

# The Complete SQL HandBook

•=Comprehensive Notes and Tips=•

HandWritten By  
Bhavana

# Contents

1. Introduction to Databases

2. Introduction to SQL [Basic Concepts]:

2.1. Creating Tables

2.2. Inserting Rows

2.3. Retrieving Data

2.4. Update Rows

2.5. Delete Rows

2.6. Alter Tables

3. Querying with SQL:

3.1. Comparison Operators

3.2. String Operators

3.3. Logical Operators

3.4. In and Between Operators

3.5. Order By and Distinct

4. Aggregations and Group By

4.1. Aggregations

4.2. Group By

5. Common Concepts:

5.1. SQL Expressions

5.2. SQL Functions

6. Modeling Databases:

6.1. Core Concepts of ER Model

6.2. Creating a Relational Database

## 7. Joins:

7.1. Natural Join

7.2. Inner Join

7.3. Left Join

7.4. Right Join

7.5. Full Join

7.6. Cross Join

## 8. Views and Subqueries

## 9. Transaction and Indexes

9.1. Transactions

9.2. Indexes

# Introduction to Databases

## Concepts in focus

- \* Data

- \* Database

- \* Database Management System (DBMS)

- Advantages

- \* Types of Databases

- Relational Database

- Non-Relational Database

⇒ Data :-

→ Any sort of information that is stored is called data.

Examples:- 1. Messages & multimedia on WhatsApp.

2. products and order on Amazon

3. Contact details in telephone directory,  
etc.

⇒ Database :-

→ An organized collection of data is called a database.

⇒ Database Management System (DBMS) :-

→ A software that is used to easily store and access data from the database in a secure way.



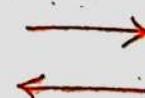
web Application  
Servers



AI Models



Recommendation  
Engines



Database

## • Advantages

- \* Security: Data is stored & maintained securely.
- \* Ease of Use: provides simpler ways to create & update data at the rate it is generated and updated respectively.
- \* Durability and Availability: Durable and provides access to all the clients at any point in time.
- \* Performance: - quickly accessible to all the clients (applications and stakeholders).

## → Types of Databases:-

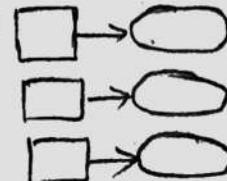
→ There are different types of databases based on how we organize the data.



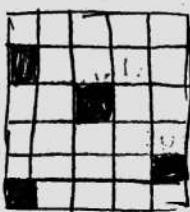
Relational



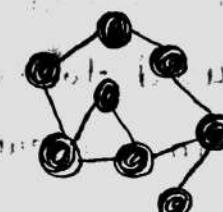
Analytical



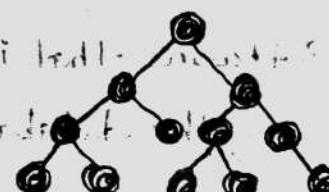
key value



Column family



Graph



Document

## • Relational Database:-

1	2	3

In relational databases, the data is organized in the form of tables.

## Non-Relational Database

- Graph, keyvalue, column family, Document.
- These four types are commonly referred as non-relational databases.

### Note:-

- \* Choice of database depends on our requirements.
- \* Relational database is the most commonly used database.

### ⇒ Relational DBMS:-

- A Relational DBMS is a DBMS designed specifically for relational database. Relational databases organise the data in the form of tables.

**Examples:-** Oracle, PostgreSQL, MySQL, SQLite, SQL Server, IBM DB2, etc.

### ⇒ Non-Relational DBMS:-

- A Non-relational DBMS is a DBMS designed specifically for non-relational databases. Non-relational databases store the data in a "non-tabular" form.

**Examples:-** Elasticsearch, CouchDB, DynamoDB, MongoDB, Cassandra, Redis, etc.

## ② Introduction to SQL [Basics]

- SQL stands for Structured Query language
- SQL is used to perform operation on Relational DBMS
- SQL is declarative. Hence, easy to learn.
- SQL provides multiple clauses (commands) to perform various operations like create, retrieve, update and delete the data.

### \* Create Table :-

- Creates a new table in the database

**Syntax :-**

```
CREATE TABLE table-name(  
    column1 type1,  
    column2 type2,  
    ....  
)
```

Here, **type1** and **type2**, in the syntax are the datatypes of **column1** and **column2** respectively. Datatypes that are supported in SQL are mentioned in [this article](#).

### Example :-

Create a player table to store the following details of players.

column-name	data-type
name	VARCHAR(200)
age	INT / INTEGER
score	INT / INTEGER

CREATE TABLE player(

name VARCHAR(200),

age INTEGER,

score INTEGER

) ;

\* We can check the details of the created table at any point in time using the **PRAGMA** command.

