

market_date	quantity	vendor_id	product_id	original_price	Product_quantity
2020-09-12	10.84	7	1	6.99	Low
2020-09-16	10.11	7	1	6.99	Low
2020-09-19	10.04	7	1	6.99	Low
2020-09-23	10.19	7	1	6.99	Low
2020-09-26	9.88	7	1	6.99	Low
2020-09-30	13.76	7	1	6.99	Low
2019-07-03	33.63	7	2	3.49	High
2019-07-06	24.56	7	2	3.49	High
2019-07-10	28.83	7	2	3.49	High
2019-07-13	29.17	7	2	3.49	High
2019-07-17	29.89	7	2	3.49	High

IFNULL:

- The IFNULL function specifies a value other than a null that is returned to your application when a null is encountered.
- The IFNULL() function is specified as follows: `IFNULL(v1,v2)`
- If the value of the first argument is not null, IFNULL returns the value of the first argument. If the first argument evaluates to a null, IFNULL returns the second argument.

Example: Use *products* table from Farmer's market database, Replace the product_size with 0 if the product_size value is null.

Query:

```
SELECT IFNULL(product_size,0) FROM product;
```

Output:

Before:

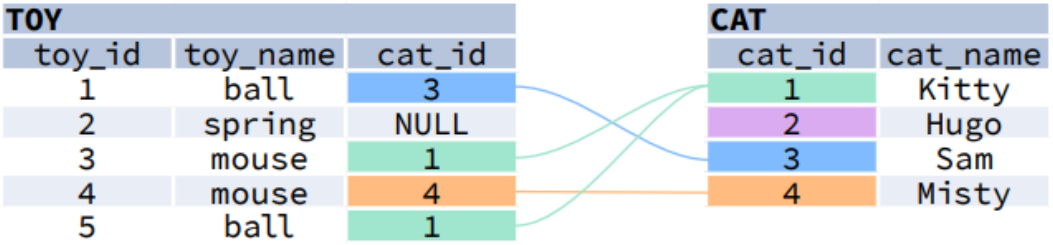
product_id	product_name	product_size	product_category...	product_qty_type
14	Red Potatoes	NULL	1	NULL

After:

product_id	product_name	product_size	product_category_id	product_qty_type
14	Red Potatoes	0	1	NULL

JOINS

- JOINS are used to combine the data from two tables. It combines rows with equal values for the specified columns.
- The JOIN condition is the equality between the primary key column in one table and columns referring to them in the other table.
- Given the following two tables.



The cat_id column is a primary key for the cat table and the cat_id column in the toy table refers to the primary key columns in the first table. The two tables can be joined using these columns.

- Syntax for JOINS :

```
SELECT
      _____

FROM [left_table]

[join_type] [right_table]

ON [left_table].[field_to_match] = [right_table].[field_to_match]
```

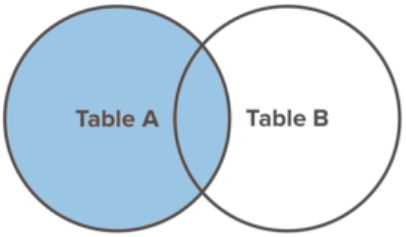
Types of JOIN

- There are the following types of JOINS:
  1. INNER JOIN
  2. OUTER JOIN
  3. LEFT JOIN
  4. RIGHT JOIN

5. SELF JOIN
6. CROSS JOIN

LEFT JOIN

- LEFT JOIN returns all rows from the left table with matching rows from the right table. Rows without a match are filled with NULLs.



SQL LEFT JOIN

- **Example:** List all the products along with their product category names. Use the Farmer's Market(Mandi) database schema and tables for this purpose.

  Solution: To get the above results, we will perform a left join considering the *product* table as the left table (since we need all the product info) using the foreign key *product_category_id.*

  Query:

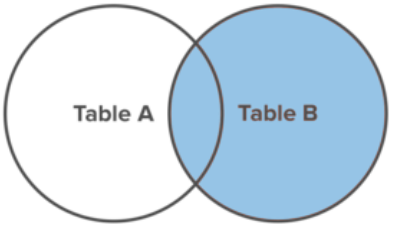
```
SELECT
     p.*,
     pc.product_category_name
FROM product as p
LEFT JOIN product_category as pc
ON p.product_category_id = pc.product_category_id;
```

  Output:

product_id	product_name	product_size	product_category_id	product_qty_type	product_category_name
1	Habanero Peppers - Organic	medium	1	lbs	Fresh Fruits & Vegetables
2	Jalapeno Peppers - Organic	small	1	lbs	Fresh Fruits & Vegetables
3	Poblano Peppers - Organic	large	1	unit	Fresh Fruits & Vegetables
4	Banana Peppers - Jar	8 oz	3	unit	Packaged Prepared Food
5	Whole Wheat Bread	1.5 lbs	3	unit	Packaged Prepared Food
6	Cut Zinnias Bouquet	medium	5	unit	Plants & Flowers
7	Apple Pie	10	3	unit	Packaged Prepared Food
8	Cherry Pie	10	3	unit	Packaged Prepared Food

RIGHT JOIN

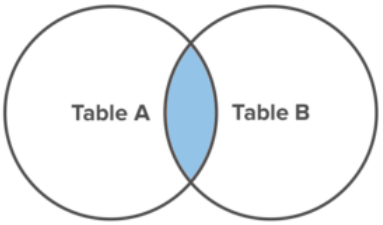
- RIGHT JOIN returns all rows from the right table with matching rows from the left table. Rows without a match are filled with NULLs.
- It is similar to the LEFT JOIN, the only difference is that we are considering all the rows on the right table instead of the left table.



