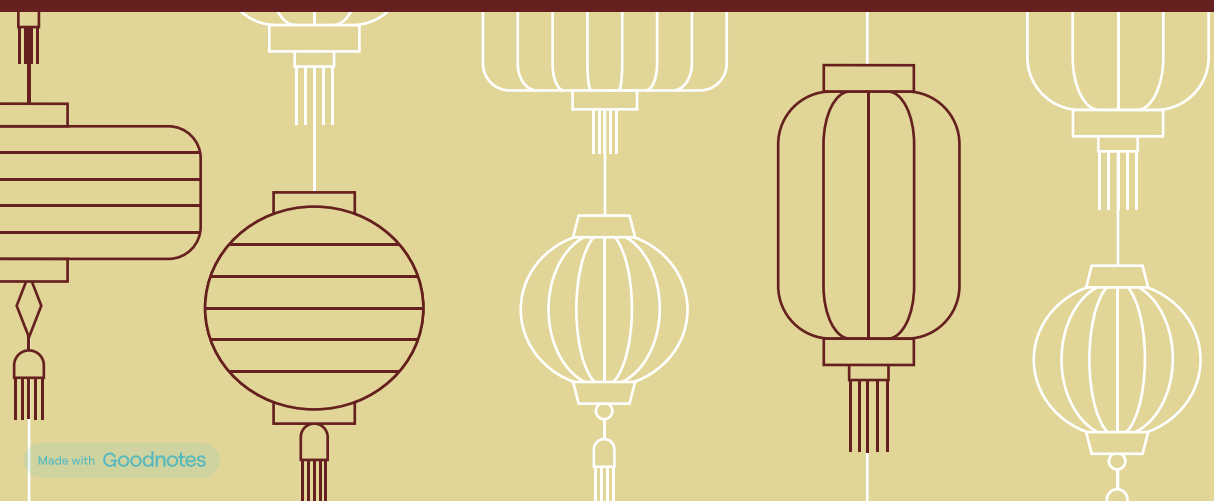


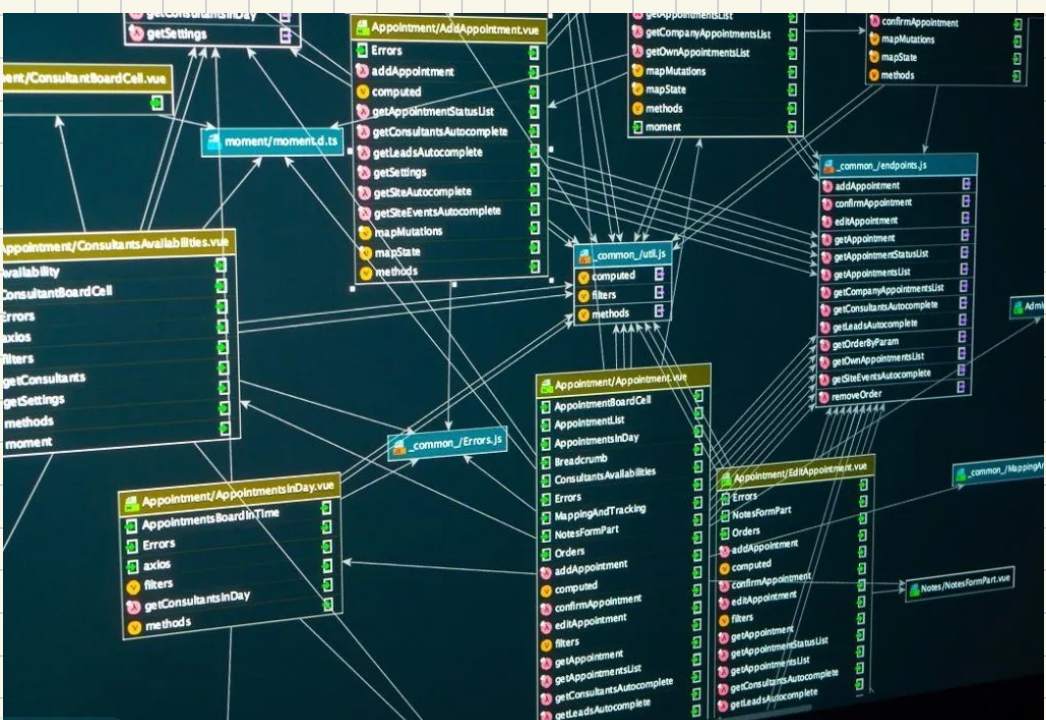
Mass coders - SQL

database:

A database is an organized collection of data that is stored and accessed electronically. The data is structured in a way that allows easy retrieval, management, and updating. Databases are essential for handling large amounts of information in a structured and efficient manner.