Data Types :-

→ Following data types are frequently used in SQL.

Datatype	Syntax
Integer	INT / INTEGER
float	FLOAT
String	VARCHAR
Text	TEXT
Date	DATE
Time	TIME
Datetime	DATETIME
Boolean	BOOLEAN

Note :-

1. Boolean values are stored as integers 0(FALSE) and 1(TRUE)

2. Date object is represented as : 'YYYY-MM-DD'

3. Datetime object is represented as: 'YYYY-MM-DD HH:MM:SS'

## PRAGMA

**PRAGMA TABLE\_INFO** command returns the information about a specific table in a database.

### Syntax:-

PRAGMA TABLE\_INFO(table-name);

### Example:-

Let's find out the information of the employee table that is present in the database.

PRAGMA TABLE\_INFO(employee);

### Note:-

If the given table name does not exist, PRAGMA TABLE\_INFO doesn't give any result.



## Inserting Rows :-

→ **INSERT** clause is used to insert new rows in a table.

### Syntax:-

INSERT INTO

table-name(column1, column2, ..., columnN)

VALUES

(value1, value2, ..., valueN),

(value1, value2, ..., valueN),

\* Any number of rows from 1 to n can be inserted into a specified table using the above syntax.

## Database

The player table that stores the details of players in a tournament respectively.  
→ player table store the name, age and score of players.

### Example:-

Insert name, age and score of 2 players in the player table.

INSERT INTO

player(name, age, score)

VALUES

( "Rakesh", 39, 35),

( "Sai", 47, 38);

Upon executing the above data code, both the entries would be added to the player table.

Let's View the added data!

→ We can retrieve the inserted data by using the following command

SELECT \*

FROM player

### Output:-

name	age	score
Rakesh	39	35
Sai	47	38

## possible Mistakes:-

### Mistake 1:

→ The number of values that we're inserting, must match with the number of column names that are specified in the query.

SQL:- `INSERT INTO player(name, age, score)`  
VALUES  
( "Virat", 31 );

### OUTPUT:-

Error: 2 values for 3 columns given

### Mistake 2:

→ we have to specify only the existing tables in the database.

SQL:- `INSERT INTO player_Information (name, age, score)`  
VALUES  
( "Virat", 31, 38 );

### output:-

Error: no such table: player\_Information

### Mistake 3:

→ Do not add additional parenthesis () post VALUES key-word in the code.

SQL:- `INSERT INTO player(name, age, score)`

## VALUES

```
( ("Rakesh", 39, 35), ("Sai", 39, 40));
```

### Output:-

Error: 2 values for 3 columns

### Mistake 4:-

→ while inserting data, be careful with the datatypes of the input values. Input value datatype should be same as the column datatype.

## INSERT INTO

```
player(name, age, score)
```

## VALUES

```
("Virat", 30, "Hundred");
```

### Warning:-

If the datatype of the input value doesn't match with the datatype of column, ~~SOLITE~~ doesn't raise an error

## Retrieving Data:-

SELECT clause is used to retrieve rows from a table.

### Database:-

The database consists of a player table stores the details of players who are a part of a tournament. player table stores the name, age and score of players.

## ⇒ Selecting Specific Columns:-

→ To retrieve the data of only specific columns from a table, add the respective column names in the SELECT clause.

## Syntax:-

```
SELECT  
    column1,  
    column2,  
    ...  
    columnN
```

```
FROM  
    table-name;
```

## Example:-

Let's fetch the **name** and **age** of the players from the **player** table.

```
SELECT  
    name,  
    age
```

```
FROM  
    player;
```

## OUTPUT:-

<u>name</u>	<u>age</u>
Virat	32
Rakesh	39
Sai	47
	...

## ⇒ Selecting All Columns

→ Sometimes, we may want to select all the columns from a table. Typing out every column name, for every time we have to retrieve the data, would be a pain.

## Syntax:-

```
SELECT *  
FROM table-name;
```

## Example

Get all the data of players from the player table.

SELECT \*

FROM player;

Output:-

<u>name</u>	<u>age</u>	<u>score</u>
Virat	32	50
Rakesh	39	35
Sai	41	30
...	...	...

## ⇒ Selecting Specific Rows

We use WHERE clause to retrieve only specific rows.

Syntax:

SELECT \*

FROM table-name

WHERE condition;

\* WHERE clause specifies a condition that has to be satisfied for retrieving the data from a database.

Example:

Get name and age of the player whose name is "Ram" from the player table

SELECT \*

FROM player

WHERE name = "Sai";

Output:-

<u>name</u>	<u>age</u>	<u>score</u>
Sai	41	30

## \* Update Rows

UPDATE clause is used to update the data of an existing table in database. We can update all the rows or only specific rows as per the requirement.