SQL RIGHT JOIN

INNER JOIN

- The INNER JOIN joins two tables based on a common column and selects rows that have matching values in these columns.



SQL INNER JOIN

- **Example:** Get a list of customers' ZIP codes who made a purchase on 2019-04-06.Use the Farmer's Market(Mandi) database schema and tables.

  Query:

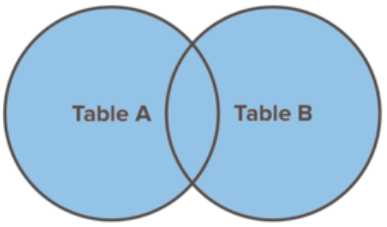
```
SELECT
     c.customer_id,
     c.customer_zip
FROM customer as c
JOIN customer_purchases as cp
ON c.customer_id = cp.customer_id
WHERE cp.market_date = "2019-04-06"
```

  Output:

customer_id	customer_zip
2	22821
5	22801
12	22821
12	22821
14	22801
14	22801
16	22801

OUTER JOIN (FULL JOIN)

- FULL JOIN returns all rows from the left table and all rows from the right table. It fills the non-matching rows with NULLs.



OUTER JOIN

- There is no direct way in MYSQL to perform the outer join. Thus, we perform it using a combination of other join types such as LEFT JOIN ad RIGHT JOIN.
- We first use LEFT JOIN and RIGHT JOIN on the tables and then use UNION to combine the results and remove the duplicate rows.
- To perform OUTER JOIN on *tableA* and *tableB*, we can use the following query:

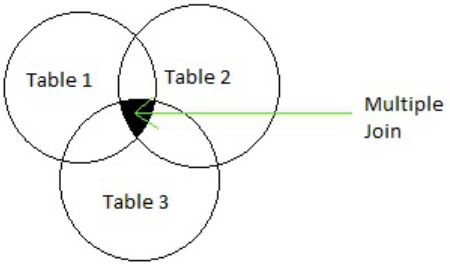
```
SELECT * FROM tableA
LEFT JOIN tableB ON tableA.id = tableB.id
UNION
SELECT * FROM tableA
RIGHT JOIN tableB ON tableA.id = tableB.id
```

UNION

- UNION combines the results of two result sets and removes duplicates.
- UNION ALL doesn't remove duplicate rows.

MULTIPLE JOINS

- Multiple joins can be described as a query containing joins of the same or different types used multiple times, thus giving them the ability to combine more than two tables.



- **Example:** Let us assume that there are three tables in our database: *Student*, *Branch*, and *Address*.

Stud_id	Name	Br_id	Email	City_id
1001	Ankit	101	ankit@bmail.com	1
1002	Pranav	105	pranav@bmail.com	2
1003	Raj	102	raj@bmail.com	2
1004	Shyam	112	shyam@bmail.com	4
1005	Duke	112	duke@bmail.com	2
1006	Jhon	102	jhon@bmail.com	3
1007	Aman	101	aman@bmail.com	4
1008	Pavan	111	pavan@bmail.com	13

Student table

Br_id	Br_name	HOD	Contact
101	CSE	SH Rao	22345
102	MECH	AP Sharma	28210
103	EXTC	VK Reddy	34152
104	CHEM	SK Mehta	45612
105	IT	VL Shelke	22521
106	AI	KH Verma	12332
107	PROD	PG Kakde	90876

Branch table

City_id	City	Pincode
1	Mumbai	400121
2	Pune	450011
3	Lucknow	553001
4	Delhi	443221
5	Kolkata	213445
6	Chennai	345432
7	Nagpur	323451
8	Sri Nagar	321321

Address table

Obtain students' names along with their branch names, HOD, city, and Pincode.
Solution: if we see the *Student* and *Branch* table have Br_id common and the *Address* and *Student* table have city_id common.
So, to retrieve data, first, we need to join two tables and then the third table.