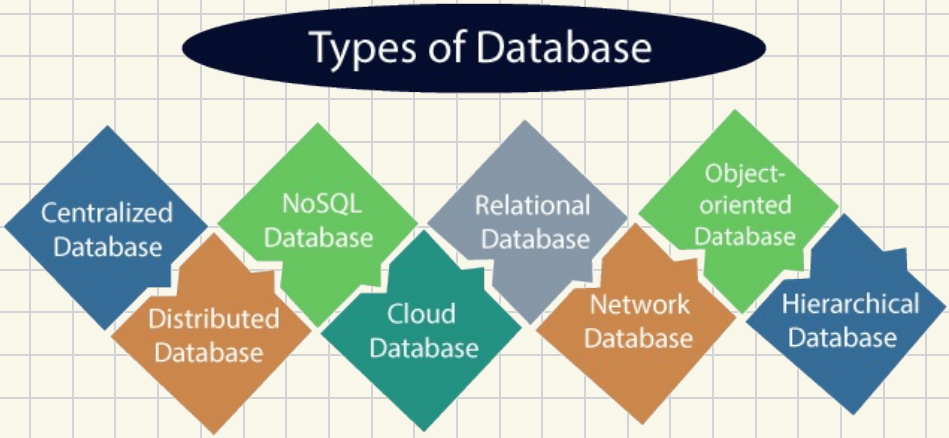
database management system:

A Database Management System (DBMS) is software that allows users to define, create, maintain, and control access to databases. It acts as an intermediary between the user and the database, ensuring that the data is stored efficiently, retrieved quickly, and managed securely.



Types of Databases:

Databases comes in various types, each designed to handle different kinds of data and use cases.



Relational

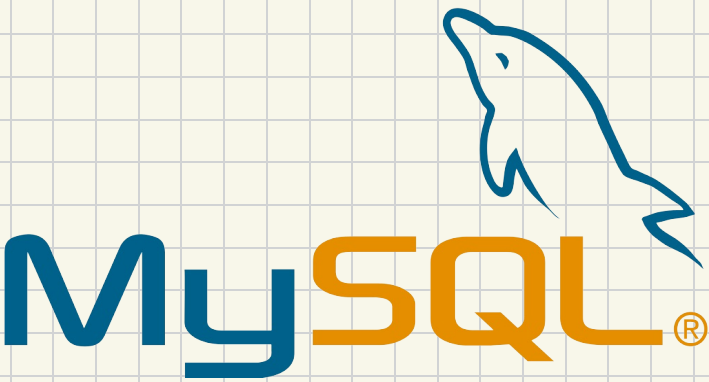
Stored in Tables form

Non-relational (NOSQL)

Not stored in tables form

Relational → MySQL

Non relational → MongoDB



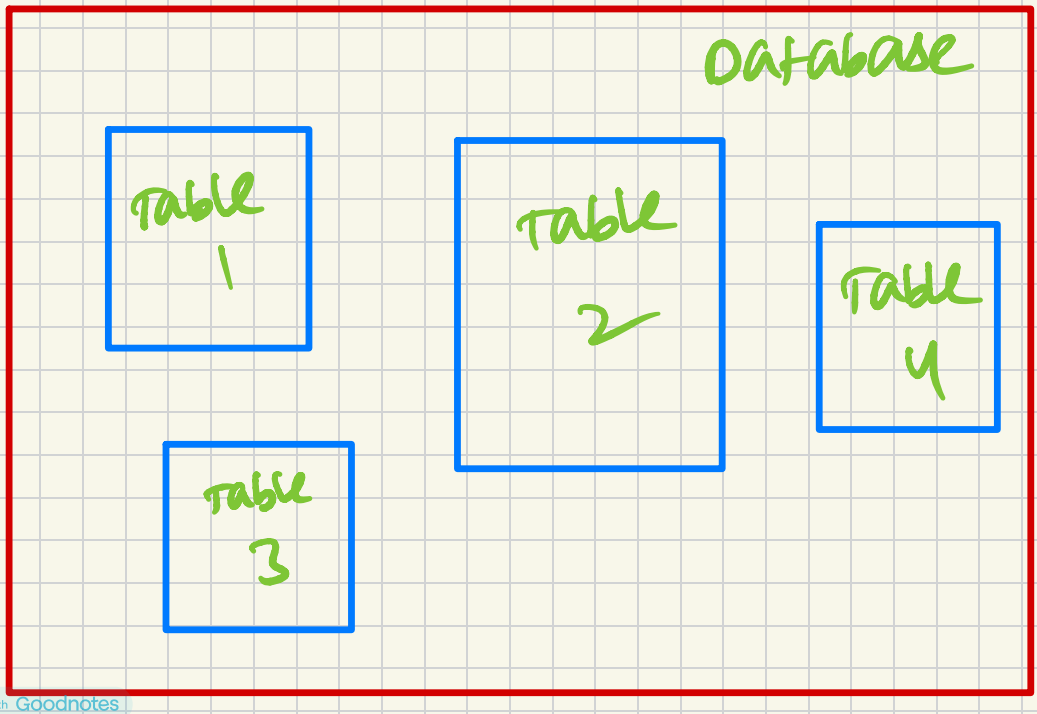
Relational databases tho deal cheyyadaniki SQL programming language vaadtam