⇒ Update all Rows

Syntax:-

```
UPDATE  
    table-name  
SET  
    column1 = value1;
```

Example:-

```
UPDATE  
    player  
SET  
    score = 100;
```

⇒ Update Specific Rows

Syntax:-

```
UPDATE  
    table-name  
SET  
    column1 = value1
```

WHERE

```
    column2 = value2;
```

Example:-

```
UPDATE  
    player  
SET  
    score = 150  
WHERE  
    name = 'Ram';
```

## Delete Rows

DELETE clause is used to delete existing records from a table.

### ⇒ Delete All Rows

Syntax:-

```
DELETE FROM  
    table-name;
```

Example:-

Delete all the Rows from player table

```
DELETE FROM  
    player;
```

### ⇒ Delete Specific Rows

Syntax:-

```
DELETE FROM  
    table-name  
WHERE  
    column1 = value1;
```

Example:-

\* Delete 'shyam' from the player table

Note:- We can uniquely identify a player by name.

```
DELETE FROM
```

```
player
```

```
WHERE
```

```
name = "shyam";
```

Warning :- We can not retrieve the data once we delete the data from the table.

## ⇒ **DROP table:-**

DROP clause is used to delete a table from the database.

### Syntax:-

**DROP TABLE**

table-name

### Example:-

\* Delete player table from the database

**DROP TABLE player;**



## Alter Table:-

ALTER clause is used to add, delete, or modify columns in an existing table.

## ⇒ **Add Column**

### Syntax

**ALTER TABLE**

table-name

**ADD**

column-name datatype;

### Example:-

Add a new column jersey-num of type integer to the player table.

ALTER TABLE

player

ADD

jersey-num INT;

### Note:-

Default values for newly added columns in the existing rows will be NULL.

### ⇒ Rename Column

#### Syntax:-

ALTER TABLE

table-name RENAME COLUMN c1 TO c2;

#### Example:-

Rename the column jersey-num in the player table to jersey-number.

ALTER TABLE

player RENAME COLUMN jersey-num TO jersey-number;

## → Drop Column :-

Syntax:-

ALTER TABLE

table\_name DROP COLUMN column-name;

Example:-

Remove the column jersey-number from the player table.

ALTER TABLE

player DROP COLUMN jersey-number;

Note:- DROP COLUMN is not supported in some DBMS, including SQLite.

# → Querying With SQL

## \* Comparison Operators:-

Ex:-

In a typical e-commerce scenario, users would generally filter the products with good ratings, or want to purchase the products of a certain brand or of a certain price. Let's see how comparison operators are used to filter such kind of data using the following database.

→ Database:-

The database contains a product table that stores the data of products like name, category, price, brand and rating.

## Comparison Operators:-

operator	Description
=	Equals to
<>	Not equals to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

## Examples:-

1. Get all the details of the products whose category is "Food" from the product table.

SELECT

\*

FROM

product

WHERE

category = "Food";

Output:-

name	category	price	brand	rating
Chocolate cake	Food	25	Britannia	3.7
Stracberry cake	Food	60	Cadbury	4.1
Chocolate cake	Food	60	Cadbury	2.5
....	...	...	...	...

2. Get all the details of the products that does not belongs to Food category from the product table.

SELECT

\*

FROM

product

WHERE

category  $\neq$  "Food".

Output:-

name	category	price	brand	rating
Blue shirt	Clothing	750	Denim	3.8
Blue jeans	Clothing	200	puma	3.6
Black jeans	Clothing	750	Denim	4.5
....	...	...	...	...

\* Similarly, we can use other comparison operators like greater than ( $>$ ), greater than or equal to ( $\geq$ ), less than ( $<$ ), less than or equal to ( $\leq$ ) to filter the data as per the requirement.

## String Operations :-

### LIKE Operator

→ LIKE operator is used to perform queries on strings. This operator is especially used in WHERE clause to retrieve all the rows that match the given pattern.

→ We write patterns using the following wildcard characters:

Symbol	Description	Example
percent sign (%)	Represents zero or more characters	ch% finds ch, chips, chocolates...
underscore (_)	Represents a single character	_at finds mat, hat, bat...