Query:

```
SELECT Student.Name, Branch.Br_id, Branch.HOD,
Address.city, Address.pincode from Branch
INNER JOIN Student
```

```
on Student.Br_id = Branch.Br_id
INNER JOIN Address
on Student.city_id = Address.city_id;
```

Output:

Name	Br_id	HOD	City	Pincode
Ankit	101	SH Rao	Mumbai	400121
Pranav	105	VL Shelke	Pune	450011
Raj	102	AP Sharma	Pune	450011
Jhon	102	AP Sharma	Lucknow	553001
Aman	101	SH Rao	Delhi	443221

SELF JOIN

- In this type of join, we join a table to itself.
- **Example:** Given the *employee* table below. For each employee, find out the name of their manager.

emp_id	emp_name	job_name	manager_id	hire_date	salary	commission	dep_id
68319	KAYLING	PRESIDENT		1991-11-18	6000.00		1001
66928	BLAZE	MANAGER	68319	1991-05-01	2750.00		3001
67832	CLARE	MANAGER	68319	1991-06-09	2550.00		1001
65646	JONAS	MANAGER	68319	1991-04-02	2957.00		2001
64989	ADELYN	SALESMAN	66928	1991-02-20	1700.00	400.00	3001
65271	WADE	SALESMAN	66928	1991-02-22	1350.00	600.00	3001

To solve the above example, we need to join the table with itself considering emp_id as the primary key and manager_id as the foreign key.

Query:

```
SELECT
     emp.emp_name AS employee,
     mgr.emp_name AS manager
FROM employees AS emp
JOIN employees AS mgr
ON mgr.emp_id = emp.manager_id
```

Output:

employee	manager
BLAZE	KAYLING
CLARE	KAYLING
JONAS	KAYLING
ADELYN	BLAZE
WADE	BLAZE
MADDEN	BLAZE
TUCKER	BLAZE
ADNRES	SCARLET
JULIUS	BLAZE

CROSS JOIN

- CROSS JOIN returns all possible combinations of rows from the left and right tables.
- For example,

TOY

toy_id	toy_name	cat_id
1	ball	3
2	spring	NULL
3	mouse	1
4	mouse	4
5	ball	1

CAT

cat_id	cat_name
1	Kitty
2	Hugo
3	Sam
4	Misty

Query:

```
SELECT *
FROM toy
CROSS JOIN cat;
```

Another syntax

```
SELECT *
FROM toy, cat;
```

Output:

toy_id	toy_name	cat_id	cat_id	cat_name
1	ball	3	1	Kitty
2	spring	NULL	1	Kitty
3	mouse	1	1	Kitty
4	mouse	4	1	Kitty
5	ball	1	1	Kitty
1	ball	3	2	Hugo
2	spring	NULL	2	Hugo
3	mouse	1	2	Hugo
4	mouse	4	2	Hugo
5	ball	1	2	Hugo
1	ball	3	3	Sam
...	...	...	...	...

Correlated subquery

- A correlated subquery is a subquery that uses values from the outer query.
- With a normal nested subquery, the inner SELECT query runs first and executes once whereas the correlated subquery references a column in the outer query and executes the subquery once for each row in the outer query.
- **Example:** Given the *employee* table. Find all the employees who have a salary greater than their department's average salary.

emp_id	emp_name	job_name	manager_id	hire_date	salary	commission	dep_id
68319	KAYLING	PRESIDENT		1991-11-18	6000.00		1001
66928	BLAZE	MANAGER	68319	1991-05-01	2750.00		3001
67832	CLARE	MANAGER	68319	1991-06-09	2550.00		1001
65646	JONAS	MANAGER	68319	1991-04-02	2957.00		2001
64989	ADELYN	SALESMAN	66928	1991-02-20	1700.00	400.00	3001
65271	WADE	SALESMAN	66928	1991-02-22	1350.00	600.00	3001

Query:

```
SELECT e1.emp_id
FROM employees AS e1
WHERE e1.salary > (
     SELECT AVG(e2.salary)
     FROM employees AS e2
     where e2.dep_id = e1.dep_id  )
```

Output:

emp_id
68319
66928
65646
64989
67858
69062

GROUP BY

- GROUP BY groups together rows that have the same values in specified columns. It computes summaries (aggregated measures) for each group using the aggregate functions.
- Syntax:

```
SELECT [column_1], [column_2],
     COUNT() AS [column_name]
     SUM() AS [column_name]
FROM [table_name]
WHERE [condition]
GROUP BY [column_1], [column_2];
```

- **Example 1:** Get the number of purchases each customer has made on each market date. Use the *customer_purchases* table from the Farmer's Market(Mandi) database schema.

Solution query:

```
SELECT
     market_date,
     customer_id,
     count(*) AS num_purchases
FROM customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date
```

Output:

market_date	customer_id	num_purchases
2019-04-03	3	1
2019-04-03	4	1
2019-04-03	5	1
2019-04-03	6	1
2019-04-03	7	1
2019-04-03	12	1
2019-04-03	16	1
2019-04-06	2	1
2019-04-06	5	1
2019-04-06	12	2

GROUP BY vs. DISTINCT

- DISTINCT is used to filter unique records out of all records in the table. It removes the duplicate rows.

- The GROUP BY clause is used to group a selected set of rows into summary rows by one or more columns or an expression.
- The group by gives the same result as distinct when no aggregate function is present.
- DISTINCT collects all of the rows, including any expressions that need to be evaluated, and then tosses out duplicates. GROUP BY can filter out the duplicate rows before performing any of that work.
- GROUP BY approach definitely leads to better performance than DISTINCT.

