SQL

SQL (Structured Query Language) is a programming language used for managing and manipulating data in relational databases. It allows you to insert, update, retrieve, and delete data in a database. It is widely used for data management in many applications, websites, and businesses. In simple terms, SQL is used to communicate with and control databases.

MY SQL

- MySQL is a relational database management system
- MySQL is open-source
- MySQL is free
- MySQL is ideal for both small and large applications
- MySQL is very fast, reliable, scalable, and easy to use
- MySQL is cross-platform
- MySQL is compliant with the ANSI SQL standard
- MySQL was first released in 1995
- MySQL is developed, distributed, and supported by Oracle Corporation

Difference between Relational and NOSQL databases

- Relational databases use the relational model, which organizes data into tables with rows and columns, and uses structured query language (SQL) to access and manipulate the data. They are well suited for structured data, such as financial transactions, and are commonly used in business applications.
- NoSQL databases, on the other hand, are designed to handle large amounts of unstructured or semi-structured data, such as social media posts, log files, or user-generated content. They use a variety of data storage models, including key-value, document-based, column-based, and graph databases. NoSQL databases are designed to be horizontally scalable, allowing them to handle large amounts of data and high levels of traffic.

In summary, relational databases are well suited for structured data, while NoSQL databases are designed to handle unstructured data and scale horizontally.

Difference between SQL and MYSQL

SQL (Structured Query Language) is a standard language for managing and manipulating relational databases. It's used to insert, update, and retrieve data in a database.

MySQL is an open-source relational database management system that uses SQL as its primary language. In other words, MySQL is a database management system that implements the SQL language. It's one of the most popular database systems in use today and is widely used for web applications and data storage.

To put it simply, SQL is a language, while MySQL is a database management system that uses the SQL language.

Database

What is a Database?

A database is an organized collection of data stored in the form of tables and accessed electronically. It's designed to help users manage, manipulate, and retrieve data efficiently. Databases are used in a wide range of applications, including e-commerce, financial systems, and customer relationship management.

Types of Database?

There are several types of databases, including:

- **Relational databases: Store data in tables with rows and columns and use structured query language (SQL) to access data.**
  - MYSQL
  - SQL Server
  - PostreSQL
  - SQLite
  - MariaDB
- **Non-relational databases (NOSQL): Store data in a format other than tables, such as key-value pairs, document-based, or graph databases.**
  - Hbase
  - mongodb
  - cassandra
- Centralized databases: Store data in a single, centralized location and allow multiple users to access the data from different locations.
- Distributed databases: Store data on multiple servers and allow multiple users to access the data from different locations.
- Operational databases: Store real-time data and are designed to support the day-to-day operations of an organization.
- Data warehouses: Store historical data for analysis and decision-making purposes.
- In-memory databases: Store data in RAM for faster access and processing.
- Cloud databases: Store data on remote servers and allow access over the internet.

These are the main types of databases, and different applications may use different types depending on their specific requirements.

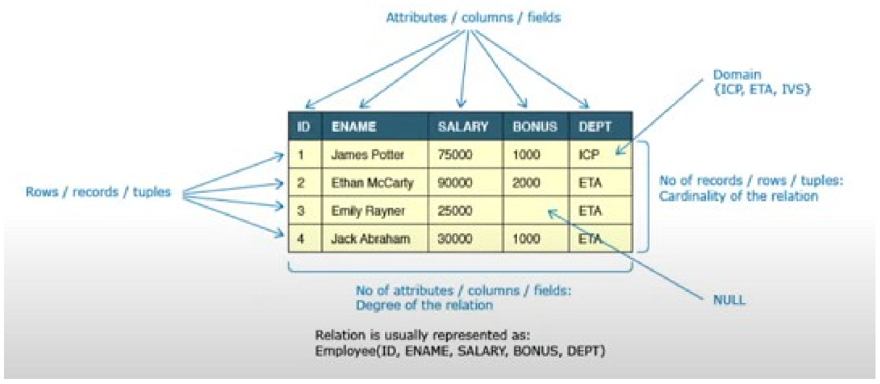
Datatypes in SQL

SQL has several standard data types, including:

- **INT :** A numeric data type that can store whole numbers. Used to store IDs, counts, or other numeric values that do not have a decimal component.
- **DECIMAL :** A numeric data type that can store numbers with a fixed number of digits to the right of the decimal point. Used to store monetary values, measurements, or other values with a known number of decimal places.
- **FLOAT :** A numeric data type that can store floating-point numbers with a decimal component. Used to store values with a large number of decimal places, such as scientific or mathematical calculations.
  - Other types : **BIGINT, SMALLINT, TINYINT, DECIMAL, NUMERIC, FLOAT, REAL**
- **VARCHAR :** A character and string data type that can store variable-length strings of characters. Used to store text values, such as names, addresses, or descriptions.
  - Other types : **CHAR, VARCHAR, TEXT**
- **DATE :** A date and time data type that stores the date (year, month, and day) without the time. Used to store dates such as birthdays, hire dates, or transaction dates.
  - Other types : **DATE, TIME, DATETIME, TIMESTAMP**
- **BOOLEAN :** A binary data type that can store either true or false values. Used to store binary data, such as yes/no or on/off values.

These are some of the most commonly used data types in SQL, and the specific data type used will depend on the type of data being stored and the requirements of the application.

Relational databases :


Attributes / columns / fields
Domain {ICP, ETA, IVS}
Rows / records / tuples
No of records / rows / tuples: Cardinality of the relation
NULL
No of attributes / columns / fields: Degree of the relation

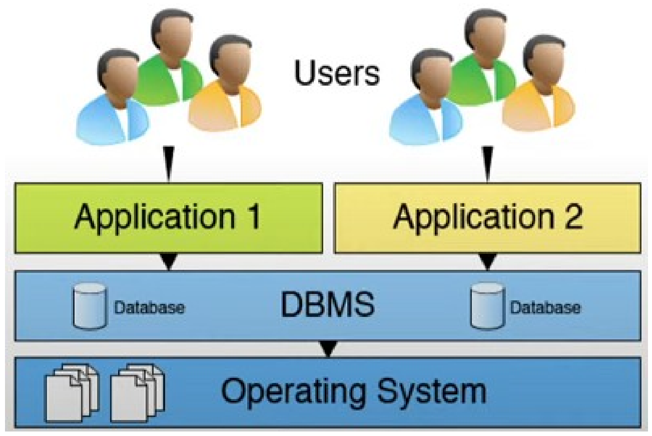
ID	ENAME	SALARY	BONUS	DEPT
1	James Potter	75000	1000	ICP
2	Ethan McCarty	90000	2000	ETA
3	Emily Rayner	25000		ETA
4	Jack Abraham	30000	1000	ETA

Relation is usually represented as:
Employee(ID, ENAME, SALARY, BONUS, DEPT)

- columns = attributes
- rows = tuples
- number of rows = cardinality
- number of columns = degree of relation
- type of column = domain

DBMS

- DBMS stands for "Database Management System." It is software that enables users and applications to interact with a database, providing functionalities for data storage, retrieval, modification, and management.
- DBMS acts as an intermediary between users and the database, ensuring data integrity, security, and efficient access to the stored information. It allows users to define, create, and manipulate databases, making it easier to organize, query, and update data in a structured manner.
- Examples of popular DBMSs include MySQL, Oracle, Microsoft SQL Server, and PostgreSQL.



Database keys:

Keys are attributes or sets of attributes that play a fundamental role in ensuring data integrity, data uniqueness, and establishing relationships between tables. Keys are used to identify and access records in a database table efficiently.

1. **Primary Key:** A unique identifier for each record in a table, ensuring data integrity and fast data retrieval.
2. **Foreign Key:** A column that establishes a link between two tables, enforcing referential integrity and maintaining data consistency across related tables.

3. **Candidate Key:** A potential primary key candidate that uniquely identifies records in a table but is not currently designated as the primary key.
4. **Unique Key:** Ensures that each value in the specified column is unique, but unlike the primary key, a table can have multiple unique keys.
5. **Composite Key:** A combination of two or more columns that, together, uniquely identify each record in a table.
6. **Super Key:** A set of one or more attributes that can uniquely identify records, including candidate keys and additional attributes.
7. **Alternate Key:** Another candidate key that is not chosen as the primary key.
8. **Surrogate Key:** When a table doesnot have a Primary or Composite key. A system-generated unique identifier used as the primary key when a natural primary key is not available or is not suitable for performance or security reasons.

example :

1. **Product_ID (Primary Key):** A unique identifier for each product.
2. **Product_Name:** The name of the product.
3. **Category_ID (Foreign Key):** A reference to the primary key of the "Categories" table, linking products to their respective categories.
4. **SKU (Unique Key):** A unique Stock Keeping Unit assigned to each product.
5. **Barcode (Unique Key):** A unique barcode for each product.
6. **Price:** The price of the product.
7. **Stock_Quantity:** The current stock quantity of the product.
8. **Product_Code (Composite Key):** A combination of Product_ID and Category_ID that uniquely identifies each product in the table.

Here's the modified sample table with some example data:

Product_ID	Product_Name	Category_ID	SKU	Barcode	Price	Stock_Quantity	Product_Code
1	Laptop	101	LAP-001	123456789012	$800.00	50	1-101
2	Smartphone	102	PHN-002	987654321098	$500.00	100	2-102
3	Headphones	103	HPD-003	456789012345	$50.00	200	3-103

In this updated example, Product_ID remains the primary key, Category_ID is the foreign key, SKU and Barcode are unique keys, and Product_Code is a composite key combining Product_ID and Category_ID to uniquely identify each product.

Criteria to become Primary Key:

To become a primary key of a table, a column must fulfill the following criteria:

1. **Uniqueness:** Every value in the column must be unique; no two rows in the table can have the same value for the primary key column.
2. **Non-Nullability:** The primary key column must not allow NULL values. Each row in the table must have a valid value for the primary key.
3. **Irreducibility:** The primary key should be minimal, meaning it should consist of the smallest number of columns required to uniquely identify each row in the table.
4. **Stability:** The primary key value should not change over time, as it serves as a stable identifier for each row.
5. **Unchangeability:** The primary key value should not be modified after its initial insertion into the table to maintain data consistency.
6. **Uniformity:** The data type and format of the primary key column should be consistent across all rows in the table.

By satisfying these criteria, a column can be designated as the primary key of a table, ensuring data integrity and efficient data retrieval through fast indexing.

Entity

Anything which can be a part of a table.

eg : student, restuarent, car number plate, age

Cardinality of Relationships

Cardinality refers to the number of unique values or tuples (rows) present in a specific relation (table) of the database. It provides insight into the uniqueness and uniqueness constraints of the data within that relation.

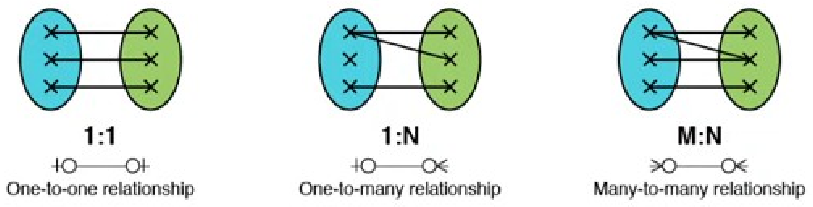
There are three main types of cardinality:

1. **One-to-One (1:1) Cardinality:** In a one-to-one cardinality, each value in one table's column is related to exactly one value in another table's column, and vice versa. This relationship indicates a strict and unique pairing between rows in the two tables.
   - eg : each person is assigned a unique government-issued identification number (such as a Social Security Number). Each identification number corresponds to only one individual, and vice versa. This is an example of one-to-one cardinality.
2. **One-to-Many (1:N) Cardinality:** In a one-to-many cardinality, each value in one table's column can be related to multiple values in another table's column, but each value in the second table's column is related to only one value in the first table's column. This type of cardinality is the most common in database relationships.
   - eg : library database where each book has an ISBN (International Standard Book Number). Each ISBN can be associated with only one book, but each book can have multiple copies in the library. This represents a one-to-many cardinality.
3. **Many-to-Many (N:N) Cardinality:** In a many-to-many cardinality, each value in one table's column can be related to multiple values in another table's column, and vice

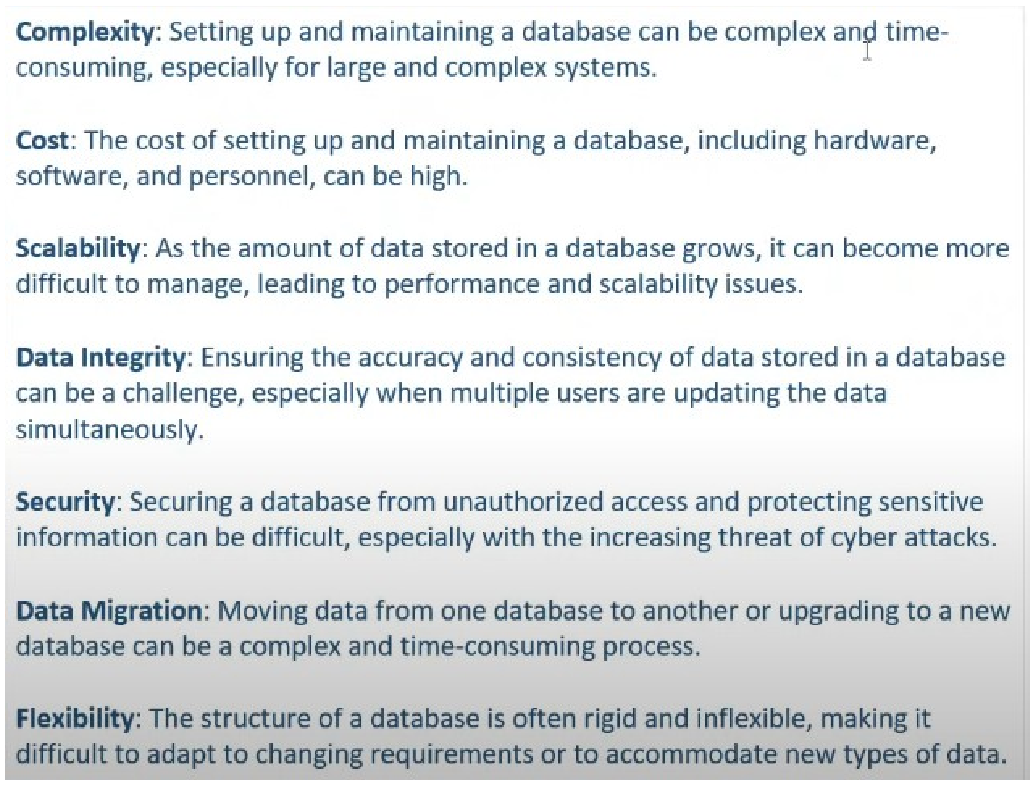
versa. This relationship requires an intermediate table (often called a junction or link table) to create unique combinations of values between the two tables.