### Common patterns:-

pattern	Example	Description
Exact Match	WHERE name LIKE "Mobiles"	Retrieves products whose name is exactly equal to "mobiles".
Starts with	WHERE name LIKE "mobiles%"	Retrieves products whose name starts with "mobiles".
Ends with	WHERE name LIKE "%mobiles"	Retrieves products whose name ends with "mobiles".
Containing	WHERE name LIKE "%mobiles%"	Retrieves products whose name contains with "mobiles".

pattern Matching	WHERE name LIKE "a-%"	Retrieves products whose name starts with "a" and have at least 2 characters in length.
---------------------	--------------------------	---

## Syntax:-

```

SELECT *
FROM
    table_name
WHERE
    c1 LIKE matching_pattern;
  
```

ANSWER

## Examples:-

- Get all the products in the "Gadgets" category from the **product** table.

```

SELECT *
FROM
    product
WHERE
    category LIKE "Gadgets";
  
```

## Output:-

name	category	price	brand	rating
Smart Watch	Gadgets	17000	Apple	4.9
Smart Cam	Gadgets	2600	Realme	4.7
Smart TV	Gadgets	40000	Sony	4.0
Realme Smart Band	Gadgets	3000	Realme	4.6

2. Get all the products whose name starts with "Bourbon" from the product table

```
SELECT *  
FROM product  
WHERE name LIKE "Bourbon%";
```

\* Here % represents that, following the string "Bourbon", there can be 0 or more characters.

Output:-

name	category	price	brand	rating
Bourbon Small	food	10	Britannia	3.9
Bourbon Special	food	15	Britannia	4.6
Bourbon with extra cookies	food	30	Britannia	4.4

3. Get all smart electronic products i.e., name contains 'smart' from the product table

```
SELECT *  
FROM product  
WHERE name LIKE "%smart%";
```

\* Here % before and after the "string" represents that there can be 0 or more characters "succeeding or preceding the string".

Output:-

name	category	price	brand	rating
Smart Watch	Gadgets	17000	Apple	4.9
Smart Cam	Gadgets	2600	realme	4.7
Smart TV	Gadgets	40000	Sony	4
Realme Smart Band	Gadgets	3000	Realme	4.6

4. Get all the products which have exactly 5 characters in brand from the product table.

SELECT

\*

FROM

product

WHERE

brand LIKE '\_\_\_\_';

Output:-

name	category	price	brand	rating
Blue Shirt	Clothing	750	Denim	3.8
Black Jeans	Clothing	150	Denim	4.5
Smart Watch	Gadgets	17000	Apple	4.9
....	...	...	....	"

Note:-

The percent Sign (%) is used when we are not sure of the number of characters present in the string.

If we know the exact length of the string, then the wildcard character underscore(\_) comes in handy.

## \* Logical Operators :-

→ Based on with logical operators, we can perform queries based on multiple conditions. Let's learn.

→ AND, OR, NOT

Operator	Description
AND	Used to fetch rows that satisfy two or more conditions
OR	Used to fetch rows that satisfy at least one of the given conditions
NOT	Used to negate a condition in the WHERE clause

Syntax:-

```
SELECT  
*  
FROM  
table_name.  
WHERE  
Condition 1,  
operator Condition 2  
operator Condition 3  
.....;
```

Example:

1. Get all the details of the products whose

\* category is "clothing" and

\* price less than equal to 1000 from the product table.

```
SELECT  
*  
FROM  
product  
WHERE  
category = "clothing"  
AND price <= 1000;
```

Output:-

name	category	price	brand	rating
Blue Shirt	Clothing	750	Denium	3.8
Blue Jeans	Clothing	800	puma	3.6
Black Jeans	Clothing	750	Denium	4.5
...	...	...	...	...

2. Ignore all the products with name containing "cake" from the list of products.

SELECT

\*

FROM

product

WHERE

NOT name LIKE "%cake%";

Output:-

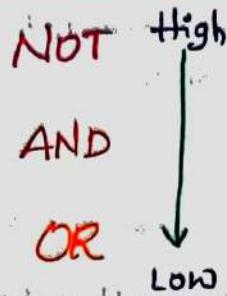
name	category	price	brand	rating
Blue Shirt	Clothing	750	Denium	3.8
Blue Jeans	Clothing	800	puma	3.6
Black Jeans	Clothing	750	Denium	4.5
...	...	...	...	...

### ⇒ Multiple Logical Operators:-

We can also use the combinations of logical operators to combine two or more conditions. These compound conditions enable us to fine-tune the data retrieval requirements.

### precedence

→ When a query has multiple operators, operator precedence determines the sequence of operations.



order of precedence:-

- \* NOT
- \* AND
- \* OR

**Example:-**

fetch the products that belongs to

- \* Redmi brand and rating greater than 4 or
- \* the products from oneplus brand

**SELECT**

\*

**FROM**

product

**WHERE**

brand = "Redmi"

AND rating > 4

OR brand = "Oneplus";

\* In the above query, AND has the precedence over OR.  
So, the above query is equivalent to;

**SELECT**

\*

**FROM**

product

**WHERE**

( brand = "Redmi" )

AND rating > 4 )

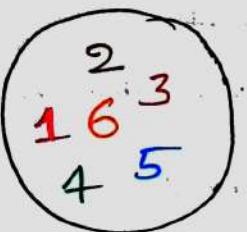
OR brand = "Oneplus";

**Quick Tip:-** It is suggested to always use parenthesis to ensure correctness while grouping the conditions.

## \* IN and BETWEEN Operators

### IN Operator:-

Retrieves the corresponding rows from the table if the value of column(c1) is present in the given values (v<sub>1</sub>, v<sub>2</sub>, ...)