SQL → Structured Query Language

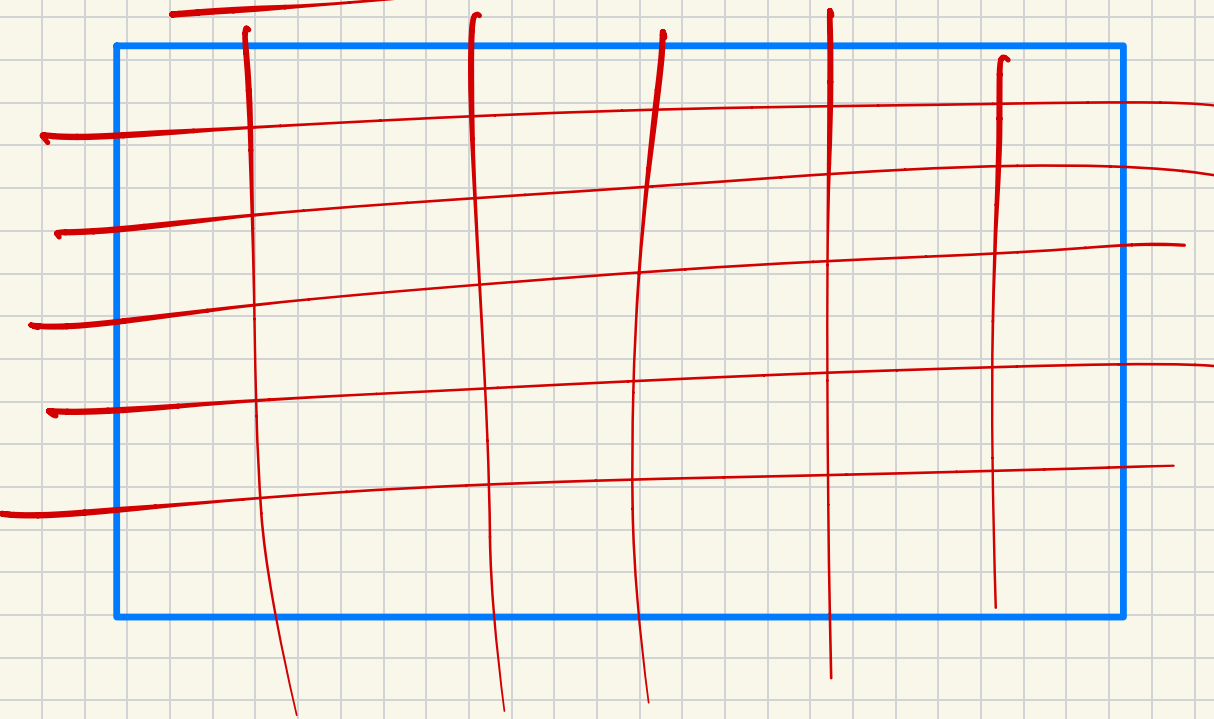
CRUD → Delete

- create
- read
- update

→ Data is stored in tables form



→ structure of a table:



Rows & columns

→ Example of table:

MovieID	MovieName	DirectorName	HeroName	HeroineName	MusicDirector	ReleaseYear
1	RRR	S. S. Rajamouli	N. T. Rama Rao Jr.	Alia Bhatt	M. M. Keeravani	2022
2	Pushpa: The Rise	Sukumar	Allu Arjun	Rashmika Mandanna	Devi Sri Prasad	2021
3	Radhe Shyam	Radha Krishna Kumar	Prabhas	Pooja Hegde	Justin Prabhakaran	2022
4	Sarkaru Vaari Paata	Parasuram	Mahesh Babu	Keerthy Suresh	Thaman S	2022
5	Acharya	Koratala Siva	Chiranjeevi	Kajal Aggarwal	Mani Sharma	2022

TFI

Baagundali :)

SQL commands

DDL

Data Definition Language

- create
- Drop
- Alter
- Truncate
- Rename

DQL

Data Query Language

- select

DML

Data manipulation Language

- Insert
- Update
- Delete
- lock

DCL

Data Control Language

- grant
- revoke

TCL

Transaction Control Language

- begin transaction
- commit
- roll back
- save point

SQL datatypes

Type of values that can be stored inside a column

Data Type	Description	Example Usage
INT	A standard integer type, typically 4 bytes in size.	`age INT`
BIGINT	A larger integer type, typically 8 bytes in size, used for large numerical values.	`total_sales BIGINT`
FLOAT	A floating-point number with precision, typically 4 bytes.	`price FLOAT`
DOUBLE	A double-precision floating-point number, typically 8 bytes.	`rating DOUBLE`
DECIMAL(p, s)	An exact numeric value with precision (`p`) and scale (`s`), often used for financial data.	`salary DECIMAL(10, 2)`
CHAR(n)	A fixed-length character string with a specified length (`n`).	`gender CHAR(1)`
VARCHAR(n)	A variable-length character string with a specified maximum length (`n`).	`email VARCHAR(255)`
TEXT	A variable-length string of unlimited length, often used for large blocks of text.	`description TEXT`
DATE	Stores date values (year, month, day) in the format `YYYY-MM-DD`.	`birth_date DATE`
TIME	Stores time values (hour, minute, second) in the format `HH:MM:SS`.	`start_time TIME`
DATETIME	Stores date and time values in the format `YYYY-MM-DD HH:MM:SS`.	`created_at DATETIME`
TIMESTAMP	Stores date and time values, typically used for tracking changes, often with time zone awareness.	`updated_at TIMESTAMP`
BOOLEAN	Stores boolean values, typically `TRUE` or `FALSE` (may be stored as `1` or `0` depending on DB).	`is_active BOOLEAN`
BLOB	A Binary Large Object, used to store binary data like images or files.	`profile_picture BLOB`

creating database:

query →

```
CREATE DATABASE db_name;
```

Deleting database:

query →

```
DROP DATABASE db_name;
```

Use database:

query →

```
USE db_name;
```

Creating Table:

```
CREATE TABLE table_name (
    column1   datatype1
    column2   datatype2
    column3   datatype3
    .....     .....
);
```

cinema chooptha mowa :)

MOVIES DATABASE

1. Movies

- `movie_id` (INT, Primary Key, Auto-increment)
- `title` (VARCHAR(255))
- `release_year` (YEAR)
- `genre` (VARCHAR(100))
- `language` (VARCHAR(50), Default: 'Telugu')
- `duration_minutes` (INT)
- `rating` (DECIMAL(3, 1))
- `director_id` (INT, Foreign Key)