AGGREGATE FUNCTIONS

- AVG(expr) − average value for rows within the group
- COUNT(expr) − count of values for rows within the group
- MAX(expr) − maximum value within the group
- MIN(expr) − minimum value within the group
- SUM(expr) − the sum of values within the group

COUNT(*) vs **COUNT(column_name)** vs **COUNT(DISTINCT column_name)**

- COUNT(*) returns the total number of records in the table. COUNT(1), COUNT(999), COUNT("xyz") are all same as COUNT(*).
- COUNT(column_name) returns the total number of records of the table including the duplicate entries in the column *column_name* but ignores the NULL values.
- COUNT(DISTINCT column_name) neither includes the duplicate entries nor the NULL values.

- **Example:** Applying each function on the *product* table of Farmer's market database schema to count the number of product sizes.

  Query:

```
SELECT
     COUNT(*),
     COUNT(product_size),
     COUNT(DISTINCT product_size)
 FROM product
```

  Output:

count(*)	count(product_size)	count(distinct product_size)
23	22	15

HAVING clause

- WHERE keyword fails when we use it with aggregate expressions like COUNT(), MAX(), AVG(), etc. along with the grouping.
- The HAVING clause is used to filter the results obtained by the GROUP BY clause based on some specific conditions.
- In a query, the HAVING clause is placed after the GROUP BY clause and before the ORDER BY clause.

- Syntax:

```
SELECT column1, column2,....,columnN
FROM tableName
WHERE [conditions]
GROUP BY column1
HAVING [conditons]
ORDER BY column_name
```

- **Example:** Given the *customer_purchases* table. Find the average amount spent on each market day where:
    1. The number of purchases was more than 3.
    2. The transaction amount was greater than 5.

Query:

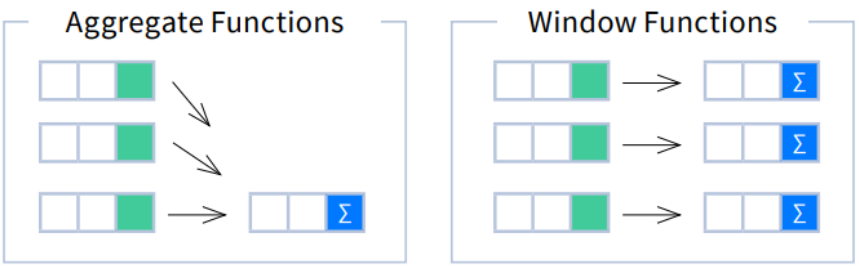
```
SELECT
    market_date,
    AVG(quantity * cost_to_customer_per_qty) AS
avg_price,
    COUNT(*) as no_of_purchases
FROM customer_purchases
WHERE quantity * cost_to_customer_per_qty > 5
GROUP BY market_date
HAVING COUNT(*)>3
ORDER BY market_date
```

Output:

market_date	avg_price	no_of_purchases
2019-04-03	13.6	5
2019-04-06	15.428571428571429	7
2019-04-10	15.333333333333334	6
2019-04-13	13.714285714285714	7
2019-04-17	13.222222222222221	9
2019-04-20	13.6	5
2019-04-24	14.285714285714286	7
2019-04-27	12.25	8

Window functions

- Window functions are used to perform some operation on a group of rows and provide a resultant value for each row in the table. They compute their result based on a sliding window frame, a set of rows that are somehow related to the current row.
- OVER clause is used with window functions to define the window.
- GROUP BY collapses the individual records into groups. i.e. after using GROUP BY, you cannot refer to any individual field because it is collapsed.
- However, Window functions do not collapse individual records and the **row-level information** is intact in the partitions. Thus, we can create queries showing data from the individual record together with the result of the window function.



- **Syntax**

```
SELECT
     <column_1>,
      <column_2>,
      <window_function>() OVER (
      PARTITION BY < column_to_partition_by >
      ORDER BY < column_to_order_by >
      <window_frame>) AS <window_column_alias>
FROM <table_name>;
```

PARTITION BY

- It divides the rows into multiple groups, called partitions, to which the window function is applied.
- With no PARTITION BY clause, the entire result set is the partition.

- **Example:** Count how many products each vendor bought to the market on each date and display the count on each row. Use the *vendor_inventory* table of the Farmer's Market(Mandi) schema.

  Solution:

We will partition the table by two columns (*market_date* and *vendor_id*), then use the window function COUNT() to count the number of products in each partition.