- eg : In a university database, students can enroll in multiple courses, and each course can have multiple students. Therefore, there is a many-to-many relationship between students and courses, requiring an intermediate link table to track the enrollments.

Cardinality plays a crucial role in designing database schemas and establishing relationships between tables. Understanding the cardinality of relations helps ensure data consistency, efficiency, and appropriate database normalization.



Drawbacks of databases:



Complexity: Setting up and maintaining a database can be complex and time-consuming, especially for large and complex systems.

Cost: The cost of setting up and maintaining a database, including hardware, software, and personnel, can be high.

Scalability: As the amount of data stored in a database grows, it can become more difficult to manage, leading to performance and scalability issues.

Data Integrity: Ensuring the accuracy and consistency of data stored in a database can be a challenge, especially when multiple users are updating the data simultaneously.

Security: Securing a database from unauthorized access and protecting sensitive information can be difficult, especially with the increasing threat of cyber attacks.

Data Migration: Moving data from one database to another or upgrading to a new database can be a complex and time-consuming process.

Flexibility: The structure of a database is often rigid and inflexible, making it difficult to adapt to changing requirements or to accommodate new types of data.

Special character use: 'kumar\'s'

View all the databases:

Create database:

Use the current dabase:

NOTE : don't use databse keyword while selecting the database

See the tables present in the database:

DROP DATABASE:

UNIQUE KEY:

In SQL, a unique key is a constraint that ensures that the values in a particular column or set of columns are unique across all rows in a table.

There are two ways to create a unique key in SQL

- **1. Single or multiple unique keys**

Creating two separate unique keys: In this approach, you create two separate unique keys, each on a different column. For example, consider a table named "Employee" with columns "EmpID", "FirstName", and "LastName". You can create two separate unique keys, one on the "EmpID" column and another on the "FirstName" and "LastName" columns:

- **2. Combination of columns as a unique key**

Combining two columns as a single unique key: In this approach, you create a single unique key on a combination of two columns. For example, consider the same "Employee" table:

Difference between PRIMARY KEY and UNIQUE KEY

In SQL, both "PRIMARY KEY" and "UNIQUE KEY" are constraints used to ensure the uniqueness of values in a table. However, there are some differences between them:

- Cardinality - A primary key constraint must be unique across all rows in the table, and cannot contain null values. A unique key constraint must also be unique, but can contain null values.
- Number of Constraints - A table can have only one primary key constraint, but can have multiple unique key constraints.
- Indexing - A primary key creates a clustered index by default, while a unique key creates a non-clustered index by default. Clustered indexes physically reorder the

rows in the table to match the index order, while non-clustered indexes create a separate structure that points to the original data.

- Reference in Foreign Key - A foreign key in another table must reference the primary key of the referenced table. A foreign key can also reference a unique key, but not recommended as it increases complexity.

Difference between Clustered and Non-clustered Indexes

- **clustered index** is a special type of index that physically reorders the rows of a table to match the order of the index. This means that the data in the table is stored in the same order as the clustered index. As a result, **a clustered index is often used as the primary key of a table**, as it can provide fast access to rows based on the primary key value. **In a table there can only be 1 clustered index.** They are physically ordered in the actual table.

- On the other hand, a **non-clustered index** is a type of index that does not physically reorder the rows of a table. Instead, it creates a separate structure that maps the values of one or more columns in the table to their physical location. When a query is executed that uses a non-clustered index, the database must first look up the index to find the physical location of the data, and then retrieve the actual data from the table.

In summary, the main difference between clustered and non-clustered indexes is that a clustered index physically reorders the rows of a table to match the index, while a non-clustered index provides a mapping of values to physical locations but does not change the physical order of the table.

Scaler and Aggregate functions:

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

- **Aggregate functions :** These functions are used to do operations from the values of the column and a single value is returned. It operate on multiple rows of data and return a single value that summarizes the data. Examples of aggregate functions include SUM, AVG, COUNT, MIN, and MAX. Aggregate functions are typically used in the SELECT clause of a query to calculate summary information for a group of rows, such as the total sum of values or the average of values.

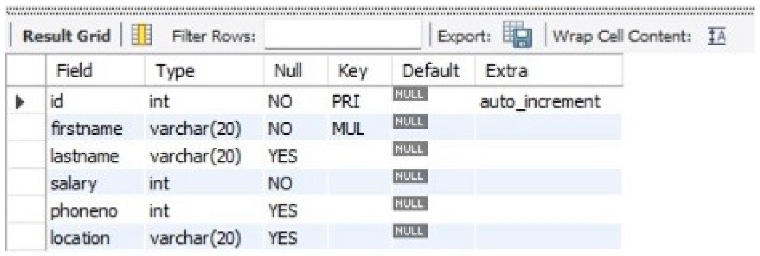
- **Scalar functions :** These functions are based on user input, these too return a single value. Scalar functions operate on a single row of data and return a single value for that row. Examples of scalar functions include mathematical operations like SUM, AVG, and COUNT, as well as string functions like UPPER, LOWER, and SUBSTRING. Scalar functions can be used in the SELECT and WHERE clauses of a query to modify or manipulate individual values in the result set.

Default values

It is the value that a column will take if no value is specified during an insert operation. You can specify a default value for a column in the table definition using the "DEFAULT" keyword. The syntax for setting a default value in SQL is as follows:

Creating tables:

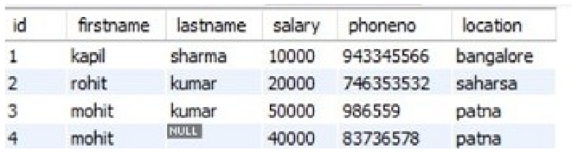
Description of a table



Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
firstname	varchar(20)	NO	MUL	NULL	
lastname	varchar(20)	YES		NULL	
salary	int	NO		NULL	
phoneno	int	YES		NULL	
location	varchar(20)	YES		NULL	

INSERTING VALUES in a table:

>> as id is in AUTO_INCREMENT we donot need to pass it in the column section



id	firstname	lastname	salary	phoneno	location
1	kapil	sharma	10000	943345566	bangalore
2	rohit	kumar	20000	746353532	saharsa
3	mohit	kumar	50000	986559	patna
4	mohit	NULL	40000	83736578	patna

>> we can pass NULL as value if NOT NULL is not defined for that column

NOTE : Don't use VALUES keyword when inserting more than one row from one table to another

eg : select statement inside INSERT :

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Deleting tables in a database:

To delete a table from a database in SQL, you can use the "DROP TABLE" statement. The basic syntax for this statement is as follows:

Difference between DROP and TRUNCATE

In SQL, "DROP" and "TRUNCATE" are two separate statements used to remove data from a table.

The "DROP" statement is used to delete a table completely. When you use "DROP", the table and all of its data are permanently deleted and cannot be recovered. The basic syntax for the "DROP" statement is as follows:

The "TRUNCATE" statement, on the other hand, is used to remove all data from a table, but the table structure remains intact. The basic syntax for the "TRUNCATE" statement is as follows:

In summary, the "DROP" statement is used to completely remove a table, while the "TRUNCATE" statement is used to remove all data from a table, but leave the table structure intact.

Difference between DELETE and DROP

In SQL, "DELETE" and "DROP" are two separate statements used to remove data from a table.

- The "DELETE" statement is used to remove one or more rows from a table based on a specified condition. The basic syntax for the "DELETE" statement is as follows:
- The "DROP" statement, on the other hand, is used to delete the entire table, including its structure and data. The basic syntax for the "DROP" statement is as follows:

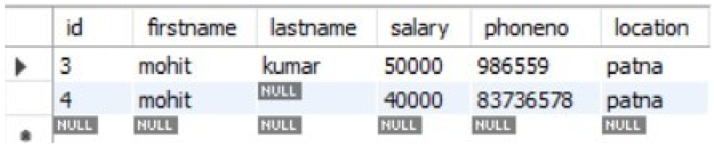
In summary, the "DELETE" statement is used to remove one or more rows from a table based on a specified condition, while the "DROP" statement is used to completely delete the entire table and its data.

DML : CRUD operations

CRUD stands for "Create, Read, Update, and Delete" and refers to the four basic operations that can be performed on a database. These operations are the foundation of database management. The equivalent CRUD operations in SQL are:

- **Create -** This operation is used to insert new data into a database. The "INSERT INTO" statement is used to perform this operation:
- **Read -** This operation is used to retrieve data from a database. The "SELECT" statement is used to perform this operation:
- **Update -** This operation is used to modify existing data in a database. The "UPDATE" statement is used to perform this operation:
- **Delete -** This operation is used to remove data from a database. The "DELETE" statement is used to perform this operation:

WHERE clause()



	id	firstname	lastname	salary	phoneno	location
▶	3	mohit	kumar	50000	986559	patna
	4	mohit	NULL	40000	83736578	patna
*	NULL	NULL	NULL	NULL	NULL	NULL

NOTE : SQL is case insensitive!!!

for it be case sentisitve use **binary**

ALIASING()

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

UPDATE() - use SET

Updating rows in the table

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

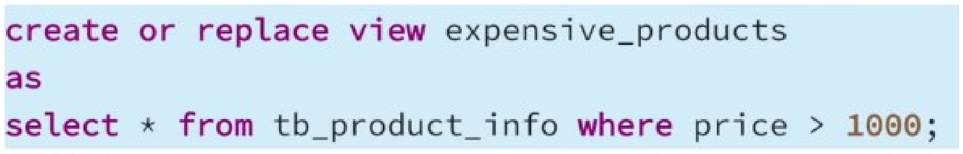
updating salary by 700

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

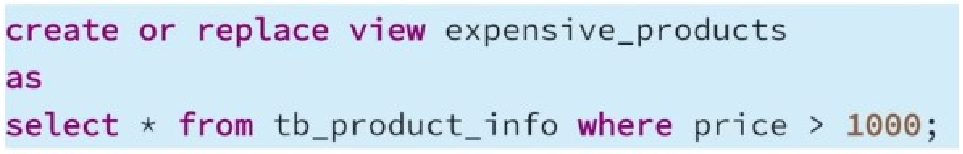
ALTER - to alter schema of Table

The ALTER command in SQL is used to modify the structure of a database table after it has already been created. It can be used to add, modify or drop columns, add or drop constraints, or modify the properties of existing columns or constraints.

1. Adding a new column:



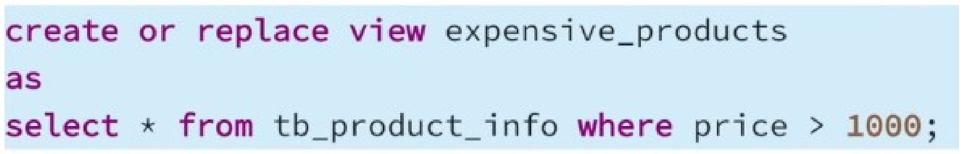
```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```