2. Directors

- `director_id` (INT, Primary Key, Auto-increment)
- `name` (VARCHAR(255))
- `dob` (DATE)
- `nationality` (VARCHAR(100))
- `awards` (TEXT)

3. Actors

- `actor_id` (INT, Primary Key, Auto-increment)
- `name` (VARCHAR(255))
- `dob` (DATE)
- `gender` (CHAR(1))
- `nationality` (VARCHAR(100))
- `debut_year` (YEAR)

4. Movie_Cast

- `movie_id` (INT, Foreign Key)
- `actor_id` (INT, Foreign Key)
- `role_name` (VARCHAR(255))
- `screen_time_minutes` (INT)

5. Box_Office

- `movie_id` (INT, Foreign Key)
- `budget` (BIGINT)
- `box_office_collection` (BIGINT)
- `domestic_collection` (BIGINT)
- `international_collection` (BIGINT)

Queries for creating tables:

```sql
-- Table: Movies
CREATE TABLE Movies (
    movie_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    release_year YEAR NOT NULL,
    genre VARCHAR(100) NOT NULL,
    language VARCHAR(50) DEFAULT 'Telugu',
    duration_minutes INT NOT NULL,
    rating DECIMAL(3, 1),
    director_id INT,
    FOREIGN KEY (director_id) REFERENCES Directors(director_id)
);

-- Table: Directors
CREATE TABLE Directors (
    director_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    dob DATE,
    nationality VARCHAR(100),
    awards TEXT
);

-- Table: Actors
CREATE TABLE Actors (
    actor_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    dob DATE,
    gender CHAR(1),
    nationality VARCHAR(100),
    debut_year YEAR
);
```

Queries for creating tables:

```sql
-- Table: Movie_Cast
CREATE TABLE Movie_Cast (
    movie_id INT,
    actor_id INT,
    role_name VARCHAR(255),
    screen_time_minutes INT,
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    FOREIGN KEY (actor_id) REFERENCES Actors(actor_id),
    PRIMARY KEY (movie_id, actor_id)
);

-- Table: Box_Office
CREATE TABLE Box_Office (
    movie_id INT,
    budget BIGINT,
    box_office_collection BIGINT,
    domestic_collection BIGINT,
    international_collection BIGINT,
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    PRIMARY KEY (movie_id)
);
```

keys

→ primary key:

* It is a unique identifier for each record in a database table.

→ foreign key:

* It is a field in a one table, that refers to the primary key in another table.

NOTE!

→ unique primary key & NOT null

→ foreign keys can be multiple and null

consider movies table →

1. Movies

- `movie_id` (INT, Primary Key, Auto-increment)
- `title` (VARCHAR(255))
- `release_year` (YEAR)
- `genre` (VARCHAR(100))
- `language` (VARCHAR(50), Default: 'Telugu')
- `duration_minutes` (INT)
- `rating` (DECIMAL(3, 1))
- `director_id` (INT, Foreign Key)

primary key

foreign key

Insertion Syntax:

→ Inserting single value:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

→ Inserting multiple values!

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES
    (value1, value2, value3, ...),
    (value4, value5, value6, ...),
    ... ;
```

Inserting values:

→ Inserting a single value:

```
INSERT INTO Movies (title, release_year, genre, language,
duration_minutes, rating, director_id)
VALUES ('Kalki 2898 AD', 2024, 'Action, Fantasy', 'Telugu', 150,
8.5, 1);
```

→ Inserting Multiple values:

```
INSERT INTO Movies (title, release_year, genre, language,
duration_minutes, rating, director_id)
VALUES
('Kalki 2898 AD', 2024, 'Action, Fantasy', 'Telugu', 150, 8.5, 1),
('Pushpa: The Rise', 2021, 'Action, Drama', 'Telugu', 179, 8.0, 2),
('RRR', 2022, 'Action, Drama', 'Telugu', 187, 8.7, 3),
('KGF Chapter 1', 2018, 'Action, Drama', 'Kannada', 156, 8.2, 4),
('KGF Chapter 2', 2022, 'Action, Drama', 'Kannada', 168, 8.4, 4),
('Bahubali: The Beginning', 2015, 'Action, Drama', 'Telugu', 159,
8.1, 5),
('Bahubali: The Conclusion', 2017, 'Action, Drama', 'Telugu', 167,
8.2, 5);
```

Tables ni choodadam ela?

SELECT:

SELECT statement in SQL is used to retrieve data from one or more tables in a database.

Basic Syntax:

```
SELECT column1, column2, ----
FROM table_name;
```

Selecting all columns:

```
SELECT * FROM table_name;
```

→ selecting specific columns:

```
SELECT title, release_year,
rating FROM movies;
```

→ selecting all columns:

```
SELECT * FROM movies;
```

Lets solve this →

→ Create a table named books with the following columns

book_id, title, author, genre, price

published_year, in_stock

→ Insert few random values

→ Display title, published_year columns

CONSTRAINTS

Specify rules for data in a table.

* **NOT NULL:**
ensures that a column cannot contain NULL values

col_name datatype NOT NULL

```
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    salary DECIMAL(10, 2)
);
```

* primary key:

```
CREATE TABLE Movies (
    movie_id INT PRIMARY KEY,
    title VARCHAR(255),
    release_year YEAR
);
```

* foreign key:

```
CREATE TABLE Movies (
    movie_id INT PRIMARY KEY,
    title VARCHAR(255),
    director_id INT,
    FOREIGN KEY (director_id)
REFERENCES Directors(director_id)
);
```

* UNIQUE:

ensures that all the values in a column (or a combination of columns) are unique across the rows of the table

```
CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    username VARCHAR(100) UNIQUE
);
```

*DEFAULT:

provides a default value for a column when no value is specified during insertion.

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    order_date DATE DEFAULT CURRENT_DATE
);
```

*CHECK:

ensures that all values in a column satisfy a specific condition. It is used to limit the range of values that can be inserted into a column.

```
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    price DECIMAL(10, 2),
    quantity INT CHECK (quantity >= 0)
);
```

→ WHERE:

It is used to filter records in a query based on specific conditions.