Query:

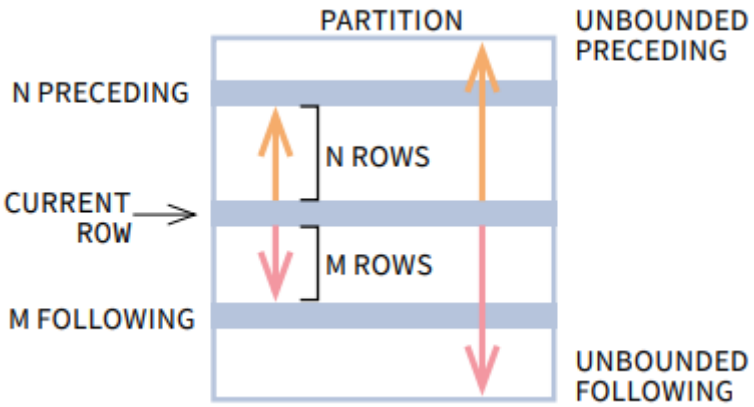
```
SELECT
    vendor_id,
    market_date,
    original_price,
    COUNT(product_id) OVER (PARTITION BY market_date,
vendor_id) AS count_product
FROM vendor_inventory
```

Output:

vendor_id	market_date	original_price	count_v_product
7	2019-04-03	4	1
8	2019-04-03	6.5	3
8	2019-04-03	18	3
8	2019-04-03	18	3
7	2019-04-06	4	1
8	2019-04-06	6.5	3
8	2019-04-06	18	3
8	2019-04-06	18	3

Window frame

- A window frame is a set of rows that are somehow related to the current row.
- The window frame is evaluated individually within each partition.



- Syntax for window frame is :

```
ROWS | RANGE BETWEEN lower_bound AND upper_bound
```

- The bounds (lower_bound, upper_bound) can be any of the following five options:
    - UNBOUNDED PRECEDING
    - n PRECEDING
    - CURRENT ROW
    - n FOLLOWING
    - UNBOUNDED FOLLOWING

Abbreviation	Meaning
UNBOUNDED PRECEDING	BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
n PRECEDING	BETWEEN n PRECEDING AND CURRENT ROW
CURRENT ROW	BETWEEN CURRENT ROW AND CURRENT ROW
n FOLLOWING	BETWEEN AND CURRENT ROW AND n FOLLOWING
UNBOUNDED FOLLOWING	BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

- **Example:** Calculate the 3-day moving average for the given *stocks* table for each row and display in front of each row.

date	price
2020-01-07	1320
2020-01-08	1300
2020-01-09	1300
2020-01-10	1300
...	...
2020-06-24	1086
2020-06-25	1095
2020-06-26	1067
2020-06-27	1067
2020-06-28	1076

Solution:

We use a window function where each row has its own window over which the average operation is performed.

The moving average of the current date of a current row is calculated by the average of 2 preceding rows and the current one using the AVG() function.

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW. This statement in the below query says that, for each row in the table, something is calculated as an aggregation of the current and the previous two rows.

Query:

```
SELECT
     date,
     price,
```

```
        AVG(Price) OVER(ORDER BY date ROWS BETWEEN 2 PRECEDING
        AND CURRENT ROW ) AS moving_average
from stocks;
```

Output:

date	price	moving_average
2020-01-07	1320	1320
2020-01-08	1300	1310
2020-01-09	1300	1306.6667
2020-01-10	1300	1300
2020-01-13	1300	1300
...	...	...
2020-06-25	1095	1089
2020-06-26	1067	1082.6667
2020-06-27	1067	1076.3333
2020-06-28	1076	1070
2020-06-29	1067	1070
2020-06-30	1067	1070

Types of window functions:

- The types of functions on which we can apply a window in SQL are:

1. **Aggregate Window Functions**
    - SUM()
    - MIN()
    - MAX()
    - COUNT()
    - AVG()

2. **Ranking Window Functions**

    - ROW_NUMBER() - This function assigns a unique number to each row in the table.

    - RANK() - This function is used to assign a value to the records in the table, and it can be the same for two or more records having the same values (lies within the same partition). It also skips ranks if the records are the same.

    - DENSE_RANK() - This function assigns a unique rank to the records in the table. It is similar to the RANK() function but it doesn't skip ranks if two or more records have the same ranks.

      Note: Ranking functions do not accept window frame definitions (ROWS, RANGE, GROUPS).

Example: In the given *employee* table, calculate the row no., rank, and dense rank of each employee according to salary within each department.

Name	Department	Salary
Suresh	Finance	50, 000
Ramesh	Finance	50, 000
Ram	Finance	20, 000
Deep	Sales	30, 000
Pradeep	Sales	20, 000

employee table

Query:

```
SELECT
      ROW_NUMBER() OVER (PARTITION BY Department ORDER BY Salary
      DESC) AS emp_row_no,
      Name, Department, Salary,
      RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS
      emp_rank,
      DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary
      DESC) AS emp_dense
FROM employee
```

Output:

emp_row_no	Name	Department	Salary	emp_rank	emp_dense_
1	Suresh	Finance	50, 000	1	1
2	Ramesh	Finance	50, 000	1	1
3	Ram	Finance	20, 000	3	2
1	Deep	Sales	30, 000	1	1
2	Pradeep	Sales	20, 000	2	2

- We can see that in dense rank, there is no gap between rank values while there is a gap in rank values after repeated rank.
- The **row_number()** function always generates a unique ranking even with duplicate records i.e. if the ORDER BY clause cannot distinguish between two rows, it will still

give them different rankings, though which record will come earlier or later is decided randomly.