### Syntax:-

```
SELECT  
*  
FROM  
table_name  
WHERE  
c1 IN (v1, v2, ...);
```

### Example:-

Get the details of all the products from product table, where the brand is either "puma", "Maffti", "Levi's", "Lee" or "Denim".

```
SELECT  
*  
FROM  
product  
WHERE  
brand IN ("puma", "Levi's", "Maffti", "Lee", "Denim")
```

Output:-

name	category	price	brand	rating
Blue shirt	clothing	750	Denim	3.8
Blue jeans	clothing	800	puma	3.6
Black jeans	clothing	150	Denim	4.5
....	...	...	...	...

→ BETWEEN Operator:-

→ Retriever's all the rows from table that have column(c1) value present between the given range (v1 and v2).

1 ----- 9

Syntax:-

SELECT

\*

FROM

table-name

WHERE

c1 BETWEEN v1

AND v2;

Note:- BETWEEN operator is inclusive, i.e., both the lower and upper limit values of the range are included.

Example:-

Find the products with price ranging from 1000 to 5000

SELECT

name

price

brand

FROM  
product

WHERE

price BETWEEN 1000  
AND 5000;

Output:-

name	price	brand
Blue shirt	1000	puma
Smart Cam	2600	Realme
Realme Smart Band	3000	Realme

possible Mistakes:-

1. When using the BETWEEN operator, the first value should be less than second value. If not, we'll get an incorrect result depending on the DBMS.

SELECT

name,  
price,  
brand

FROM

product

WHERE

price BETWEEN 500  
AND 300;

Output:-

name price brand

2. we have to give both lower limit and upper limit while specifying range.

**SELECT**

name,

price,

brand

**FROM**

product

**WHERE**

price BETWEEN

**AND** 300;

OUTPUT:-

Error near " AND": Syntax error

3. The data type of the column for which we're using the BETWEEN operator must match with the data types of the lower and upper limits.

**SELECT**

name,

price,

brand

**FROM**

product

**WHERE**

name BETWEEN 300

**AND** 500;

Output:-

name	price	brand

# \* ORDER BY and DISTINCT

## ORDER BY

we use ORDER BY clause to order rows. By default, ORDER BY sorts the data in the ascending order.

Element	Apple
FOX	
Balloon	Balloon
COW	COW
APPLE	Dog
Dog	Element
	FOX

## Syntax:-

```
SELECT column1,  
       column2,  
       ... columnN,  
FROM table-name [WHERE condition]
```

## ORDER BY

column1 ASC / DESC,

column2 ASC / DESC;

Example :- Get all products in order of lowest price and highest rating in "puma" brand.

SELECT name, price, rating

FROM puma

ORDER BY price

ASC, rating

DESC;

**FROM**

product

**WHERE**

brand = "puma"

**ORDER BY**

price ASC,

rating DESC;

Output:-

name	price	rating
Black shirt	600	4.8
Blue jeans	800	3.6
Blue shirt	1000	4.3

**DISTINCT**

DISTINCT clause is used to return the distinct i.e., unique values.

Syntax:-

**SELECT**

DISTINCT column 1,

column 2,...

column N

**FROM**

table\_name

**WHERE**

[condition];

### Example :-

Get all the brands present in the product table

**SELECT**

**DISTINCT brand**

**FROM**

**product**

**ORDER BY**

**brand;**

### Output:-

Brand

Absa

Apple



### Pagination:-

Using pagination, only a chunk of the data can be sent to the user based on their request. And, the next chunk of data can be fetched only when the user asks for it.

→ We use **LIMIT** and **OFFSET** clauses, to select a chunk of the results.

### **LIMIT:-**

LIMIT clause is used to specify the number of rows( $n$ ) we would like to have in result.

## Syntax:-

```
SELECT column1,  
       column2,...  
       columnN  
FROM   table-name  
LIMIT n;
```

## Example:-

```
SELECT name,  
       price,  
       rating  
FROM   product  
WHERE  brand = "puma"  
ORDER BY rating DESC  
LIMIT 2;
```

## Output:-

name	price	rating
Black shirt	600	4.8
Blue shirt	1000	4.3

### Note:-

If the limit value is greater than the total number of rows, then all rows will be retrieved.

### OFFSET

OFFSET clause is used to specify the position (from  $(n+1)$ th row) from where the chunk of the results are to be selected.