Syntax:

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

Basic example:

```
SELECT title, release_year FROM Movies
WHERE release_year = 2022;
```

→ Using operators with WHERE:

Arithmetic operators: +, -, *, /, %

Comparison operators: =, !=, >, >=, <, <=

Logical operators: AND, OR, NOT, IN, BETWEEN, ALL, LIKE, ANY

Bitwise operators: &, |

Lets see few examples.....

→ Arithmetic operators example:

profit or loss of a movie

= box office collection − budget

```
SELECT movie_id, box_office_collection - budget AS profit_or_loss FROM Box_Office;
```

Doubling a movie budget ⇒

Doubled Budget = budget * 2

```
SELECT
    movie_id,
    budget * 2 AS doubled_budget
FROM
    Box_Office;
```

→ comparison operators example:

(=)

```
SELECT title, release_year
FROM movies
WHERE release_year = 2022;
```

(!=)

```
SELECT title, language
FROM movies
WHERE language != 'Telugu';
```

(>)

```
SELECT title, duration_minutes
FROM movies
WHERE duration_minutes > 150;
```

(<)

```
SELECT title, rating
FROM movies
WHERE rating < 5.0;
```

→ logical operators example:

(AND)

```
SELECT title, release_year, language
FROM movies
WHERE language = 'Telugu' AND
release_year = 2022;
```

(OR)

```
SELECT title, release_year, language
FROM movies
WHERE language = 'Telugu' OR
release_year = 2022;
```

(NOT)

```
SELECT title, language
FROM movies
WHERE NOT language = 'Telugu';
```

(IN)

```
SELECT title, language
FROM movies
WHERE language IN ('Telugu', 'Hindi');
```

(BETWEEN)

```
SELECT title, release_year
FROM movies
WHERE release_year BETWEEN 2015
                   AND 2022;
```

(LIKE)

```
SELECT title
FROM movies
WHERE title LIKE 'Bahu%';
```

→ Note:

- '%' represents zero or more characters
- '_' represents a single character

→ ORDER BY:

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

Example:

```
SELECT title, release_year
FROM movies ORDER BY release_year DESC;
```

→ LIMIT:

```
SELECT column1, column2, ----
FROM table_name LIMIT number_of_rows;
```

Example:

```
SELECT title, release_year FROM
movies LIMIT 5;
```

→ Aggregate functions :

(COUNT)

```
SELECT COUNT(*) FROM MOVIES;
```

(MAX)

```
SELECT MAX(budget) FROM BOX_OFFICE;
```

(MIN)

```
SELECT MIN(rating) FROM MOVIES;
```

(SUM)

```
SELECT SUM(box_office_collection)
    FROM BOX_OFFICE;
```

(AVG)

```
SELECT AVG(rating) FROM MOVIES;
```

→ GROUP BY:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Example:

```
SELECT language, SUM(box_office_collection)
AS total_collection FROM movies m
JOIN BOX_office b ON
m.movie_id = b.movie_id GROUP BY language;
```

The GROUP BY clause groups rows that have the same values in specified columns into summary rows, like "total", "average", "count", etc. It is often used with aggregate functions.

returns total box office collection for each language

→ HAVING:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

Example:

```
SELECT language, SUM(box_office_collection)
AS total_collection FROM movies m
JOIN box_office b ON
m.movie_id = b.movie_id GROUP BY language
HAVING SUM(box_office_collection) > 500000000;
```

The HAVING clause is used to filter groups created by the GROUP BY clause. It is similar to the WHERE clause but is used with aggregate functions.