- The **rank()** and **dense_rank()** will give the same ranking to rows that cannot be distinguished by the order by clause, but dense_rank will always generate a contiguous sequence of ranks like (1,2,3,...), whereas rank() will leave gaps after two or more rows with the same rank.
- The **row_number()** function can be used for:
    - Assigning sequential numbers to rows
    - Finding the top N rows of every group
    - Removing duplicate rows
    - Pagination (i.e., Assigns each row in the result set a unique number)

3. Value Window Functions

- LEAD() - This function allows us to retrieve data from the next row of the current row in the same result set.
  Syntax: `LEAD(expression, offset, default_value) OVER(PARTITION BY columns ORDER BY columns)`

- LAG() - This function allows us to retrieve data from the preceding row of the current row in the same result set.
  Syntax: `LAG(expression, offset, default_value) OVER(PARTITION BY columns ORDER BY columns)`

Note: Both LEAD() and LAG() functions have similar syntax and take three arguments:

- the name of a column or an expression,
- the offset to be skipped below or above, and
- the default value to be returned if the stored value obtained from the row below is empty. It is specified only if you specify the second argument, the offset.

- NTILE() - This function divides rows within a partition as equally as possible into n groups, and assigns each row its group number.

Example: Display each vendor's booth assignment for each market_date alongside their previous and next booth assignments. Use the *vendor_booth_assignment* table from Farmer's market (Mandi) schema.

Query:

```
SELECT
     market_date,
     vendor_id,
     booth_number,
     LAG(booth_number,1, NULL) OVER (PARTITION BY vendor_id
     ORDER BY  market_date) AS previous_booth,
```

```
        LEAD(booth_number,1, NULL) OVER (PARTITION BY vendor_id
        ORDER BY market_date) AS next_booth

    FROM vendor_booth_assignments
```

Output:

market_date	vendor_id	booth_number	previous_booth	next_booth
2019-04-03	1	2	NULL	2
2019-04-06	1	2	2	7
2019-04-10	1	7	2	2
2019-04-13	1	2	7	2
2019-04-17	1	2	2	2
2019-04-20	1	2	2	2
2019-04-24	1	2	2	2
2019-04-27	1	2	2	2
2019-05-01	1	2	2	2

- We can't use window functions in the WHERE clause, because the logical order of execution of an SQL query is completely different from the SQL syntax.

Execution order in SQL

1. **FROM -** the database gets the data from tables in the FROM clause and if necessary performs the JOINs,
2. **WHERE -** the data are filtered with conditions specified in the WHERE clause,
3. **GROUP BY -** the data are grouped by conditions specified in WHERE clause,
4. **Aggregate functions -** the aggregate functions are applied to the groups created in the GROUP BY phase,
5. **HAVING -** The groups are filtered with the given condition,
6. **Window functions**,
7. **SELECT -** the database selects the given columns,
8. **DISTINCT** - repeated values are removed,
9. **UNION/INTERSECT/EXCEPT** - the database applies set operations,
10. **ORDER BY** - the results are sorted,
11. **OFFSET** - the first rows are skipped,
12. **LIMIT/FETCH/TOP** - only the first rows are selected

- We can clearly see that the window functions execute after the WHERE clause. They would not have been evaluated yet at the time the WHERE filters are processed. Thus, we can't include window functions in a WHERE clause. For this purpose, we use the **subqueries**.

Some more window functions:

- **NTH_VALUE():** This function returns the value of a given expression from the N^{th} row of the window frame. If that Nth row does not exist, the function returns NULL.

The syntax of the NTH_VALUE() function is given as

```
NTH_VALUE(expression, N)
```

- **FIRST_VALUE():** This window function returns the first value in an ordered partition of a result set.

- **LAST_VALUE():** This window function returns the first value in an ordered partition of a result set.

Example: Given the *employees* table. Get the second highest salary in each department in each row.

emp_id	emp_name	job_name	manager_id	hire_date	salary	commission	dep_id
68319	KAYLING	PRESIDENT		1991-11-18	6000.00		1001
66928	BLAZE	MANAGER	68319	1991-05-01	2750.00		3001
67832	CLARE	MANAGER	68319	1991-06-09	2550.00		1001
65646	JONAS	MANAGER	68319	1991-04-02	2957.00		2001
64989	ADELYN	SALESMAN	66928	1991-02-20	1700.00	400.00	3001

employees table

Query:

```
SELECT *,
 NTH_VALUE(salary,2) OVER (PARTITION BY dep_id ORDER BY salary desc
 RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
 AS second_highest
FROM employees;
```

Output:

emp_id	emp_name	job_name	manager_id	hire_date	salary	commission	dep_id	second_highest
68319	KAYLING	PRESIDENT		1991-11-18	6000.00		1001	2550.00
67832	CLARE	MANAGER	68319	1991-06-09	2550.00		1001	2550.00
69324	MARKER	CLERK	67832	1992-01-23	1400.00		1001	2550.00
67858	SCARLET	ANALYST	65646	1997-04-19	3100.00		2001	3100.00
69062	FRANK	ANALYST	65646	1991-12-03	3100.00		2001	3100.00
65646	JONAS	MANAGER	68319	1991-04-02	2957.00		2001	3100.00
68736	ADNRES	CLERK	67858	1997-05-23	1200.00		2001	3100.00

DATE and Time functions

1. **STR_TO_DATE()**
    - This function converts the string value to date or time or DateTime values.