```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

ADDING COLUMN between 2 Columns

2. Modifying an existing column: use MODIFY



```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

3. Dropping a column:

4. Adding a new constraint:

add primary key id column

5. Dropping a constraint:

drop primary key id column

NOTE : donot need to specify column name 'id' as a parameter while dropping PRIMARY KEY

It's important to note that you do not need to specify the name of the column or columns that make up the primary key when dropping the primary key constraint. The DROP PRIMARY KEY clause alone is enough to remove the constraint.

DROP a UNIQUE Constraint :

NOTE : Constraints cannot be Modeified, they need to be deleted and added again in the correct format

6. Renaming column names

note: use tilde ` insted of ' or " for column name

DDL vs DML

DDL stands for Data Definition Language.

It includes the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database. Examples of DDL statements include **CREATE, ALTER, DROP, TRUNCATE.**

DML stands for Data Manipulation Language.

It includes the SQL commands that can be used to manage data stored in the database. This includes inserting, updating, and deleting data. Examples of DML statements include **SELECT, INSERT, UPDATE, DELETE**

____In short, DDL is used to create and modify database structure, while DML is used to manage the data stored in the database.____

DATA INTEGRITY

Session 31 - SQL DDL Commands : https://www.youtube.com/watch?v=ny1mh6VUpnQ

Data integrity in databases refers to the accuracy, completeness, and consistency of the data stored in a database. It is a measure of the reliability and trustworthiness of the data and ensures that the data in a database is protected from errors, corruption, or unauthorized changes.

There are various methods used to ensure data integrity, including:

- **Constraints:** Constraints in databases are rules or conditions that must be met for data to be inserted, updated, or deleted in a database table. They are used to enforce the integrity of the data stored in a database and to prevent data from becoming inconsistent or corrupted.
- **Transactions:** a sequence of database operations that are treated as a single unit of work.
- **Normalization:** a design technique that minimizes data redundancy and ensures data consistency by organizing data into separate tables.

CONSTRAINTS

Constraints in MySQL are rules that enforce data integrity and consistency. They specify the conditions that data must meet in order to be inserted, updated, or deleted from a table. Constraints ensure that the data in a table remains consistent and meets certain requirements.

There are several types of constraints in MySQL:

- **PRIMARY KEY:** Enforces uniqueness and defines a column or a combination of columns as the primary key of a table.

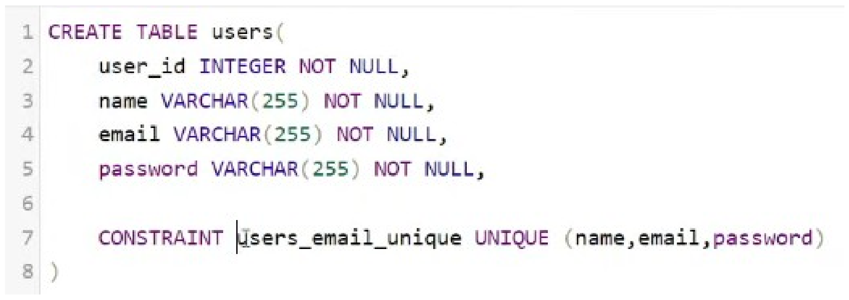
- **FOREIGN KEY:** Enforces referential integrity and ensures that the values in a foreign key column match the values in the referenced column of a referenced table.
- **UNIQUE:** Enforces uniqueness and ensures that the values in a column or a combination of columns are unique within the table.
- **NOT NULL:** Enforces non-NULL values and ensures that a value is entered in a column for every row in a table.
- **CHECK:** Enforces conditional constraints and allows you to specify conditions that data must meet in order to be inserted, updated, or deleted from a table.
- **DEFAULT:** Specifies a default value for a column.
- **AUTO-INCREMENT:**

Constraints can be specified when creating a table, or they can be added or modified later using ALTER TABLE statements. They are an important tool for maintaining the integrity and consistency of your data.

In this query:

- `EmployeeID` is an ***auto-incrementing primary key column*** representing the unique identifier for each employee.
- `EmployeeName` is a VARCHAR column storing the name of the employee, and it ***cannot be NULL***.
- `EmployeeCode` is an ***INT column with a UNIQUE constraint,*** ensuring each employee has a unique employee code.
- `Salary` is a DECIMAL column representing the ***salary of the employee, with a default value of 0.00.***
- `Department` is a VARCHAR column representing the department in which the employee works. It has a ***CHECK constraint to ensure the department name is longer than 2 characters.***
- `HireDate` is a DATE column representing the date on which the employee was hired.
- `ManagerID` is an INT column serving as a ***foreign key, referencing the primary key*** ***`ManagerID`*** in the table `Managers`. It establishes a relationship between employees and their managers.

Q - constraint is comibnation of name, email and password cannot be duplicate:



```
CREATE TABLE users(
    user_id INTEGER NOT NULL,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,

    CONSTRAINT users_email_unique UNIQUE (name,email,password)
)
```

CHECK

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a column it will allow only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

FOREIGN KEY

- A foreign key is a column in a table that is a reference to the primary key of another table.
- It is used to establish a relationship between two tables, ensuring data integrity and consistency.
- The purpose of a foreign key is to prevent actions that would create orphaned records in the child table (referenced table) when records in the parent table (referencing table) are deleted or updated.
- **Foreign Key constraint is used to prevent actions that would destroy links between two tables**
- ***The table with the foreign key is called child table, the table with the primary key is called the parent table or refrenced table***

because if a course id is not present in courses table we cannot add it in students table

-- Create the referenced table

CREATE TABLE Departments (DepartmentID INT PRIMARY KEY, DepartmentName VARCHAR(100), -- Other columns as needed);

-- Add a foreign key constraint to link Employees table to Departments table

ALTER TABLE Employees ADD CONSTRAINT fk_DepartmentID FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID);

Benefits of adding foreign key:

- **Enforcing referential integrity :** A foreign key constraint is used to enforce referential integrity, which ensures that data entered into the database is consistent and accurate. The foreign key constraint ensures that a value in the foreign key column of a table must match a value in the primary key column of another table.
- **Preventing orphaned records :** A foreign key constraint helps to prevent orphaned records, which are records that have no corresponding parent record in the referenced table. When a foreign key constraint is in place, the database will not allow you to delete a record in the referenced table if there are matching records in the referencing table.
- **Simplifying data retrieval :** By establishing relationships between tables through foreign key constraints, you can simplify data retrieval by joining tables to retrieve related data.
- **Improving database performance :** By establishing relationships between tables through foreign key constraints, the database can use the relationships to optimize query performance by using indexes on the foreign key and primary key columns.

Overall, adding a foreign key and referencing it is an important aspect of database design and helps to ensure the integrity, accuracy, and performance of your database.

How Foreign Key can be violated

Foreign key constraints can be violated in several ways. Some common violations include:

- Inserting a value into the foreign key column that doesn't match a value in the referenced column: This can occur when you try to insert a row into a table that has a foreign key constraint, and the value in the foreign key column doesn't match a value in the referenced column of the referenced table.
- Deleting a row from the referenced table: This can occur when you try to delete a row from the referenced table that is being referenced by one or more rows in the referencing table.
- Updating a value in the referenced column: This can occur when you try to update a value in the referenced column that is being referenced by one or more rows in the referencing table.
- Setting the foreign key column to NULL: This can occur when you try to insert or update a row in the referencing table and set the value in the foreign key column to NULL, but the foreign key constraint requires non-NULL values.
- Duplicate values in the foreign key column: This can occur when you try to insert or update a row in the referencing table and the value in the foreign key column is not unique, but the foreign key constraint requires unique values.

adding courseno from students table and courseid from courses table as FOREIGN KEY

CASCADE KEYS

CASCADE keys refer to the cascading effects that occur when performing certain operations on a table that has a foreign key constraint. When a CASCADE action is specified for a foreign key, it means that changes made to the referenced primary key in the parent table will automatically propagate to the child table with the foreign key.

There are several types of CASCADE actions that can be applied to foreign keys:

- **CASCADE UPDATE:** When the primary key value in the parent table is updated, the corresponding foreign key value in the child table will also be updated automatically.
- **CASCADE DELETE:** When a row is deleted from the parent table, all related rows in the child table with matching foreign key values will also be automatically deleted.

Here's an example SQL query that demonstrates the implementation of a CASCADE DELETE foreign key constraint:

In this example, the foreign key constraint `fk_AuthorID` in the `Books` table has the `ON DELETE CASCADE` option, meaning that when an author with `AuthorID = 1` is deleted from the `Authors` table, all related books with `AuthorID = 1` will also be automatically deleted from the `Books` table. This ensures that the database remains consistent and avoids orphaned records in the child table.

USING TIMESTAMP

MySQL comes with the following data types for storing a date or a date/time value in the database:

- **DATE - format YYYY-MM-DD**
- **DATETIME - format: YYYY-MM-DD HH:MI:SS**
- **TIMESTAMP - format: YYYY-MM-DD HH:MI:SS**
- **YEAR - format YYYY or YY**

In MySQL, you can use the **NOW()** function to insert the current date and time as the value of a TIMESTAMP column.

Here's an example of how you would insert a new row into a table with a TIMESTAMP column named created_at:

OR

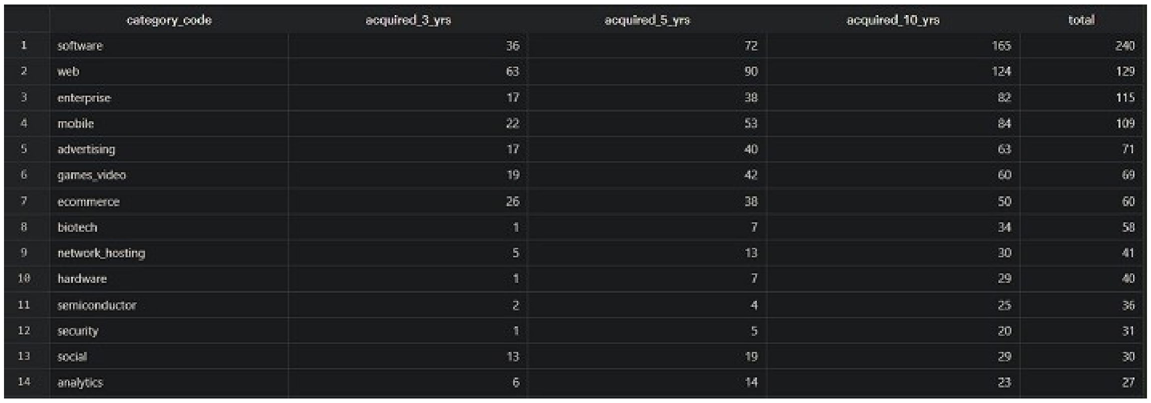
You can also set the created_at column to automatically have the current date and time inserted whenever a new row is inserted, by defining the column with the DEFAULT keyword and the NOW() function:

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Chaning into timestamp datatype using ::timestamp

Write a query that counts the number of companies acquired within 3 years, 5 years, and 10 years of being founded (in 3 separate columns). Include a column for total companies acquired as well. Group by category and limit to only rows with a founding date.



	category_code	acquired_3_yrs	acquired_5_yrs	acquired_10_yrs	total
1	software	36	72	165	240
2	web	63	90	124	129
3	enterprise	17	38	82	115
4	mobile	22	53	84	109
5	advertising	17	40	63	71
6	games_video	19	42	60	69
7	ecommerce	26	38	50	60
8	biotech	1	7	34	58
9	network_hosting	5	13	30	41
10	hardware	1	7	29	40
11	semiconductor	2	4	25	36
12	security	1	5	20	31
13	social	13	19	29	30
14	analytics	6	14	23	27

DISTINCT()

DISTINCT COMBINATION OF 2 COLUMNS:

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

ORDER BY()

ORDER BY on 2 columns, one with ASCENDING other with DESCENDING:

WILDCARD

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the operator. The operator is used in a clause to search for a specified pattern in a column. LIKE WHERE

- **% : Matches zero or more characters.**

For example, the pattern '%m%' matches any string that contains an "m" character, such as "mat", "rampart", or "maze".

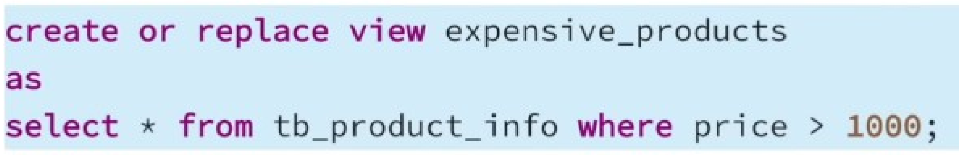
- **_ (underscore):** Matches a single character.

For example, the pattern '_og' matches any string that contains exactly three characters and ends with "og", such as "dog" or "fog".

These wildcard characters can be used with the LIKE operator and the REGEXP operator in SQL to perform flexible and powerful searches on data stored in tables. Note that the use of wildcard characters may impact the performance of your queries, especially if the data you're searching is large or complex.

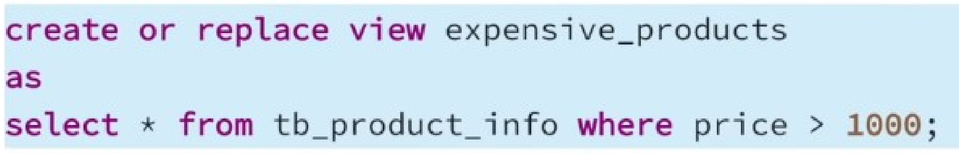
LIKE ()

patterns of location which starts with pat or ran:



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

name of students with 2 charcaters



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

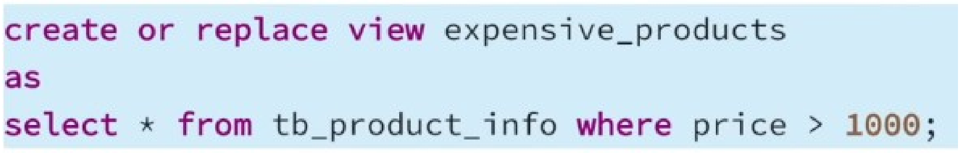
starts with letter 'a' and have atleast 3 characters :

WHERE CustomerName LIKE __'a_%_%'__

Finds any values that starts with "a" and are at least 3 characters in length

ORDER OF EXECUTION :

https://www.youtube.com/watch?v=JUCTcHsNkyM&list=PLtgiThe4j67rAoPmnCQmcgLS4iIc5ungg&index=8



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Flash jumped Wonder girl having se* during office

The order of execution in SQL refers to the order in which SQL statements are executed and the order in which the data is processed. In general, the order of execution in SQL can be summarized as follows:

- **From Clause:** The database engine selects the data from the specified tables and views.
- **Where Clause:** The database engine filters the data based on the conditions specified in the WHERE clause.
- **Group By Clause:** The database engine groups the data based on the columns specified in the GROUP BY clause.
- **Having Clause:** The database engine filters the grouped data based on the conditions specified in the HAVING clause.
- **Select Clause:** The database engine applies the calculations and functions specified in the SELECT clause.
- **Order By Clause:** The database engine sorts the data based on the columns specified in the ORDER BY clause.
- **Limit Clause:** The database engine limits the number of rows returned by the query based on the value specified in the LIMIT clause.

AGGREGATE FUNCTIONS

- **AVG:** returns the average value of a column.
- **COUNT:** returns the number of rows in a column.
- **SUM:** returns the sum of values in a column.
- **MIN:** returns the minimum value in a column.
- **MAX:** returns the maximum value in a column.

1. COUNT()

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

2. Group BY()

THIS WON"T WORK

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

GROUPBY on multiple columns

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

GROUP BY - CampusX

https://youtu.be/nsKcmOly0UY?t=2399

Group smartphones by brand and get the count, average price, max rating, avg screen size, and avg battery capacity

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

GROUP BY on 2 columns

Group smartphones by the brand and processor brand and get the average price and the average primary camera resolution (rear)

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

3. MIN

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

This won't work : student name with min years of experience

instead this will be correct

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

4. MAX

MAX number of years_of_joing from each student_company

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

5. SUM()

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

6. AVERAGE()

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

7. STD() - standard deviation

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

8. VARIANCE() - Variance

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

ROUND OF DECIMALS:

while creating the column instead of INT we can pass DECIMAL(5,2) i.e 5 digits before decimal and in that 2 digit after decimal

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

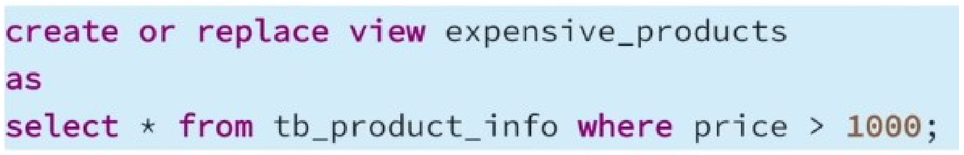
BETWEEN ()

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

IN()

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

NOT IN()



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

CAST ()

The CAST() function converts a value (of any type) into the specified datatype.

ROUND()

The ROUND() function rounds a number to a specified number of decimal places.

eg : Round the number to 2 decimal places