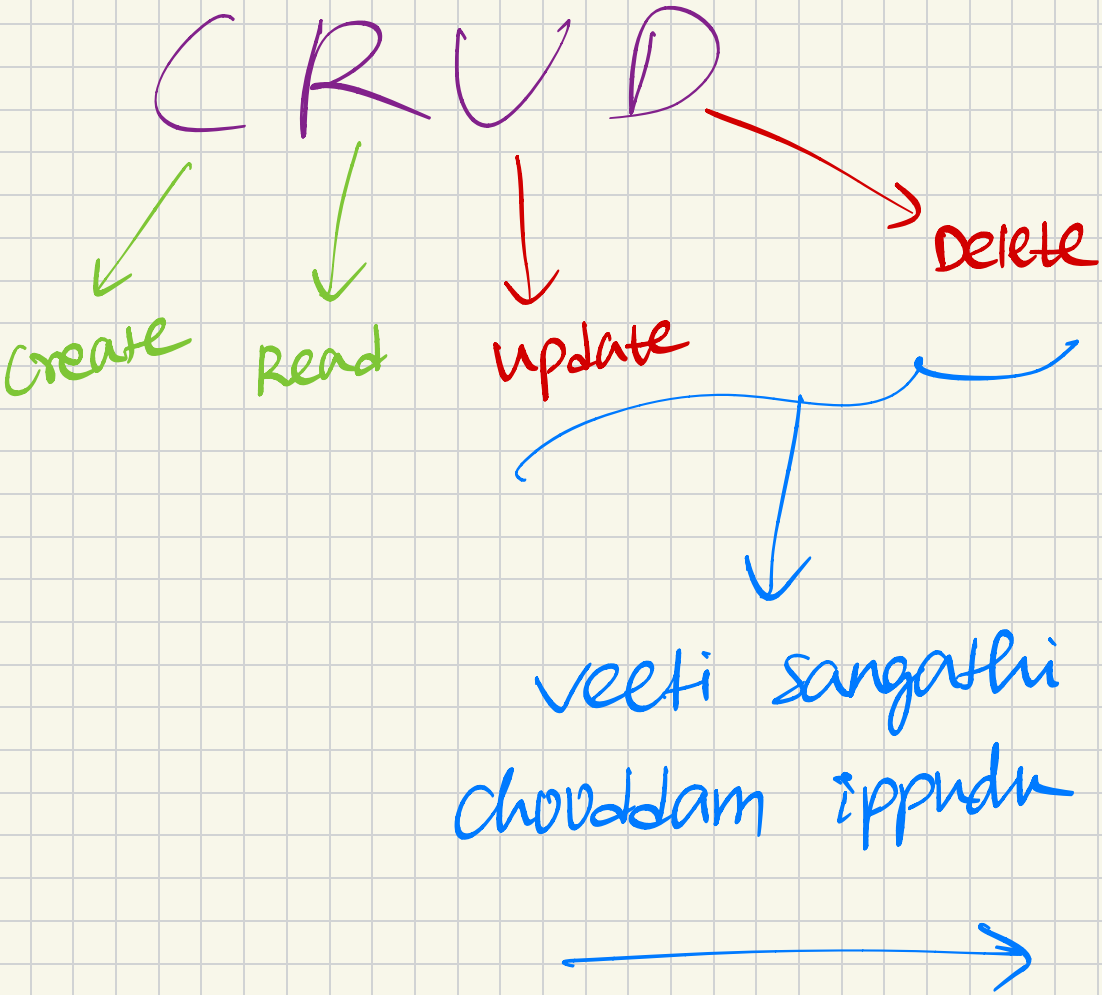
Languages that have a total box office collection greater than 500 million.

→ order of mentioning SQL clauses:

SELECT
FROM
JOIN
WHERE
GROUP BY
HAVING
ORDER BY
LIMIT

```
SELECT m.title, b.box_office_collection
FROM Movies m
JOIN Box_Office b ON m.movie_id = b.movie_id
WHERE m.language = 'Telugu' AND b.budget > 200000000
GROUP BY m.title
HAVING SUM(b.box_office_collection) > 0
ORDER BY b.box_office_collection DESC
LIMIT 3;
```

SELECT: Retrieves the movie title and box office collection.
FROM: Specifies the tables involved.
JOIN: Joins the Movies and Box_Office tables.
WHERE: Filters to include only Telugu movies with a budget greater than 200 million.
GROUP BY: Groups the results by movie title.
HAVING: Filters to include only movies with a positive box office collection.
ORDER BY: Orders the results by box office collection in descending order.
LIMIT: Limits the result set to the top 3 movies.



→ UPDATE:

Modify existing records

```
UPDATE table_name
SET column1 = value1, column2 = value2, ----
WHERE condition;
```

Example:

```
UPDATE Movies SET rating = 9.0
WHERE title = 'RRR';
```

→ DELETE:

Remove existing records

```
DELETE FROM table_name WHERE condition;
```

Example:

```
DELETE FROM Movies WHERE
title = 'KALKI 2898 AD';
```

→ cascading for foreign keys:

* ON DELETE CASCADE:

Automatically deletes all rows in the child table that reference the deleted row in the parent table.

```
CREATE TABLE Movies (
    movie_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255),
    release_year YEAR,
    genre VARCHAR(100),
    language VARCHAR(50) DEFAULT 'Telugu',
    duration_minutes INT,
    rating DECIMAL(3, 1)
);

CREATE TABLE Box_Office (
    box_office_id INT PRIMARY KEY AUTO_INCREMENT,
    movie_id INT,
    budget BIGINT,
    box_office_collection BIGINT,
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id)
    ON DELETE CASCADE
);
```

Behavior!

```
-- Delete the movie "RRR"
DELETE FROM Movies WHERE title = 'RRR';
```

*ON UPDATE CASCADE:

Automatically updates all rows in the child table when the referenced row in the parent table is updated.

```
CREATE TABLE Movies ( movie_id INT PRIMARY KEY
AUTO_INCREMENT, title VARCHAR(255), release_year YEAR,
genre VARCHAR(100), language VARCHAR(50) DEFAULT
'Telugu', duration_minutes INT, rating DECIMAL(3, 1) );

CREATE TABLE Box_Office ( box_office_id INT PRIMARY KEY
AUTO_INCREMENT, movie_id INT, budget BIGINT,
box_office_collection BIGINT, FOREIGN KEY (movie_id)
REFERENCES Movies(movie_id) ON UPDATE CASCADE );
```

Behavior:

```
-- Update the movie_id of "RRR"
UPDATE Movies SET movie_id = 10 WHERE title = 'RRR';
```

→ other cascading options!