- **Syntax**: `STR_TO_DATE(string, format)`
- The string of percent signs and letters in single quotes at the end is an input parameter that tells the function how the date and time are formatted.
- **%Y is a 4-digit year,**
- **%m is a 2-digit month,**
- **%d is a 2-digit day,**
- **%h is the hour,**
- **%i represents the minutes, and**
- **%p indicates there is an AM/PM**

2. **EXTRACT()**
   - The EXTRACT() function is used to extract a part from a given date.
   - **Syntax**: `EXTRACT(part FROM date)`

Example: Given the date_time_demo table. From each market_start_datetime, extract the following:

- day of the week,
- month of year,
- year,
- hour and
- minute from the timestamp

market_date	market_start_time	market_end_time	market_start_datetime	market_end_datetime
2019-03-02	8:00 AM	2:00 PM	2019-03-02 08:00:00	2019-03-02 14:00:00
2019-03-09	9:00 AM	2:00 PM	2019-03-09 09:00:00	2019-03-09 14:00:00
2019-03-13	4:00 PM	7:00 PM	2019-03-13 16:00:00	2019-03-13 19:00:00
2019-03-16	8:00 AM	2:00 PM	2019-03-16 08:00:00	2019-03-16 14:00:00
2019-03-20	4:00 PM	7:00 PM	2019-03-20 16:00:00	2019-03-20 19:00:00
2019-03-23	8:00 AM	2:00 PM	2019-03-23 08:00:00	2019-03-23 14:00:00
2019-03-27	4:00 PM	7:00 PM	2019-03-27 16:00:00	2019-03-27 19:00:00
2019-03-30	8:00 AM	2:00 PM	2019-03-30 08:00:00	2019-03-30 14:00:00
2019-04-03	4:00 PM	7:00 PM	2019-04-03 16:00:00	2019-04-03 19:00:00
2019-04-06	8:00 AM	2:00 PM	2019-04-06 08:00:00	2019-04-06 14:00:00

date_time_demo table

Query:

```
SELECT
     market_start_datetime,
     EXTRACT(DAY FROM
        market_start_datetime) AS start_day,
     EXTRACT(YEAR FROM
        market_start_datetime) AS date_year,
     EXTRACT(MONTH FROM
        market_start_datetime) AS month_of_year,
     EXTRACT(HOUR FROM
        market_start_datetime) AS hour_of_day,
     EXTRACT(MINUTE FROM
        market_start_datetime) AS minute_of_time
FROM date_time_demo;
```

Output:

market_start_datetime	mkt_day	mkt_month	mkt_year	mkt_hour	mkt_minute
2019-03-09 09:00:00	9	3	2019	9	0
2019-03-13 16:00:00	13	3	2019	16	0
2019-03-16 08:00:00	16	3	2019	8	0
2019-03-20 16:00:00	20	3	2019	16	0
2019-03-23 08:00:00	23	3	2019	8	0
2019-03-27 16:00:00	27	3	2019	16	0
2019-03-30 08:00:00	30	3	2019	8	0
2019-04-03 16:00:00	3	4	2019	16	0
2019-04-06 08:00:00	6	4	2019	8	0
2019-04-10 16:00:00	10	4	2019	16	0

3. **DATE_ADD()/DATE_SUB()**

   - The DATE_ADD/DATE_SUB() function add/subtracts a time/date interval from a date and then returns the date respectively.
   - **Syntax**: `DATE_SUB(date, INTERVAL value interval)`
     `DATE_ADD(date, INTERVAL value interval)`

   - **Example**: `SELECT DATE_SUB("2019-06-14", INTERVAL 10 DAY);`
     - The above query will subtract 10 days from the date 2019-06-14 and return the date. i.e. 2019-06-04.

4. **DATEDIFF()**
   - The DATEDIFF() function returns the number of days between two date values.
   - **Syntax:** `DATEDIFF(date1, date2)`

   - **Example**: `SELECT DATEDIFF("2019-01-01", "2016-12-24");`
     - Return the number of days between two dates 2019-01-01 and 2016-12-24. i.e. 738

5. **CURDATE()**
   - The CURDATE() function returns the current date.
   - The date is returned as "YYYY-MM-DD" (string) or as YYYYMMDD (numeric).

Example 1: Find out how long a customer has been part of the market. Use the *customer_purchases* table from Farmer's Market(Mandi) database schema.