```
SELECT ROUND(135.375657, 2)
```

135.38

CEIL ()

The CEIL() function returns the smallest integer value that is bigger than or equal to a number.

Note: This function is equal to the CEILING() function.

26

FLOOR()

The FLOOR() function returns the largest integer value that is smaller than or equal to a number.

Note: Also look at the ROUND(), CEIL(), CEILING(), TRUNCATE(), and DIV functions.

25

TRUNCATE ()

The TRUNCATE() function truncates a number to the specified number of decimal places.

eg :

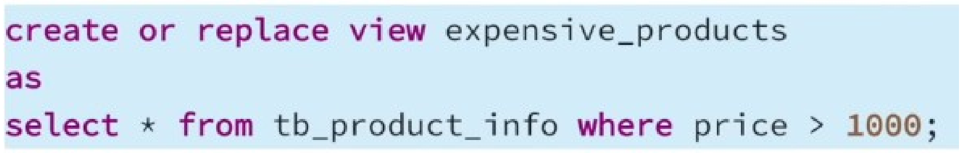
eg : Return a number truncated to 0 decimal places:

Parameter → Description

number → Required. The number to be truncated

decimals → Required. The number of decimal places to truncate to

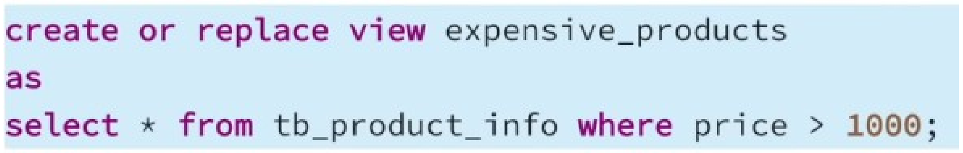
ABS() : Absolute value of a number i.e negative becomes positive number



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

3rd largest:

find the phone with 3rd largest battery

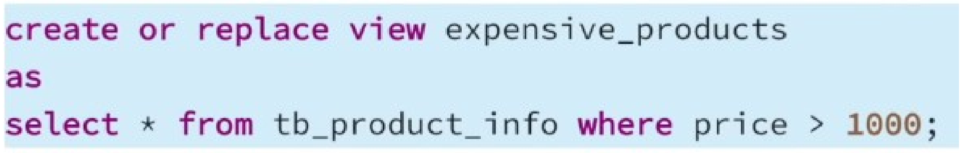


```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

LIMIT 2,1 means go to 2nd row and 1 means take the next row after 2nd row.

3rd and 4th Largest:

LIMIT 2,2 means go to 2nd row and 2 means take the next 2 rows i.e 3rd and 4th after 2nd row.



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

CASES()

the CASE statement is used to control the flow of a query by allowing you to perform conditional logic in a query.

The CASE statement is often used in the SELECT clause of a query to transform data on the fly and to provide an alternate value if a condition is met.

- The CASE statement always goes in the SELECT clause:
- **CASE must include the following components: WHEN, THEN, and END.** ELSE is an optional component.
- You can make any conditional statement using any conditional operator (like WHERE) between WHEN and THEN. This includes stringing together multiple conditional statements using AND and OR.

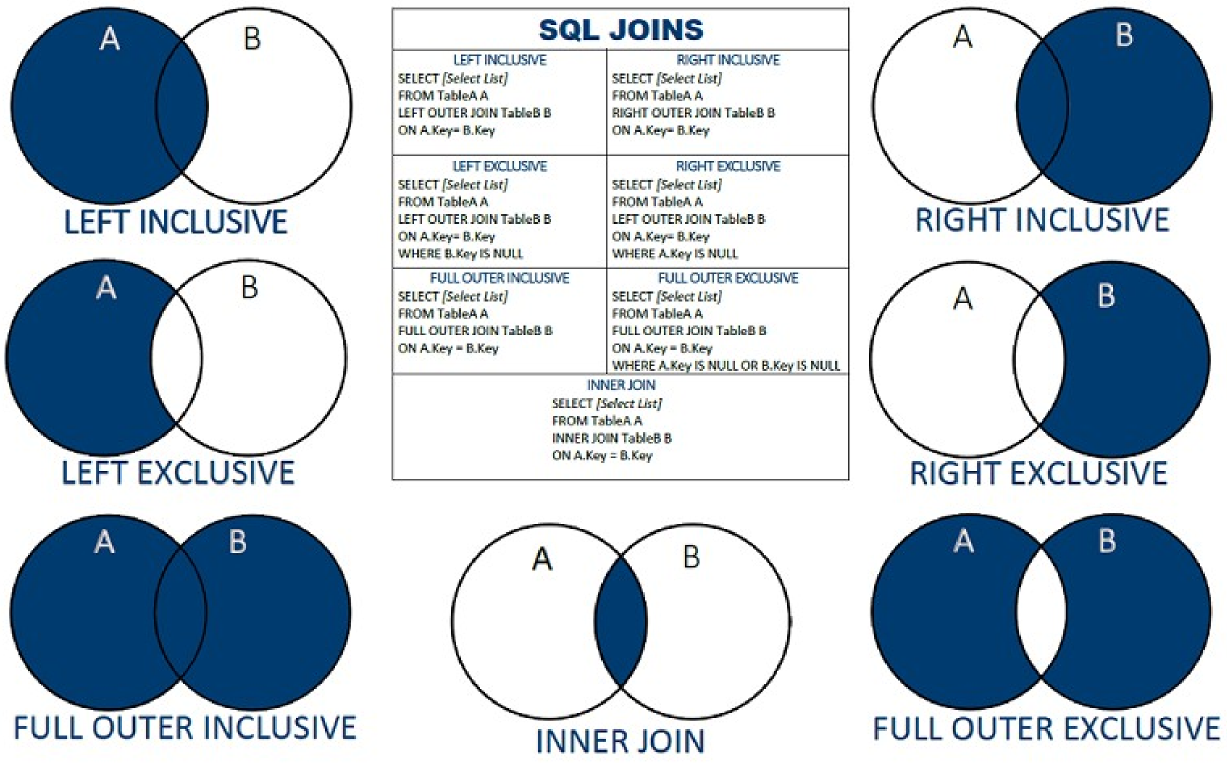
- You can include multiple WHEN statements, as well as an ELSE statement to deal with any unaddressed conditions.

for years_of_experience with less than 4 label as fresher between 4 to 8 as lead else MANAGER in a new column

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

JOINS

https://www.youtube.com/watch?v=9joG6P9ZhPM&list=PLtgiThe4j67rAoPmnCQmcgLS4iIc5ungg&index=12



SQL JOINS

LEFT INCLUSIVE	RIGHT INCLUSIVE

SELECT *[Select List]*
FROM TableA A
LEFT OUTER JOIN TableB B
ON A.Key= B.Key
	SELECT *[Select List]*
	FROM TableA A
	RIGHT OUTER JOIN TableB B
	ON A.Key= B.Key

LEFT EXCLUSIVE	**RIGHT EXCLUSIVE**

SELECT *[Select List]*
FROM TableA A
LEFT OUTER JOIN TableB B
ON A.Key= B.Key
WHERE B.Key IS NULL
	SELECT *[Select List]*
	FROM TableA A
	LEFT OUTER JOIN TableB B
	ON A.Key= B.Key
	WHERE A.Key IS NULL

FULL OUTER INCLUSIVE	**FULL OUTER EXCLUSIVE**

SELECT *[Select List]*
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
	SELECT *[Select List]*
	FROM TableA A
	FULL OUTER JOIN TableB B
	ON A.Key = B.Key
	WHERE A.Key IS NULL OR B.Key IS NULL

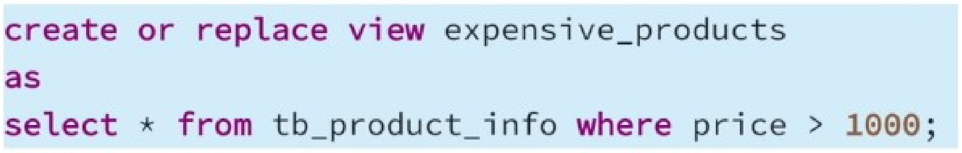
INNER JOIN	

SELECT *[Select List]*
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key	

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

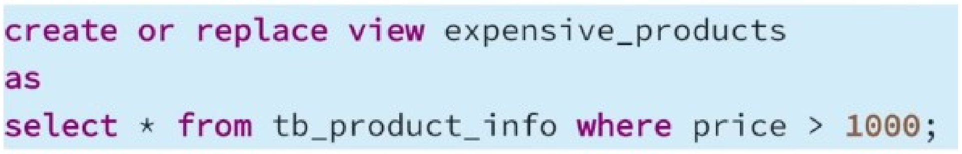
In SQL, there are several types of joins that you can use to combine data from two or more tables. The main types of joins are:

- **Inner Join:** Returns only the rows that have matching values in both tables.