SET NULL: If a row in the parent table is deleted or updated, the foreign key in the child table is set to NULL.
SET DEFAULT: If a row in the parent table is deleted or updated, the foreign key in the child table is set to a default value.
NO ACTION: The default behavior where no action is taken on the child table when the parent table is updated or deleted. This can lead to a constraint violation if the operation leaves orphaned rows in the child table.
RESTRICT: Prevents the delete or update operation on the parent table if there are related rows in the child table.

kaasepu tables tho aadukundam :)

→ ALTER TABLE:

```
ALTER TABLE table_name
ADD column_name datatype constraints;
```

↳ Add new columns

```
ALTER TABLE table_name
MODIFY column_name datatype constraints;
```

↳ Modify existing column

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

↳ Drop columns

```
ALTER TABLE Movies
ADD director_id INT;

ALTER TABLE Movies
MODIFY rating DECIMAL(4, 2);

ALTER TABLE Movies
DROP COLUMN duration_minutes;
```

→ DROP TABLE:

DROP TABLE table_name;

```
DROP TABLE Movies;
```

→ remove table from database

→ RENAME TABLE:

RENAME TABLE old_table_name TO new_table_name;

```
RENAME TABLE Movies TO Films;
```

→ Rename table

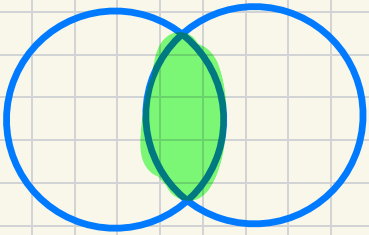
→ TRUNCATE TABLE:

TRUNCATE TABLE table_name;

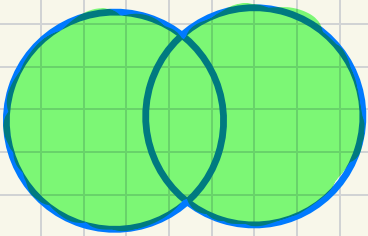
```
TRUNCATE TABLE Movies;
```

→ Remove all records in table without deleting actual table

→ JOINS:

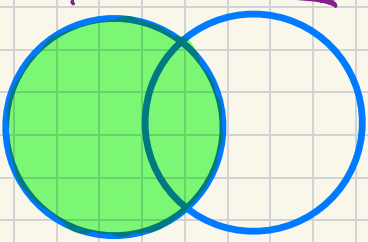


Inner Join

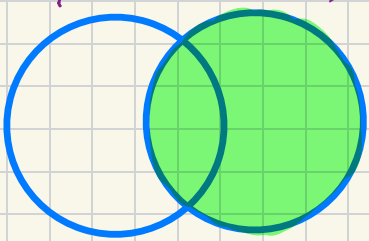


Full Join

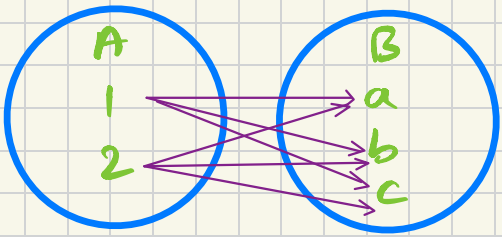
outer Join



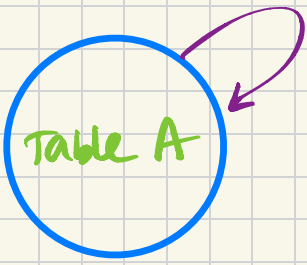
Left Join



Right Join



CROSS Join



Self Join

→ Inner Join:

returns matching values in both tables.

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

- Movies:

movie_id	title	director_id
1	RRR	101
2	Pushpa	102
3	KGF	103

- Directors:

director_id	name
101	S. S. Rajamouli
102	Sukumar
104	Prashanth Neel

```
SELECT m.title, d.name
FROM Movies m
INNER JOIN Directors d ON
m.director_id = d.director_id;
```

- Result:

title	name
RRR	S. S. Rajamouli
Pushpa	Sukumar

- **Explanation**: Only the movies with matching `director_id` in both `Movies` and `Directors` tables are returned.

→ Left Join:

returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

- **Movies:**

movie_id	title	director_id
1	RRR	101
2	Pushpa	102
3	KGF	103

- **Directors:**

director_id	name
101	S. S. Rajamouli
102	Sukumar
104	Prashanth Neel

Example:

```
SELECT m.title, d.name
FROM Movies m
LEFT JOIN Directors d ON m.director_id = d.director_id;
```

- **Result:**

title	name
RRR	S. S. Rajamouli
Pushpa	Sukumar
KGF	NULL

- **Explanation**: All movies are returned. For "KGF", which does not have a matching `director_id` in the `Directors` table, the `name` column returns `NULL`.

→ Right JOIN:

Similar to a LEFT JOIN, but it returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for columns from the left table.

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

- Movies:

movie_id	title	director_id
1	RRR	101
2	Pushpa	102
3	KGF	103

- Directors:

director_id	name
101	S. S. Rajamouli
102	Sukumar
104	Prashanth Neel

Example:

```
SELECT m.title, d.name
FROM Movies m
RIGHT JOIN Directors d ON
m.director_id = d.director_id;
```

- **Result:**

title	name
RRR	S. S. Rajamouli
Pushpa	Sukumar
NULL	Prashanth Neel

- **Explanation**: All directors are returned. For "Prashanth Neel", who does not have a matching `director_id` in the `Movies` table, the `title` column returns `NULL`.