Solution:

To solve the above example, we need to find the **first and last** purchase of each customer. This can be done by grouping the data using the *customer_id* column and then applying MIN() and MAX() functions to the grouped data.
Finally, we can use the DATEDIFF() function to get the difference between the first and last purchase.

Query:

```
SELECT customer_id,
    MIN(market_date) AS first_purchase,
    MAX(market_date) AS last_purchase,
    DATEDIFF(MAX(market_date), MIN(market_date)) AS
days_between_first_last
 FROM customer_purchases
 GROUP BY customer_id
```

Output:

customer_id	first_purchase	last_purchase	days_between_first_last
14	2019-04-06	2020-09-30	543
15	2019-05-22	2020-09-30	497
16	2019-04-03	2020-09-23	539
22	2019-07-03	2020-09-19	444
4	2019-04-03	2020-09-26	542
12	2019-04-03	2020-09-09	525
23	2019-07-06	2020-09-26	448

Example 2: Today's date is May 31, 2019, and the marketing director of the farmer's market wants to give infrequent customers an incentive to return to the market in June. Write a query to pull up the list of customers who purchased only once in the previous month.
Use the *customer_purchases* table from Farmer's Market(Mandi) database schema.

Solution: Here, we need to first filter out the rows where the market dates are from the previous month.
For this, we use the DATEDIFF() function to get the difference between the current date and all the dates. If this difference comes out to be between 0 and 31, then we say that the date is from the previous month.
After filtering the data for all the dates from the previous month, grouping is done by the *customer_id* column and then the number of distinct dates is counted using COUNT() function.

Query:

```
SELECT
    customer_id,
    COUNT(DISTINCT market_date) AS market_count
FROM customer_purchases
WHERE DATEDIFF('2019-05-31',market_date) BETWEEN 0 AND 31
GROUP BY customer_id
HAVING COUNT(DISTINCT market_date) = 1;
```

Output:

customer_id	market_count
5	1
14	1
16	1
18	1

Ad-hoc reporting

In the data analysis world, being asked questions, exploring a database, writing SQL statements to find and pull the data needed to determine the answers, and conducting the analysis of that data to calculate the answers to the questions, is called **ad-hoc reporting.**

- In advanced SQL, we'll take those skills to the next level and demonstrate how to think through multiple analysis questions,
- simulate what it might be like to write queries to answer a question posed by a business stakeholder.
- We'll design and develop analytical datasets that can be used repeatedly to facilitate ad-hoc reporting.

There are multiple ways to store queries (and the results of queries) for reuse in reports and other analyses.

Here, we will cover two approaches for more easily querying from the results of custom dataset queries you build:

1. **Common Table Expressions**
2. **views.**

Common Table Expressions

- Common Table Expressions (CTEs) allow us to create temporary named results sets that exist temporarily within the execution scope of SQL statements such as SELECT, INSERT, UPDATE, DELETE, and MERGE.
- The WITH clause in SQL provides a better way to write the auxiliary/helper statement which can be later used in larger queries.
- These statements are referred to as common table expressions which are nothing but defining a temporary relational table to be used later in a SQL statement.
- The table is being called temporary because it exists only during the scope of the SQL statement written after CTEs.

The syntax for CTEs is:

```
WITH [query_alias] AS (
[query]
),
[query_2_alias] AS (
[query_2]
)
SELECT [column list]
FROM [query_alias]
```

... [remainder of query that references aliases created above]

-> where "[**query_alias**]" is a placeholder for the name you want to use to refer to a query later, and "[**query**]" is a placeholder for the query you want to reuse.

Advantages of CTE:

- Making recursive queries.
- Hold a query output virtually in a temporary area named as given while definition.
- No need to save Metadata.
- Useful when there is a need to do more operations on some query output.

Views

- Another approach to CTEs is Views.
- This involves storing the query as a database view.
- A view is treated just like a table in SQL, the only difference being that it has run when it's referenced to dynamically generate a result set (where a table stores the data instead of storing the query),
- So queries that reference views can take longer to run than queries that reference tables.
- However, the view is retrieving the latest data from the underlying tables each time it is run, so you are working with the freshest data available when you query from a view.

How it works!

Syntax:

- "`CREATE VIEW [db_name.]view_name [(column_list)]`
  `AS`
  `select-statement;` "
- If you want to store your dataset as a view, you simply precede your SELECT statement by replacing the bracketed statements with the actual schema name, and the name you are giving the view.

SQL CTE vs. View: When to Use Each One

Although some differences exist, common table expressions and views seem to perform very similarly.

So, when should you use each one?

- **Ad-hoc queries.** For queries referenced occasionally (or just once), it's usually better to use a CTE. If you need the query again, you can just copy the CTE and modify it if necessary.

- **Frequently used queries.** Creating a corresponding view is a good idea if you often reference the same query. However, you'll need to **create view permission** in your database to create a view.

- **Access management.** A view might restrict particular users' database access while allowing them to get the necessary information. You can give users access to specific views that query the data they're allowed to see without exposing the whole database. In such a case, a view provides an additional access layer.