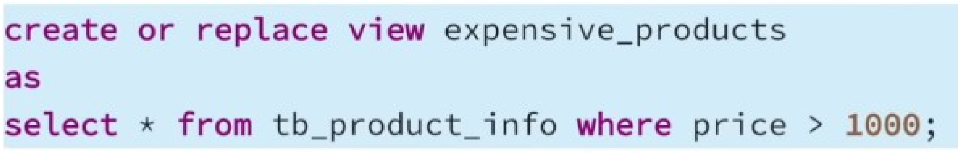
```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

- **Left Join (or Left Outer Join):** Returns all the rows from the left table (table1), and the matching rows from the right table (table2). If there is no match, NULL values will be returned for right table's columns.



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

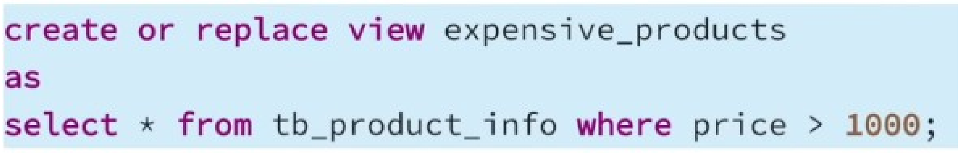
- **Right Join (or Right Outer Join):** Returns all the rows from the right table (table2), and the matching rows from the left table (table1). If there is no match, NULL values will be returned for left table's columns.



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

- **Full Outer Join:** Returns all the rows from both tables, and matching rows will have matching values. If there is no match, NULL values will be returned for non-matching columns.

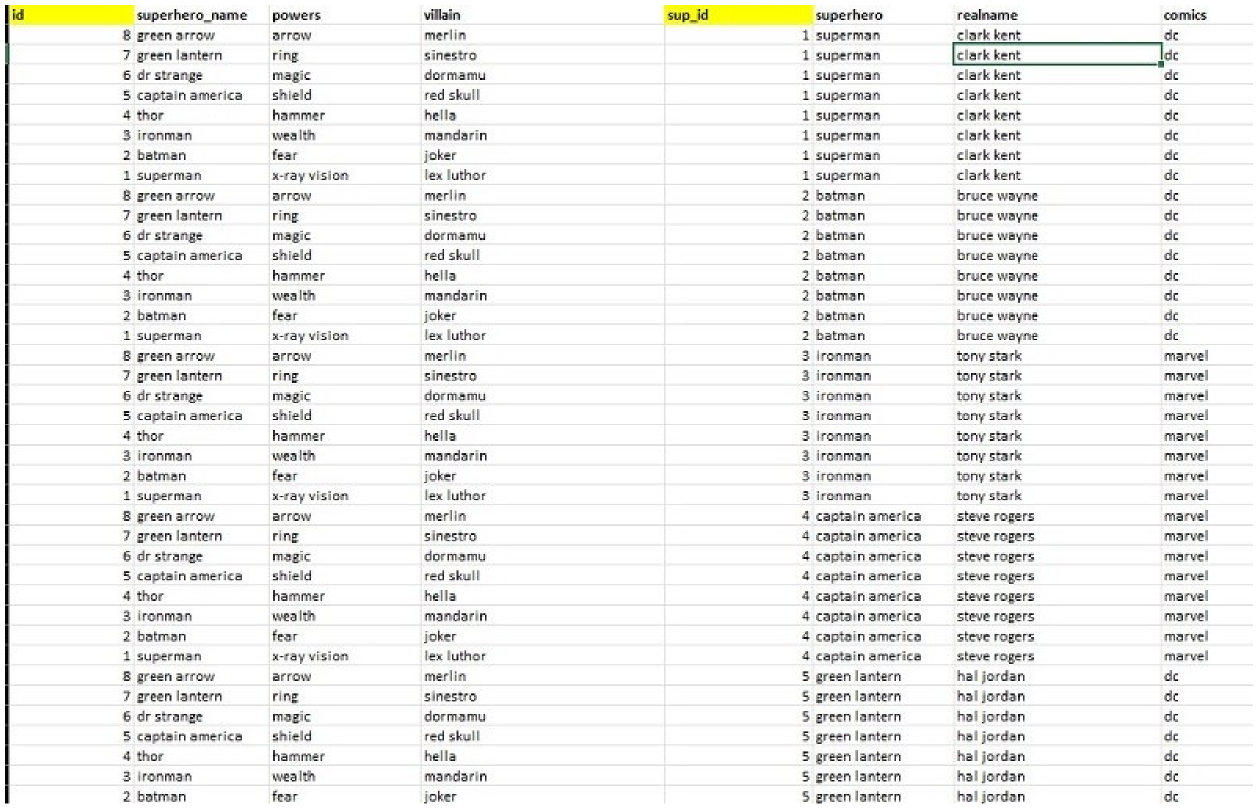
NOTE : MYSQL doesnot have FULL OUTER JOIN instead use combining a LEFT JOIN and a RIGHT JOIN:



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

- **Cross Join:** Returns the Cartesian product of the two tables, which means it returns every possible combination of rows from both tables.

NOTE : in CROSS JOIN we donot need a common key

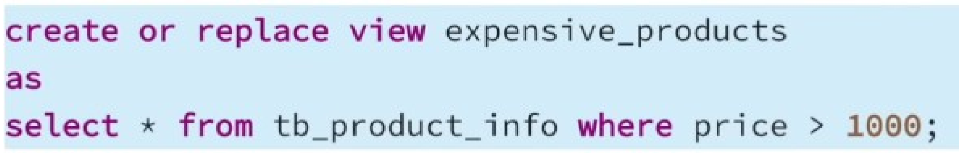


id	superhero_name	powers	villain	sup_id	superhero	realname	comics
8	green arrow	arrow	merlin	1	superman	clark kent	dc
7	green lantern	ring	sinestro	1	superman	clark kent	dc
6	dr strange	magic	dormamu	1	superman	clark kent	dc
5	captain america	shield	red skull	1	superman	clark kent	dc
4	thor	hammer	hella	1	superman	clark kent	dc
3	ironman	wealth	mandarin	1	superman	clark kent	dc
2	batman	fear	joker	1	superman	clark kent	dc
1	superman	x-ray vision	lex luthor	1	superman	clark kent	dc
8	green arrow	arrow	merlin	2	batman	bruce wayne	dc
7	green lantern	ring	sinestro	2	batman	bruce wayne	dc
6	dr strange	magic	dormamu	2	batman	bruce wayne	dc
5	captain america	shield	red skull	2	batman	bruce wayne	dc
4	thor	hammer	hella	2	batman	bruce wayne	dc
3	ironman	wealth	mandarin	2	batman	bruce wayne	dc
2	batman	fear	joker	2	batman	bruce wayne	dc
1	superman	x-ray vision	lex luthor	2	batman	bruce wayne	dc
8	green arrow	arrow	merlin	3	ironman	tony stark	marvel
7	green lantern	ring	sinestro	3	ironman	tony stark	marvel
6	dr strange	magic	dormamu	3	ironman	tony stark	marvel
5	captain america	shield	red skull	3	ironman	tony stark	marvel
4	thor	hammer	hella	3	ironman	tony stark	marvel
3	ironman	wealth	mandarin	3	ironman	tony stark	marvel
2	batman	fear	joker	3	ironman	tony stark	marvel
1	superman	x-ray vision	lex luthor	3	ironman	tony stark	marvel
8	green arrow	arrow	merlin	4	captain america	steve rogers	marvel
7	green lantern	ring	sinestro	4	captain america	steve rogers	marvel
6	dr strange	magic	dormamu	4	captain america	steve rogers	marvel
5	captain america	shield	red skull	4	captain america	steve rogers	marvel
4	thor	hammer	hella	4	captain america	steve rogers	marvel
3	ironman	wealth	mandarin	4	captain america	steve rogers	marvel
2	batman	fear	joker	4	captain america	steve rogers	marvel
1	superman	x-ray vision	lex luthor	4	captain america	steve rogers	marvel
8	green arrow	arrow	merlin	5	green lantern	hal jordan	dc
7	green lantern	ring	sinestro	5	green lantern	hal jordan	dc
6	dr strange	magic	dormamu	5	green lantern	hal jordan	dc
5	captain america	shield	red skull	5	green lantern	hal jordan	dc
4	thor	hammer	hella	5	green lantern	hal jordan	dc
3	ironman	wealth	mandarin	5	green lantern	hal jordan	dc
2	batman	fear	joker	5	green lantern	hal jordan	dc

- **SELF JOIN :** A SELF JOIN in SQL is a regular join, but the table is joined with itself.

In other words, a SELF JOIN combines rows from the same table based on a related column between two or more rows within the same table.

To perform a SELF JOIN, you must specify an alias for the table because you're joining it with itself. This allows you to distinguish between the two copies of the same table in the join condition.

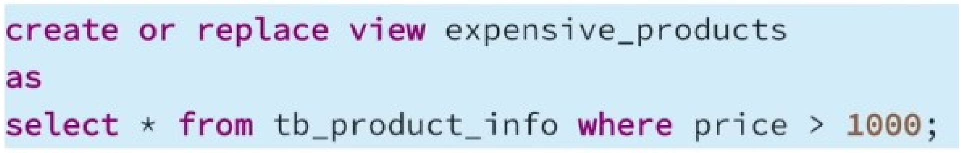


```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

NOTE : we are joining same table with itself, so both table names will be set as different

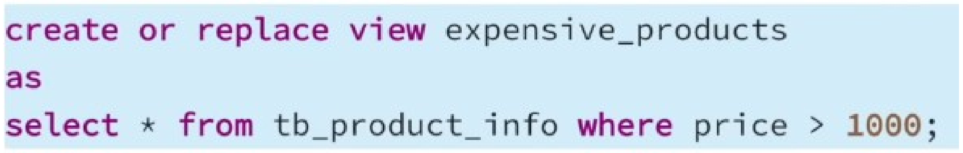
eg: fetch child name and their age corresponding to their parent and parent age:

parent_id will be matched with the same member_id and name will be fetched



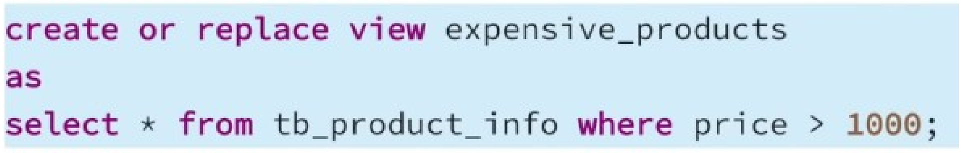
```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

using self join



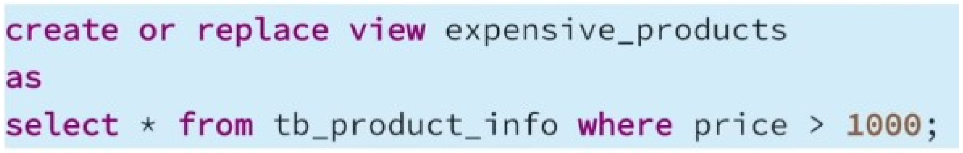
```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

using Left join (Left Exclusive)



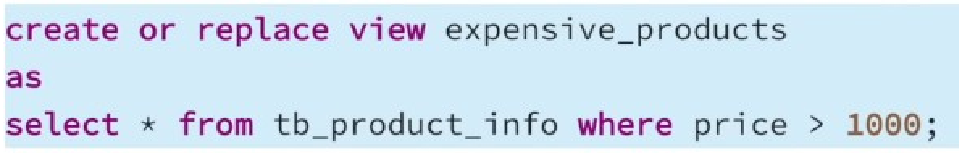
```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

using right join (Right Exclusive)



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

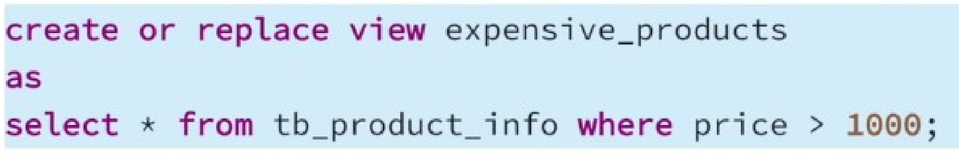
JOIN on basis of 2 columns :



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

JOINING 3 tables

JOINING 4 tables

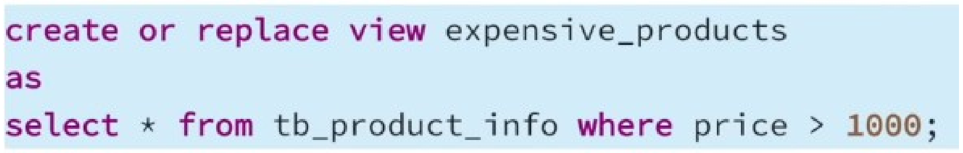


```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

SET Operations

1. **UNION:** The UNION operator is used to combine the results of two or more SELECT statements into a single result set. The UNION operator removes duplicate rows between the various SELECT statements.
2. **UNION ALL:** The UNION ALL operator is similar to the UNION operator, but it does not remove duplicate rows from the result set.
3. **INTERSECT:** The INTERSECT operator returns only the rows that appear in both result sets of two SELECT statements.
4. **EXCEPT:** The EXCEPT or MINUS operator returns only the distinct rows that appear in the first result set but not in the second result set of two SELECT statements.

EXCEPT : find customers who have never ordered



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

UNION

RULES to be followed:

- The number of columns in each table must be the same.
- The data types of the columns in each table must be compatible and correspond to each other in order.
- The names of the columns in the result set are taken from the first table, so it's a good idea to give the columns in both tables meaningful names to make the result set easier to understand.
- The tables must be from the same database or accessible from the same database connection.

Difference between UNION and UNION ALL()

The UNION and UNION ALL operators in SQL are used to combine the results of two or more SELECT statements into a single result set.

The main difference between UNION and UNION ALL is how duplicate rows are handled:

- **UNION:** The UNION operator removes duplicate rows from the final result set. If the same row is present in multiple SELECT statements, it will appear only once in the final result set.
- **UNION ALL:** The UNION ALL operator does not remove any duplicate rows. All rows, including duplicates, from each SELECT statement are included in the final result set.

Difference between WHERE and HAVING clause in SQL

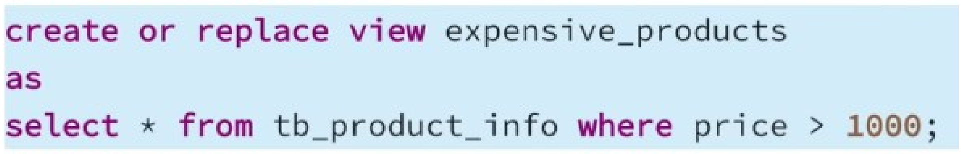
The WHERE and HAVING clauses in SQL are both used to filter rows from a result set based on certain conditions.

However, they are used in different contexts and have different purposes:

- **WHERE clause :** The WHERE clause is used to filter rows from a result set before aggregating the data. It filters rows based on the values in individual columns and returns only the rows that meet the specified conditions. The WHERE clause is applied to individual rows, and it filters out rows that do not meet the conditions.
- **HAVING clause :** The HAVING clause is used to filter rows from a result set after aggregating the data. It filters groups of rows based on the result of an aggregate function, such as SUM, AVG, COUNT, etc. The HAVING clause is applied to groups of rows, and it filters out groups that do not meet the conditions.