### Syntax:-

```
SELECT  
    column1,  
    column2,...  
    columnN  
FROM  
    table_name  
LIMIT m  
OFFSET n;
```

### Example:-

Q. Get all the details of 5 top-rated products, starting from 7th row.

```
SELECT  
    name,  
    price,  
    rating  
FROM  
    product  
ORDER BY  
    rating DESC  
LIMIT 5  
OFFSET 6;
```

## Output:-

name	price	rating
Burbon Special	15	4.6
Realme Smart Band	3000	4.6
Harry Potter and the Goblet of fire	431	4.6
Black Teanz	750	4.5
potato chips cream & onion	63	4.5

## Possible Mistakes:-

- \* Using "OFFSET" before the "LIMIT" clause.

```

SELECT
*
FROM
product
    OFFSET 2
    LIMIT 4;
  
```

Output:- Error: near "2"; Syntax error

- \* Using only "OFFSET" clause

```

SELECT
*
FROM
product
    OFFSET 2;
  
```

Output:- Error! near "2"; Syntax error

Note:- "OFFSET" clause should be placed after the "LIMIT" clause. Default "OFFSET" value is 0.

**Notes by Bhavana**

4

## Aggregations and Group By :-

→ we perform aggregations in such scenarios to combine multiple values into a single value, i.e., individual scores to an average score.

### Aggregation Functions:-

Combining multiple values into a single value is called aggregation. Following are the functions provided by SQL to perform aggregations on the given data.

Aggregation Functions	Description
COUNT	Counts the number of values
SUM	Adds all the values
MIN	Returns the minimum value
MAX	Returns the maximum value
AVG	Calculate the average of the values

### Syntax:-

SELECT

aggregate\_function (C1)

aggregate\_function (C2)

FROM

TABLE;

### Note:-

We can calculate multiple aggregate functions in a single query.

### Examples:-

1. Get the total runs scored by "Ram" from the player-match-details table.

SELECT

SUM(score)

FROM

player-match-details

WHERE

name = "Ram";

Output:-

SUM(score)

221

2. Get the highest and least scores among all the matches that happened in the year 2011.

SELECT

MAX(score),

MIN(score)

FROM

player-match-details

WHERE

year = 2011;

Output:-

MAX(score)

MIN(score)

75

62

## COUNT Variants:-

\* Calculate the total number of matches played in the tournament.

### Variant 1:-

```
SELECT COUNT(*)  
FROM player-match-details;
```

### Variant 2:-

```
SELECT COUNT(1)  
FROM player-match-details;
```

### Variant 3:-

```
SELECT COUNT()  
FROM player-match-details;
```

Output of Variant 1, Variant 2 and Variant 3

All the variants i.e., Variant 1, Variant 2 and Variant 3 give the same result: 18.

### Note:-

In SQL, there's a difference between using COUNT(\*) and COUNT(column\_name) :

COUNT(\*) : This function counts the total number of rows in a table, regardless of whether any specific column contains NULL values. It counts all rows, including those NULL values, and returns the total count.

COUNT (column-name): This function counts the number of Non-NULL values in the specified column. It excludes NULL values from the count and only considers the Non-NULL values within the specified column.

### Special Cases:

→ When SUM function is applied on non-numeric data types like strings, date, time, datetime, etc., SQLite DBMS returns 0.0 and PostgreSQL DBMS returns none.

→ Aggregate functions on strings and their outputs :-

<u>Aggregate functions</u>	<u>Output</u>
MIN, MAX	Based on lexicographic ordering
SUM, AVG	0 (depends on DBMS)
COUNT	Default behaviour

→ NULL Values are ignored while computing the aggregation values.

→ When aggregate functions are applied on only NULL values

<u>Aggregate functions</u>	<u>Output</u>
MIN	NULL
MAX	NULL
SUM	NULL
COUNT	0
AVG	NULL

## Alias

→ Using keyword AS, we can provide alternate temporary names to the columns in the output.

### Syntax

```
SELECT  
    c1 AS a1,  
    c2 AS a2,  
    ...  
FROM  
    table-name;
```

### Example

→ Get all the names of players with column name as "player\_name".

```
SELECT  
    name AS player-name  
FROM  
    player-match-details;
```

### Output:-

<u>player-name</u>
Ram
Joseph
...

# Group By with Having

## GROUP BY

The GROUP BY clause in SQL is used to group rows which have same values for the mentioned attributes.

### Syntax:-

```

SELECT
    c1
    aggregate_function(c2)
FROM
    table_name
GROUP BY c1;
  
```

### Example:-

\* Get the total score of each player in the database

```

SELECT
    name, SUM(score) as total_score
FROM
    player_match_details
GROUP BY name;
  
```

### Output:-

<u>name</u>	<u>total_score</u>
David	105
Joseph	116
Lokesh	186
...	...

## GROUP BY with WHERE

⇒ We can use WHERE clause to filter the data before performing aggregation.