→ FULL JOIN:

returns all rows when there is a match in either the left or right table. If there is no match, NULL values are returned for columns from the non-matching table.

```
SELECT columns
FROM table1
FULL JOIN table2
ON table1.column = table2.column;
```

- Movies:

movie_id	title	director_id
1	RRR	101
2	Pushpa	102
3	KGF	103

- Directors:

director_id	name
101	S. S. Rajamouli
102	Sukumar
104	Prashanth Neel

Example:

```
SELECT m.title, d.name
FROM Movies m
FULL JOIN Directors d ON m.director_id = d.director_id;
```

- **Result:**

title	name
RRR	S. S. Rajamouli
Pushpa	Sukumar
KGF	NULL
NULL	Prashanth Neel

- **Explanation**: All movies and all directors are returned. If there is no match, `NULL` is returned for columns from the table that doesn't have the match.

→ CROSS JOIN:

returns the cartesian product of the two tables, meaning it returns all possible combination of rows. This type of join is rarely used because it can produce a very large number of rows.

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

Example:

- **Movies:**

movie_id	title	director_id
1	RRR	101
2	Pushpa	102
3	KGF	103

- **Directors:**

director_id	name
101	S. S. Rajamouli
102	Sukumar
104	Prashanth Neel

```
SELECT m.title, d.name
FROM Movies m
CROSS JOIN Directors d;
```

- **Result:** (Example results if there are 3 movies and 3 directors)

title	name
RRR	S. S. Rajamouli
RRR	Sukumar
RRR	Prashanth Neel
Pushpa	S. S. Rajamouli
Pushpa	Sukumar
Pushpa	Prashanth Neel
KGF	S. S. Rajamouli
KGF	Sukumar
KGF	Prashanth Neel

- **Explanation:** Every movie is paired with every director.

→ SELF JOIN:

It is a join of a table with itself. This is useful for comparing rows within the same table.

```
SELECT a.column, b.column
FROM table a, table b
WHERE condition;
```

- Movies:

movie_id	title	director_id
1	RRR	101
2	Pushpa	102
3	KGF	103

Example:

```
SELECT a.title AS Movie1, b.title AS Movie2, a.release_year
FROM Movies a, Movies b
WHERE a.release_year = b.release_year AND a.movie_id < b.movie_id;
```

- **Result:**

Movie1	Movie2	release_year
RRR	Pushpa	2021

- **Explanation**: This query finds pairs of movies released in the same year.

→ UNION :

It is used to combine the result-set of two or more SELECT statements.

```
SELECT columns FROM table1
UNION
SELECT columns FROM table2;
```

- Telugu_Movies:

movie_id	title	release_year
1	RRR	2022
2	Pushpa	2021
3	KGF	2018

- Hindi_Movies:

movie_id	title	release_year
1	Dangal	2016
2	Bajrangi Bhaijaan	2015
3	KGF	2018

```
SELECT title, release_year
FROM Telugu_Movies
UNION
SELECT title, release_year
FROM Hindi_Movies;
```

- Result:

title	release_year
RRR	2022
Pushpa	2021
KGF	2018
Dangal	2016
Bajrangi Bhaijaan	2015

- **Explanation**: The result includes the titles from both tables, but only one instance of "KGF" (even though it's present in both tables) since `UNION` removes duplicates.

```
SELECT title, release_year
FROM Telugu_Movies
UNION ALL
SELECT title, release_year
FROM Hindi_Movies;
```

- Result:

title	release_year
RRR	2022
Pushpa	2021
KGF	2018
Dangal	2016
Bajrangi Bhaijaan	2015
KGF	2018

- **Explanation**: The result includes all titles from both tables, including both instances of "KGF" since `UNION ALL` does not remove duplicates.

→ Subqueries:

Subquery in the WHERE Clause

Example: Find movies that have a higher budget than the average budget of all movies.

```
SELECT title, budget
FROM Movies
WHERE budget > (SELECT AVG(budget) FROM Movies);
```

Explanation: The subquery (SELECT AVG(budget) FROM Movies) calculates the average budget of all movies. The outer query then selects the titles and budgets of movies where the budget is higher than this average.

Subquery in the SELECT Clause

Example: List all movies with their respective directors' names.

```
SELECT title,
    (SELECT name FROM Directors WHERE Directors.director_id = Movies.director_id) AS
director_name
FROM Movies;
```

Explanation: The subquery in the SELECT clause retrieves the name of the director for each movie.

Subquery in the FROM Clause

Example: Find the average budget of movies released each year.

```
SELECT release_year, AVG(budget) AS average_budget
FROM (SELECT release_year, budget FROM Movies) AS yearly_budgets
GROUP BY release_year;
```

Explanation: The subquery in the FROM clause retrieves the release_year and budget for all movies. The outer query then calculates the average budget per year.

Subquery in the HAVING Clause

Example: Find genres that have an average movie rating higher than the overall average movie rating.

```
SELECT genre, AVG(rating) AS avg_genre_rating
FROM Movies
GROUP BY genre
HAVING AVG(rating) > (SELECT AVG(rating) FROM Movies);
```

Explanation: The subquery (SELECT AVG(rating) FROM Movies) calculates the overall average rating for all movies. The main query groups the movies by genre and calculates the average rating for each genre. The HAVING clause filters out any genres where the average rating (AVG(rating)) is not greater than the overall average rating returned by the subquery.

THANK YOU

-Dodagatta Nihar