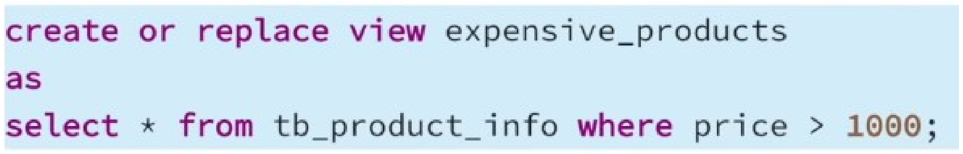
NOTE : After GROUP BY we can't use WHERE clause, we have to use HAVING clause

NOTE : what WHERE is to SELECT is what HAVING is to GROUP BY



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

only using where clause :



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

using where and having clause

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

HAVING clause - Campusx

https://youtu.be/nsKcmOly0UY?t=5262

find the avg rating of smartphone brands that have more than 20 phones

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

USING WHERE and HAVING together

Find the top 7 brands with the highest avg ram that has a refresh rate of at least 90 Hz and fast charging available and don't consider brands that have less than 10 phones

```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

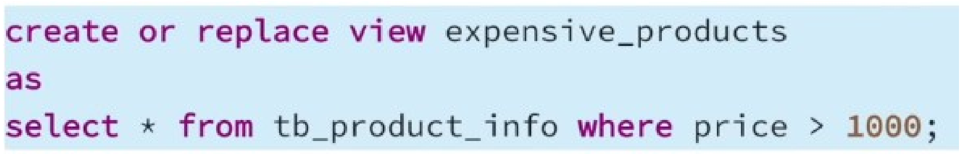
OVER clause

The OVER clause in MySQL is used to define a window function that performs a calculation across a set of rows that are related to the current row.

It is used with aggregate functions like SUM, AVG, MIN, MAX, etc. to perform calculations over a specified range of rows.

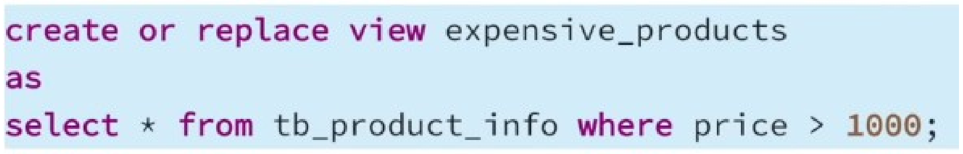
NOTE : if nothing is passed insider OVER() then it aggregates over entire column

USING JOIN



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

SAME using OVER AND PARTITION BY



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

PARTITION

PARTITION BY is a clause in the GROUP BY statement in MySQL. It divides the rows of a result set into partitions based on the values of the specified column or columns.

The purpose of using PARTITION BY is to provide an additional level of grouping within a GROUP BY statement. With it, you can perform aggregate operations (such as SUM, AVG, MIN, etc.) on each partition separately, rather than on the entire result set.

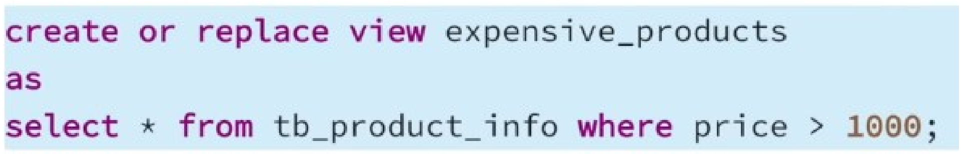
The PARTITION BY clause is useful in situations where you want to perform the same aggregate operation on multiple partitions of data.

BENEFIT of PARTITION over GROUPBY :

WE can use non-aggregated columns also in partition by. Unlike in GROUPBY we can only use columns passed to GROUPBY.

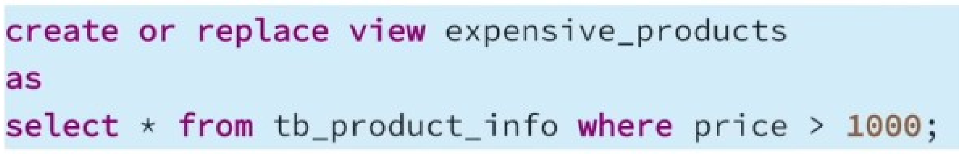
NOTE : 'AS' used for rename the result column name is used after partition

partition on location and assign rank to years of experience based on location



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Highest years of experience from each location



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

marks greater than respective branch's avg marks

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

PARTITION BY on MUltiple columns

cheapest flights between 2 cities

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

SUBQUERIES in SQL - Campusx

https://youtu.be/YYq47MN3TZI

Subquery

In SQL, a subquery is a query within another query. It is a SELECT statement that is nested inside another SELECT, INSERT, UPDATE, or DELETE statement. The subquery is executed first, and its result is then used as a parameter or condition for the outer query.

Combination of 2 conditions :

4. Find the most profitable movie of each year

SELECT * FROM movies WHERE (year,gross - budget) IN (select year, max(gross - budget) as profit from movies GROUP BY year);

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

2nd example query on 2 conditions:

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

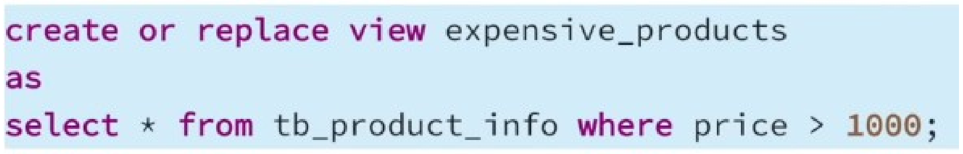
Co-related subquery

https://youtu.be/YYq47MN3TZI?t=4944

Find all the movies that have a rating higher than the average rating of movies in the same genre:

SUB query in SELECT statement

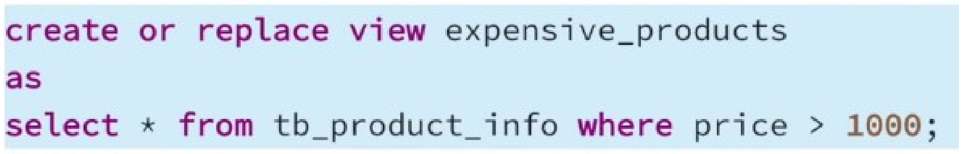
Get the percentage of votes for each movie compared to the total number of votes.



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Not efficient sub-query : SELECT inside SELECT

Display all movie names ,genre, score and avg(score) of genre



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

WINDOW FUNCTIONS

Window functions in SQL are a type of aggregate function that operate over a set of rows, defined by a sliding window or a set of rows. They are used to perform calculations on a subset of rows, rather than on the entire result set of a query.

The window specification is defined using the **OVER() clause in SQL**, which specifies the partitioning and ordering of the rows that the window function will operate on.

Window functions allow you to perform calculations that depend on the values of multiple rows in a query, and can be useful for tasks such as calculating running totals, moving averages, percentiles, and more.

NOTE : It is mandatory to use OVER () while using WINDOWS Function.

How WINDOW Function is different from GROUP BY of pandas

WINDOW Functions:

- Operate within the context of individual rows.

- Calculate values based on a window of related rows around each row.
- Values are added as new columns to the existing rows.
- Suitable for analytical calculations within rows.
- **Retains the same number of rows in the result.**

GROUP BY in pandas:

- Groups rows with the same values in specified columns.
- Aggregates data within each group using functions like sum, mean, etc.
- Reduces the number of rows, representing each group with a single row.
- Used for summarizing data based on column values.
- **Changes the number of rows in the result by collapsing groups.**

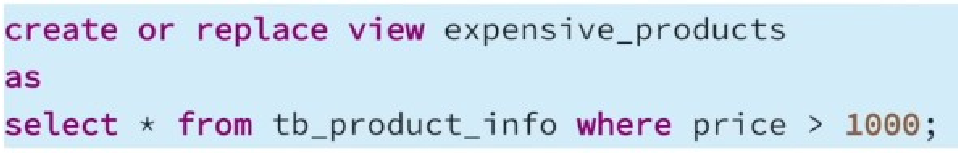
Types of Window functions :

- **ROW_NUMBER:** Assigns a unique number to each row in the result set, based on the order specified in the ORDER BY clause of the OVER clause.
- **RANK:** Assigns a unique rank to each row in the result set, based on the order specified in the ORDER BY clause of the OVER clause. Rows with the same values receive the same rank, and a gap is left in the ranking for the next unique value.
- **DENSE_RANK:** Assigns a unique rank to each row in the result set, based on the order specified in the ORDER BY clause of the OVER clause. Rows with the same values receive the same rank, and there is no gap in the ranking for the next unique value.
- **NTILE:** Divides the result set into a specified number of groups, or tiles, and assigns a number to each row indicating which tile it belongs to.
- **PERCENT_RANK:** Calculates the relative rank of each row within the result set as a fraction between 0 and 1.
- **CUME_DIST:** Calculates the cumulative distribution of a value within the result set, expressed as a fraction between 0 and 1.
- **LEAD and LAG:** Return the value of a specified column from a row at a specified offset from the current row, either ahead (LEAD) or behind (LAG) in the result set.

You can use a window function by including it as part of an aggregate function in a query, and using the OVER clause to specify the window for the function.

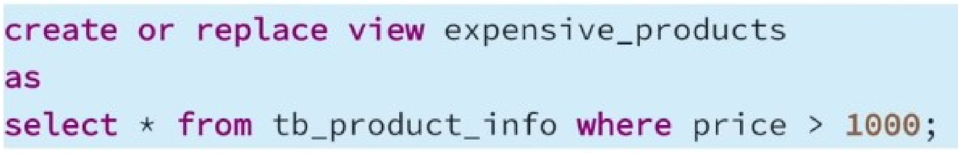
1. ROW NUMBER

NOTE : Row number() is always used with order by



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

5th highest salary



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

2. RANK and DENSE RANK

RANK and DENSE_RANK are both used to assign a unique rank to each row within a result set, based on the values in one or more columns.

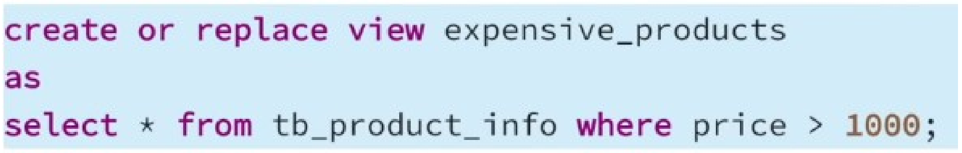
However, there is a difference in the way that they handle ties (rows with equal values in the ranking column).

- RANK assigns the same rank to tied rows and skips the next rank number. For example, if two rows have the same value and are assigned rank 1, the next row would be assigned rank 3.
- DENSE_RANK, on the other hand, assigns the same rank to tied rows and does not skip any rank numbers. For example, if two rows have the same value and are assigned rank 1, the next row would be assigned rank 2.

In summary, RANK can have "gaps" in the rank numbers, while DENSE_RANK always assigns consecutive rank numbers.

NOTE : For RANK and DENSE RANK ORDER BY is mandatory

ROW NUMBER



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

RANK

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

rank branch wise

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

DENSE RANK

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

RANK () vs DENSE_RANK()

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

3. LEAD and LAG

Return the value of a specified column from a row at a specified offset from the current row, either ahead (LEAD) or behind (LAG) in the result set.

LAG : Grouping on source of joining

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

null output is for first record of each group of source_of_joining now passing 2 rows before for each group

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

LEAD : Grouping on source of joining

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

null output is for last record of each group of source_of_joining

now passing 2 rows after for each group

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Month on Month Growth using LAG

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

4. First_value() - first record of years for each group

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Student name who has highest marks overall

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

name and branch and marks of only toppers

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

5. Last_value() - last years for each group

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

NOTE : We are not getting correct results bcoz of FRAME () clause

Default FRAME clause:

over (partition by source_of_joining order by years_of_experience desc **range between unbounded preceding and current row)** as least_experienced

changing current row to unbounded following for last_value to work:

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

another example

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

name and branch of lowest marks

OR

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

FRAME clause()

- FRAME is a subset of partition created by Windows function.

- It defines the scope of the calculation performed by a window function, and it's used to specify which rows should be included in the calculation based on their relative position to the current row performed by a window function.

The ROWS clause specifies how many rows should be included in the frame relative to the current row. For example, ROWS 3 PRECEDING means that the frame includes the current row and the three rows that precede it in the partition.

The BETWEEN clause specifies the boundaries of the frame.

The FRAME clause has two components:

- **ROWS BETWEEN :** Specifies the range of rows to include in the calculation, either UNBOUNDED PRECEDING and UNBOUNDED FOLLOWING, to include all rows, or a range such as 1 PRECEDING and 1 FOLLOWING to include only the current row and its two neighbors.

- **EXCLUSIVE or INCLUSIVE :** Specifies whether the first and last rows in the frame should be included in the calculation or excluded.

Examples

- **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW** - means that the frame includes all rows from the beginning of the partition up to and including the current row.

- **ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING:** the frame includes the current row and the row immediately before and after it.

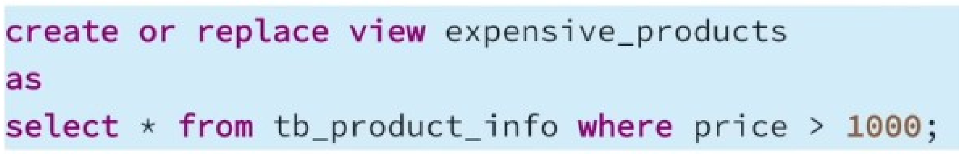
- **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING:** the frame includes all rows in the partition.

- **ROWS BETWEEN 3 PRECEDING AND 2 FOLLOWING:** the frame includes the current row and the three rows before it and the two rows after it.

6. Nth_Value

The NTH_VALUE function in MySQL is a window function that returns the nth value **(any value from a position specified by us)** in a set of values, based on a specified order. The function returns the value of a specified expression for the nth row in the window frame, where the frame is defined using the OVER clause.

2nd most experienced person from each group



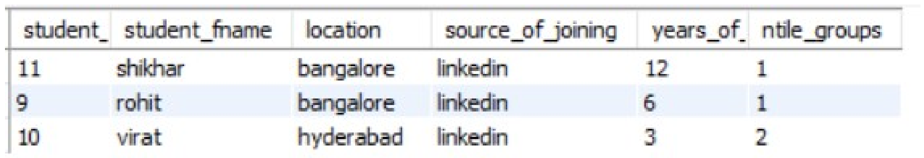
```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

NOTE : if number of rows in any group is less than tha nth_value provided, that group will return NULL

7. Ntile

Segmentation using NTILE is a technique in SQL for dividing a dataset into equal- sized groups based on some criteria or conditions, and then performing calculations or analysis on each group separately using window functions.It returns a number representing the group or tile that each row belongs to.