Syntax:-

SELECT

C<sub>1</sub>,

aggregate\_function(C<sub>2</sub>)

FROM

table-name

WHERE

C<sub>3</sub> = v<sub>1</sub>

GROUP BY C<sub>1</sub>;

Example:- Get the number of half-centuries scored by each player

SELECT

name, COUNT(\*) AS half-centuries

FROM

player-match-details

WHERE score >= 50

GROUP BY name;

Output:-

<u>name</u>	<u>half-centuries</u>
David	1
Joseph	2
Lokesh	3
...	...

## HAVING :-

HAVING clause is used to filter the resultant rows after the application of GROUP BY clause.

### Syntax:-

```

SELECT
    c1,
    c2,
    aggregate_function(c1)
FROM
    table_name
GROUP BY
    c1, c2

```

### HAVING

condition;

Example:- Get the name and number of half-centuries of players who scored more than one half century.

```

SELECT
    name
    count(*) AS half_centuries
FROM

```

player-match-details

WHERE

score >= 50

GROUP BY

name

HAVING half\_centuries > 1;

Output:-

<u>name</u>	<u>half_centuries</u>
-------------	-----------------------

Lokesh	2
--------	---

Ram	3
-----	---

Note:- WHERE vs HAVING :- WHERE is used to filter rows and this operation is performed before grouping.

→ HAVING is used to filter groups and this operation is performed after grouping.

## ⑤ Common Concepts

### \* SQL Expressions

→ We can write expressions in various SQL clauses. Expressions can comprise of various data types like integers, floats, strings, datetime etc.

#### → Using Expressions in SELECT clause

\* Example:- Get profits of all movies.

Note :- Consider profit as difference between collection and budget.

**SELECT**

    id, name, (collection\_in\_cr - budget\_in\_cr) as profit  
**FROM**  
    movie;

Output:-

<u>id</u>	<u>name</u>	<u>profit</u>
1	The matrix	40.31
2	Inception	67.68
3	The Dark night	82.5
...	...	...

Note:-

We use "||" Operator to concatenate strings in sqlite3

Example 2:- Get the movie name and genre in the following format : movie-name = genre

**SELECT**

name || " - " || genre AS movie-genre

**FROM**

movie;

Output:

movie\_genre

The Matrix - Sci-Fi

Inception - Action

The Dark knight - Drama

Toy Story 3 - Animation

...

→ Using Expressions in WHERE Clause:

Example:- Get all the movies with a profit at least 50 crores.

**SELECT**

\*

**FROM**

movie

**WHERE**

(collection\_in\_cr - budget\_in\_cr)  $\geq 50$ ;

Output:-

<u>id</u>	<u>name</u>	<u>genre</u>	<u>budget_in_cr</u>	<u>collection_in_cr</u>	<u>rating</u>	<u>release</u>
2	Inception	Action	16.0	82.68	8.8	2010-07-24
3	The dark knight	Action	18.0	100.5	9.0	2008-07-16
4	Toy Story 3	Animation	20.0	100.67	8.5	2010-06-25
...	....	...	...	...	...	...

## ⇒ Using Expressions in UPDATE clause.

Example :- Scale down ratings from 10 to 5 in movie table

UPDATE

SET rating = rating / 2

## ⇒ Expressions in HAVING Clause:-

Example :-

Get all the genres with an average profit of at least 100 crores.

SELECT

genre

FROM

movie

GROUP BY

genre

HAVING

Avg(collection\_in\_cr - budget\_in\_cr) >= 100;

Output:-

genre

Action

Animation

Mystery

...

...

## SQL Functions:-

- SQL provides many built-in functions to perform various operations over data that is stored in tables.
- SQL functions can be divided into different categories such as:  
(1) Date functions (2) cast Functions (3) Arithmetic functions

### Date Functions:

→ Date functions are used to extract the date or time from a datetime field. One important function in date functions is the strftime() function.

#### strftime() :-

strftime() function is used to extract year, month, day, hour, etc. from a date (or) datetime field based on a specific format as strings.

#### Syntax:

strftime (format, field-name)

#### Example:-

strftime ("%Y", release\_date)

→ Various formats in date functions with an example:-

<u>Format</u>	<u>Description</u>	<u>function</u>
%Y	Year	strftime ("%Y", field-name)
%m	month	strftime ("%m", field-name)
%d	day	strftime ("%d", field-name)
%H	Hour	strftime ("%H", field-name)

## How to use strftime()

1. Choose the format of the datetime that you want, such as the year, the month, or the day, etc.
2. Write the function using `strftime(Format, field_name)` in SQL query.

Note:- `strftime()` extracts date and time in the String format.