The NTILE function in SQL is a window function that returns the ntile value for a **set of rows (buckets)** based on a specified order.



student_	student_fname	location	source_of_joining	years_of_	ntile_groups
11	shikhar	bangalore	linkedin	12	1
9	rohit	bangalore	linkedin	6	1
10	virat	hyderabad	linkedin	3	2

NOTE : if total rows is uneven when divided, then 1st quantile group gets more records

Using cases in Ntile

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

8. CUM_DIST() - cummulative distribution

The cumulative distribution function is used to describe the probability distribution of random variables.

It can be used to describe the probability for a discrete, continuous or mixed variable. It is obtained by summing up the probability density function and getting the cumulative probability for a random variable

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

students having marks greater than 90 percentile:

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Another example:

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

round of to 3 places after decimal using round(,3)

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

using it to fetch first 35% from each group

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

9. PERCENT_RANK

percent_rank is a window function in SQL that calculates the **relative rank of a row within a set of rows,** represented as a decimal value between 0 and 1, where 0 represents the lowest rank and 1 represents the highest rank.

The percent_rank function is used in a similar way to the rank function, but instead of returning the rank as an integer, it returns the rank as a decimal value. The percent_rank function is calculated as: **(rank - 1) / (total number of rows - 1).**

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Difference between cum_dist() and percent_rank()

The difference between percent_rank and cume_dist lies **in the way they calculate the relative position of a row within a set of rows.**

- **percent_rank** returns the relative rank of a row as a decimal value between 0 and 1, where 0 represents the lowest rank and 1 represents the highest rank.

The percent_rank function is calculated as (rank - 1) / (total number of rows - 1).

- **cume_dist** returns the cumulative distribution of a value within a set of values, represented as a decimal value between 0 and 1.

The cume_dist function calculates the fraction of rows that are less than or equal to the current row, within the set of rows defined by the PARTITION BY clause.

10. CUMULATIVE SUM

Cumulative sum is another type of calculation that can be performed using window functions. A cumulative sum calculates the sum of a set of values up to a given point in time, and includes all previous values in the calculation.

career runs of viart kohli after 50th, 100th match

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

11. CUMULATIVE AVERAGE

Cumulative average is another type of average that can be calculated using window functions. A cumulative average calculates the average of a set of values up to a given point in time, and includes all previous values in the calculation.

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

12. MOVING AVERAGE

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

13. Percentage of total

Percent of total refers to the percentage or proportion of a specific value in relation to the total value. It is a commonly used metric to represent the relative importance or contribution of a particular value within a larger group or population.

Percentage of total sales an item brings to restaurants

14. Percentage of change

Percent change is a way of expressing the difference between two values as a percentage of the original value. It is often used to measure how much a value has increased or decreased over a given period of time, or to compare two different values.

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

15 . Percentiles & Quantiles

A **Quantile** is a measure of the distribution of a dataset that divides the data into any number of equally sized intervals. For example, a dataset could be divided into **deciles** (ten equal parts), **quartiles** (four equal parts), **percentiles** (100 equal parts), or any other number of intervals.

Each quantile represents a value below which a certain percentage of the data falls. For example, the 25th percentile (also known as the first quartile, or Q1) represents the value below which 25% of the data falls. The 50th percentile (also known as the median) represents the value below which 50% of the data falls, and so on.

EXISTS

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns TRUE if the subquery returns one or more records

In this example, we're using EXISTS in a subquery to check if there are any rows in the orders table that have a customer_id equal to the id of a row in the customers table. If the subquery returns any rows, the EXISTS condition is considered to be true, and the outer query will return all rows from the customers table. If the subquery doesn't return any rows, the EXISTS condition is considered to be false, and the outer query won't return any rows.

ANY and ALL

The ANY operator :

- returns a boolean value as a result
- returns TRUE if ANY of the subquery values meet the condition
- ANY means that the condition will be true if the operation is true for any of the values in the range.

```
SELECT *
FROM employees
WHERE salary > ANY (
  SELECT salary
  FROM employees
  WHERE department = 'Sales'
);
```

The ALL operator:

- returns a boolean value as a result
- returns TRUE if ALL of the subquery values meet the condition
- is used with SELECT, WHERE and HAVING statements
- ALL means that the condition will be true only if the operation is true for all values in the range.

CREATE INDEX

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

CREATE UNIQUE INDEXES

DROP INDEX :

VIEW()

- 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 statements and functions to a view and present the data as if the data were coming from one single table.
- A view is created with the CREATE VIEW statement.

CREATE VIEW Syntax

USES of View:

Here are some common uses of VIEWs in MySQL:

- **Data Abstraction :** You can use a VIEW to abstract complex data structures, making it easier to work with the data. For example, you can create a VIEW that combines data from multiple tables and present it as a single, simplified table.
- **Data Security :** A VIEW can be used to restrict access to sensitive data by limiting the columns or rows that are visible to specific users.
- **Data Consistency :** A VIEW can be used to enforce data consistency by preventing direct updates to the underlying tables and forcing all updates to be made through the VIEW.
- **Simplifying Complex Queries :** You can use a VIEW to simplify complex queries by encapsulating the logic of the query in the VIEW definition. This makes it easier to understand and maintain the query, and also makes it easier to reuse the query in multiple places.

Once a VIEW is created, you can use it just like a regular table in your SELECT, UPDATE, and DELETE statements.

WITH clause() or Common Table Expression (CTE)

https://www.youtube.com/watch?v=QNfnuK-1YYY&list=PLavw5C92dz9Ef4E-1Zi9KfCTXS_IN8gXZ&index=8

The WITH clause in MySQL is also known as a **Common Table Expression (CTE)** and is used to simplify complex SQL queries by creating a temporary, named result set that can be referred to within the main query.

A CTE is essentially a named subquery that can be used within a SELECT, INSERT, UPDATE, or DELETE statement.

we start with WITH clause and then give an alias such cte, then pass the column name required

average_salary here is a temp table

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Q- Apple Stores with sales greater than average sale of combined stores :

Steps :

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

- without using WITH clause(), using subqueries

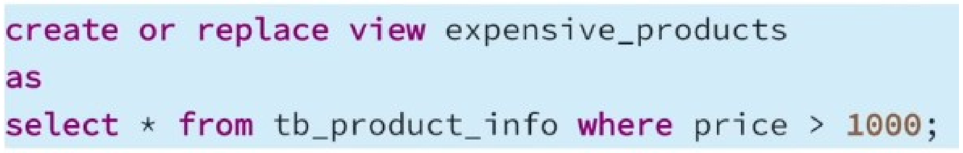
```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

- using WITH clause()

- using with clause() we will get the average salary then we give that as a condition to filter data
- first with clause will be Total_Sales
- Second with clause is Avearge_sales

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```



```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Benefits of using CTE (WITH clause) :

- **Improved readability:** A CTE can make a complex query easier to understand by breaking it down into smaller, named sub-queries that can be referenced within the main query.
- **Reusability:** A CTE can be used multiple times within a single query or across multiple queries, which can improve code reuse and make maintenance easier.
- **Improved performance :** A CTE can be used to break down a complex query into smaller, more manageable parts, which can improve query performance by reducing the amount of data that needs to be processed at each stage.
- **Improved maintainability:** By encapsulating a sub-query within a CTE, the query becomes self-contained, making it easier to understand and maintain, especially for complex queries.
- **Better error handling :** If a query that includes a CTE encounters an error, only the CTE is affected, rather than the entire query, which can simplify debugging and error handling.
- **Improved optimization:** By breaking down a complex query into smaller sub-queries, the optimizer can better evaluate the cost of each part, leading to more efficient query execution.

In summary, the use of the WITH clause provides several benefits that can lead to improved performance, readability, and maintainability of SQL queries.

VIEW ()

https://www.youtube.com/watch?v=cLSxasHg9WY&list=PLavw5C92dz9Ef4E-1Zi9KfCTXS_IN8gXZ&index=11

- View is a database object created over a SQL query.
- Doesnot stores any data but stores structure of the query
- Just executed the SQL query underlying it.
- It is like a virtual table
- only stores data not the query. internally the query will be executed
- doesnot capture the changes in the database automatically.

view over a select statement :

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

calling the view table :

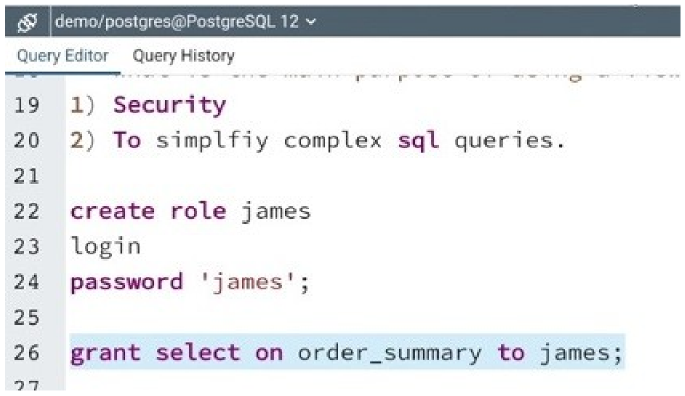
select * from order_summary;

Advantages of using view:

- **Simplify complex queries :** Views break down complex queries into smaller pieces.
- **Improve data security :** Views restrict access to sensitive data by only exposing a subset of columns.
- **Reuse and maintain code :** Views encapsulate complex calculations and logic for reuse and easy maintenance.
- **Improve performance :** Views simplify complex queries and make them more efficient.
- **Abstract data :** Views provide a simplified and unified view of data.

NOTE : VIEW is not faster than a normal query

USE of view : Creating new user for privacy

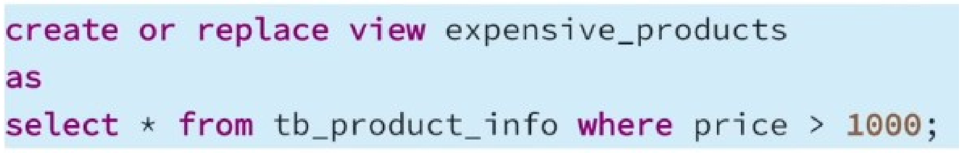


```
1) Security
2) To simplfiy complex sql queries.

create role james
login
password 'james';

grant select on order_summary to james;
```

```
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```



```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

NOTE : client will not be able to see the query behind the view

CREATE OR REPLACE in view:

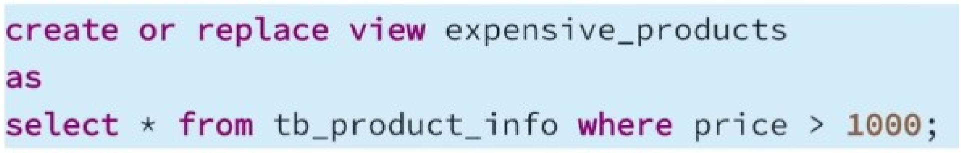
- Name and Datatype of columns cannot be changed
- Order of column cannot be changed
- New columns can only be added at the end

ALTER VIEW in view:

- Columns can be renamed using ALTER

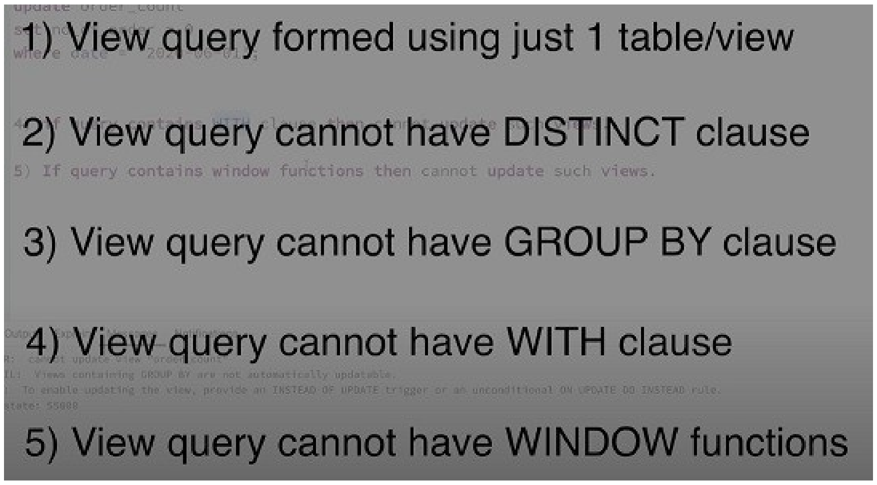
NOTE: FOR VIEW :

If we add new data to the table then view will show it automatically but we change the structure of the table then view wont show it and we need to recreate or refresh the view.



```sql
create or replace view expensive_products
as
select * from tb_product_info where price > 1000;
```

Updatable views should follow the below rules :



1) View query formed using just 1 table/view

2) View query cannot have DISTINCT clause

3) View query cannot have GROUP BY clause

4) View query cannot have WITH clause

5) View query cannot have WINDOW functions

MATERIALIZED VIEW

https://www.youtube.com/watch?v=WzkBZ0byoYE&list=PLavw5C92dz9Ef4E-1Zi9KfCTXS_IN8gXZ&index=17

Mysql doesnot have materialized view

A Materialized View is a pre-computed table that is stored on disk, which can be used to improve query performance by reducing the amount of data that needs to be processed at runtime.

- The results will be same as using a select statement but faster.
- The data in materialized view will not automatically update if table gets updated. Can only be updated using refresh

eg : refresh materialized view mv_random_tab;

- Donot create materialized view for each and every data

Benefits of using Materialized View:

- **Improved query performance :** Materialized Views can significantly improve query performance by reducing the amount of data that needs to be processed at runtime. This is because the data in a Materialized View is pre-computed and stored on disk, so the database can retrieve it quickly without having to perform complex calculations at runtime.
- **Reduced data redundancy:** Materialized Views can help reduce data redundancy by storing commonly used data in a centralized location, so it can be easily referenced by multiple queries.
- **Improved data consistency:** Materialized Views can be used to enforce data consistency by preventing updates to the underlying data until the Materialized View has been refreshed.
- **Improved scalability:** Materialized Views can be used to distribute data across multiple nodes in a distributed database, which can help improve scalability and performance in high-traffic environments.
- **Improved resource utilization :** Materialized Views can help improve resource utilization by reducing the amount of processing required at runtime, which can help reduce CPU, memory, and disk I/O utilization.
- **Simplified data management:** Materialized Views can simplify data management by reducing the complexity of the underlying data, making it easier to understand, maintain, and update.

In summary, Materialized Views are a powerful tool that can significantly improve query performance, reduce data redundancy, enforce data consistency, improve scalability, and simplify data management in SQL databases.

Difference between view and materialized view()

Materialized view stores the data and the query but view only stores the datanot the query .

- A View is a virtual table that does not store data but contains a SELECT statement that is executed each time the View is referenced in a query.
- A Materialized View is a pre-computed table that stores data on disk and can be used to improve query performance.
- Views always reflect the most up-to-date data, but have no performance benefits.
- Materialized Views need to be refreshed and use more disk space, but can significantly improve query performance.

PROCEDURES()

https://www.youtube.com/watch?v=yLR1w4tZ36I&list=PLavw5C92dz9Ef4E-1Zi9KfCTXS_IN8gXZ&index=14

TRIGGERS

Triggers are a feature in MySQL that allow you to execute a set of actions automatically in response to specific events, such as the insertion, update, or deletion of data in a table. Triggers can be used to enforce business rules, maintain data integrity, and perform actions that are related to the change in data, such as logging changes or updating other tables.

A trigger is defined using the CREATE TRIGGER statement and consists of two parts: the trigger definition and the trigger body. The trigger definition specifies the event that activates the trigger, such as an insert into a specific table, and the trigger body contains the SQL statements that are executed when the trigger is activated.

In this example, the trigger is named update_last_update, is activated AFTER an update to the mytable table, and FOR EACH ROW updates the value of the last_update column to the current time.

Triggers are a powerful tool for automating tasks and enforcing data constraints, but they can also be complex and difficult to debug, so it's important to use them carefully and test them thoroughly.

SEEK and SCAN in sql

"Seek" and "Scan" are two methods used by the MySQL database management system to **search for data in a table.**

- **Seek** is a direct lookup method, where MySQL uses the index of a table to quickly find a specific row of data based on its unique key. This method is fast and efficient, but it can only be used when searching for an exact match of a unique key value.
- **Scan** on the other hand, is a method where MySQL scans the entire table to find the rows that match a certain condition. This method is slower than "Seek", but it can be used to find all rows that match a certain condition, even if no index exists for the columns being searched. Scans can also be used to return all rows in a table if no specific search condition is provided.

In summary, "Seek" is a fast, direct lookup method for finding a specific row in a table, while "Scan" is a slower method for finding all rows that match a certain condition.

USER-DEFINED VARIABLE

In MySQL, the @ symbol is used as a user-defined variable. It allows you to store a value or expression and reuse it multiple times within a single query or across multiple queries.

For example, you can use the following syntax to define a user-defined variable in MySQL:

COALESCE ()

The COALESCE function in SQL is used to return the first non-NULL value from a list of expressions. It takes a list of one or more expressions as its arguments, and returns the first expression that is not NULL. If all expressions are NULL, then COALESCE returns NULL.

In this example, the COALESCE function will return the value in column1 if it's not NULL, and if column1 is NULL, it will return the value in column2, and so on. If all columns are NULL, then the COALESCE function will return NULL.

The COALESCE function is often used to provide a default value for a column when it's NULL, for example:

In this case, if the value in column1 is NULL, the COALESCE function will return 0 instead.

POINTS TO REMEMBER

- Whenever you pass id column as PRIMARY KEY use AUTO_INCREMENT or set some default value as it cannot have NULL value.
- for deafult timestamp as time right now use TIMESTAMP DEFAULT NOW()
- The columns on which GROUP BY is applied should be present in SELECT option

MySQL String Functions

- **ASCII** ⇒ Returns the ASCII value for the specific character
- **CHAR_LENGTH** ⇒ Returns the length of a string (in characters)
- **CHARACTER_LENGTH** ⇒ Returns the length of a string (in characters)
- **CONCAT** ⇒ Adds two or more expressions together
- **CONCAT_WS** ⇒ Adds two or more expressions together with a separator
- **FIELD** ⇒ Returns the index position of a value in a list of values
- **FIND_IN_SET** ⇒ Returns the position of a string within a list of strings
- **FORMAT** ⇒ Formats a number to a format like "#,###,###.##", rounded to a specified number of decimal places
- **INSERT** ⇒ Inserts a string within a string at the specified position and for a certain number of characters
- **INSTR** ⇒ Returns the position of the first occurrence of a string in another string
- **LCASE** ⇒ Converts a string to lower-case
- **LEFT** ⇒ Extracts a number of characters from a string (starting from left)
- **LENGTH** ⇒ Returns the length of a string (in bytes)
- **LOCATE** ⇒ Returns the position of the first occurrence of a substring in a string
- **LOWER** ⇒ Converts a string to lower-case
- **LPAD** ⇒ Left-pads a string with another string, to a certain length
- **LTRIM** ⇒ Removes leading spaces from a string
- **MID** ⇒ Extracts a substring from a string (starting at any position)
- **POSITION** ⇒ Returns the position of the first occurrence of a substring in a string
- **REPEAT** ⇒ Repeats a string as many times as specified
- **REPLACE** ⇒ Replaces all occurrences of a substring within a string, with a new substring
- **REVERSE** ⇒ Reverses a string and returns the result

- **RIGHT** ⇒ Extracts a number of characters from a string (starting from right)
- **RPAD** ⇒ Right-pads a string with another string, to a certain length
- **RTRIM** ⇒ Removes trailing spaces from a string
- **SPACE** ⇒ Returns a string of the specified number of space characters
- **STRCMP** ⇒ Compares two strings
- **SUBSTR** ⇒ Extracts a substring from a string (starting at any position)
- **SUBSTRING** ⇒ Extracts a substring from a string (starting at any position)
- **SUBSTRING_INDEX** ⇒ Returns a substring of a string before a specified number of delimiter occurs
- **TRIM** ⇒ Removes leading and trailing spaces from a string
- **UCASE** ⇒ Converts a string to upper-case
- **UPPER** ⇒ Converts a string to upper-case

MySQL Numeric Functions

- **ABS** ⇒ Returns the absolute value of a number
- **ACOS** ⇒ Returns the arc cosine of a number
- **ASIN** ⇒ Returns the arc sine of a number
- **ATAN** ⇒ Returns the arc tangent of one or two numbers
- **ATAN2** ⇒ Returns the arc tangent of two numbers
- **AVG** ⇒ Returns the average value of an expression
- **CEIL** ⇒ Returns the smallest integer value that is >= to a number
- **CEILING** ⇒ Returns the smallest integer value that is >= to a number
- **COS** ⇒ Returns the cosine of a number
- **COT** ⇒ Returns the cotangent of a number
- **COUNT** ⇒ Returns the number of records returned by a select query
- **DEGREES** ⇒ Converts a value in radians to degrees
- **DIV** ⇒ Used for integer division
- **EXP** ⇒ Returns e raised to the power of a specified number
- **FLOOR** ⇒ Returns the largest integer value that is <= to a number
- **GREATEST** ⇒ Returns the greatest value of the list of arguments
- **LEAST** ⇒ Returns the smallest value of the list of arguments
- **LN** ⇒ Returns the natural logarithm of a number
- **LOG** ⇒ Returns the natural logarithm of a number, or the logarithm of a number to a specified base
- **LOG10** ⇒ Returns the natural logarithm of a number to base 10
- **LOG2** ⇒ Returns the natural logarithm of a number to base 2
- **MAX** ⇒ Returns the maximum value in a set of values

- **MIN** ⇒ Returns the minimum value in a set of values
- **MOD** ⇒ Returns the remainder of a number divided by another number
- **PI** ⇒ Returns the value of PI
- **POW** ⇒ Returns the value of a number raised to the power of another number
- **POWER** ⇒ Returns the value of a number raised to the power of another number
- **RADIANS** ⇒ Converts a degree value into radians
- **RAND** ⇒ Returns a random number
- **ROUND** ⇒ Rounds a number to a specified number of decimal places
- **SIGN** ⇒ Returns the sign of a number
- **SIN** ⇒ Returns the sine of a number
- **SQRT** ⇒ Returns the square root of a number
- **SUM** ⇒ Calculates the sum of a set of values
- **TAN** ⇒ Returns the tangent of a number
- **TRUNCATE** ⇒ Truncates a number to the specified number of decimal places

MySQL Advanced Functions

- **BIN** ⇒ Returns a binary representation of a number
- **BINARY** ⇒ Converts a value to a binary string
- **CASE** ⇒ Goes through conditions and return a value when the first condition is met
- **CAST** ⇒ Converts a value (of any type) into a specified datatype
- **COALESCE** ⇒ Returns the first non-null value in a list
- **CONNECTION_ID** ⇒ Returns the unique connection ID for the current connection
- **CONV** ⇒ Converts a number from one numeric base system to another
- **CONVERT** ⇒ Converts a value into the specified datatype or character set
- **CURRENT_USER** ⇒ Returns the user name and host name for the MySQL account that the server used to authenticate the current client
- **DATABASE** ⇒ Returns the name of the current database
- **IF** ⇒ Returns a value if a condition is TRUE, or another value if a condition is FALSE
- **IFNULL** ⇒ Return a specified value if the expression is NULL, otherwise return the expression
- **ISNULL** ⇒ Returns 1 or 0 depending on whether an expression is NULL
- **LAST_INSERT_ID** ⇒ Returns the AUTO_INCREMENT id of the last row that has been inserted or updated in a table
- **NULLIF** ⇒ Compares two expressions and returns NULL if they are equal. Otherwise, the first expression is returned
- **SESSION_USER** ⇒ Returns the current MySQL user name and host name
- **SYSTEM_USER** ⇒ Returns the current MySQL user name and host name
- **USER** ⇒ Returns the current MySQL user name and host name
- **VERSION** ⇒ Returns the current version of the MySQL database

MySQL Date Functions

- **ADDDATE** ⇒ Adds a time/date interval to a date and then returns the date
- **ADDTIME** ⇒ Adds a time interval to a time/datetime and then returns the time/datetime

- **CURDATE** ⇒ Returns the current date
- **CURRENT_DATE** ⇒ Returns the current date
- **CURRENT_TIME** ⇒ Returns the current time
- **CURRENT_TIMESTAMP** ⇒ Returns the current date and time
- **CURTIME** ⇒ Returns the current time
- **DATE** ⇒ Extracts the date part from a datetime expression
- **DATEDIFF** ⇒ Returns the number of days between two date values
- **DATE_ADD** ⇒ Adds a time/date interval to a date and then returns the date
- **DATE_FORMAT** ⇒ Formats a date
- **DATE_SUB** ⇒ Subtracts a time/date interval from a date and then returns the date
- **DAY** ⇒ Returns the day of the month for a given date
- **DAYNAME** ⇒ Returns the weekday name for a given date
- **DAYOFMONTH** ⇒ Returns the day of the month for a given date
- **DAYOFWEEK** ⇒ Returns the weekday index for a given date
- **DAYOFYEAR** ⇒ Returns the day of the year for a given date
- **EXTRACT** ⇒ Extracts a part from a given date
- **FROM_DAYS** ⇒ Returns a date from a numeric datevalue
- **HOUR** ⇒ Returns the hour part for a given date
- **LAST_DAY** ⇒ Extracts the last day of the month for a given date
- **LOCALTIME** ⇒ Returns the current date and time
- **LOCALTIMESTAMP** ⇒ Returns the current date and time
- **MAKEDATE** ⇒ Creates and returns a date based on a year and a number of days value
- **MAKETIME** ⇒ Creates and returns a time based on an hour, minute, and second value
- **MICROSECOND** ⇒ Returns the microsecond part of a time/datetime
- **MINUTE** ⇒ Returns the minute part of a time/datetime
- **MONTH** ⇒ Returns the month part for a given date
- **MONTHNAME** ⇒ Returns the name of the month for a given date
- **NOW** ⇒ Returns the current date and time
- **PERIOD_ADD** ⇒ Adds a specified number of months to a period
- **PERIOD_DIFF** ⇒ Returns the difference between two periods
- **QUARTER** ⇒ Returns the quarter of the year for a given date value
- **SECOND** ⇒ Returns the seconds part of a time/datetime
- **SEC_TO_TIME** ⇒ Returns a time value based on the specified seconds
- **STR_TO_DATE** ⇒ Returns a date based on a string and a format
- **SUBDATE** ⇒ Subtracts a time/date interval from a date and then returns the date
- **SUBTIME** ⇒ Subtracts a time interval from a datetime and then returns the time/datetime
- **SYSDATE** ⇒ Returns the current date and time
- **TIME** ⇒ Extracts the time part from a given time/datetime
- **TIME_FORMAT** ⇒ Formats a time by a specified format
- **TIME_TO_SEC** ⇒ Converts a time value into seconds
- **TIMEDIFF** ⇒ Returns the difference between two time/datetime expressions
- **TIMESTAMP** ⇒ Returns a datetime value based on a date or datetime value

- **TO_DAYS** ⇒ Returns the number of days between a date and date "0000-00-00"
- **WEEK** ⇒ Returns the week number for a given date
- **WEEKDAY** ⇒ Returns the weekday number for a given date
- **WEEKOFYEAR** ⇒ Returns the week number for a given date
- **YEAR** ⇒ Returns the year part for a given date
- **YEARWEEK** ⇒ Returns the year and week number for a given date