Example:- Get the movie title and year for every movie from the database

```
SELECT  
    name,  
    strftime ('%Y', release_date)  
FROM  
    movie;
```

Output:-

<u>name</u>	<u>strftime ('%Y', release_date)</u>
The Matrix	1999
Inception	2010
The Dark Knight	2008
...	
...	

## CAST Function :-

In database management systems, the CAST function is used to convert a value from one data type to another data type.

### Syntax:-

CAST(value as data-type)

### Example :-

CAST(strftime('%Y', release\_date) AS integer)

→ The CAST function takes :-

1. Value :- The value that you want to convert into a specific data type.

2. Data type :- The data type to which you want to convert the value.

Example :- Find how many movies were released in each month of the year 2010.

SELECT

strftime("%m", release\_date) AS month,  
COUNT(\*) AS total\_movies

FROM

movie

WHERE

CAST(strftime('%Y', release\_date) AS integer) = 2010

GROUP BY

month;

## ArithmetiC Functions:-

→ ArithmetiC Functions in SQL are used to perform mathematical operations on numeric values. Some commonly used arithmetic function are FLOOR, CEIL, ROUND

### FLOOR Function :-

→ The FLOOR function rounds a number to the nearest integer below its current value.

#### Syntax:-

$\text{FLOOR}(\text{number})$

#### Example:-

SELECT FLOOR(2.3);

#### Output:-

FLOOR  
2

### CEIL Function:-

⇒ The CEIL function rounds a number to the nearest integer above its current value.

#### Syntax:-

$\text{CEIL}(\text{number})$

#### Example:-

SELECT CEIL(-2.7);

#### Output:-    CEIL

-2

## → ROUND Function

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

### Syntax

ROUND(number, decimal\_places)

### Example

SELECT ROUND(2.345, 2);

SELECT ROUND(2.345, 1);

### Output:-

#### ROUND

2.35

2.3

## String Functions

→ String functions in SQL are used to manipulate and operate on string values or character data.

### SQL function

UPPER()

### Behaviour

Converts a string to upper case

LOWER()

Converts a string to lowercase

### Example:-

SELECT

name

FROM

movie

WHERE

### Output:-

#### name

→ Avengers ; Age of Ultron

Avengers ; Endgame

UPPER(name) LIKE UPPER("%avengers%");

### Note:-

Usually, UPPER() AND LOWER() functions can help us to perform case-insensitive searches.

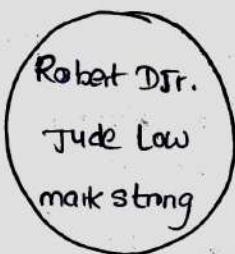
**Notes by Bhavana**

## SQL Set Operations:-

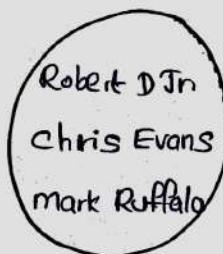
⇒ The SQL set operation is used to combine the two or more SQL queries.

⇒ Let us understand common set operations by performing operations on two sets.

- \* cast in "Sherlock Holmes" movie
- \* cast in "Avengers Endgame" movie



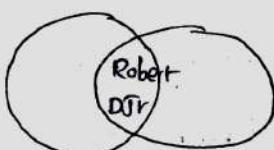
Sherlock Holmes



Avengers Endgame

### Common Set Operators:

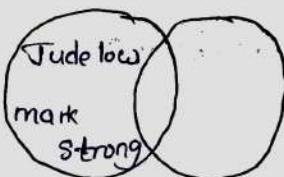
#### INTERSECT



Actors who acted in both Sherlock Holmes and Avengers Endgame.

Result :- Robert D Jr.

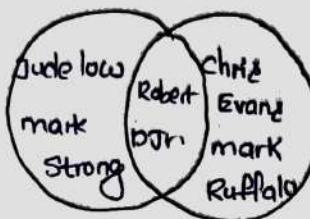
#### MINUS



Actors who acted in Sherlock Holmes and not in Avengers Endgame.

Result :- Jude Law, Mark Strong

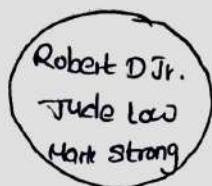
## UNION



Unique actors who acted in Sherlock Holmes or in Avenger's Endgame

Result :- Jude Law, Mark Strong, Robert D Jr., Chris Evans, Mark Ruffalo.

## UNION ALL



Doesn't eliminate duplicate results

Result : Jude Law, Mark strong, Robert D Jr., Robert D Jr., Robert D Jr., Chris Evans, Mark Ruffalo

## Applying Set Operations

⇒ We can apply these set operations on the two or more SQL queries to combine their results.

### Syntax:-

SELECT

C<sub>1</sub>, C<sub>2</sub>

FROM

table\_name - 1

SET\_OPERATOR

SELECT

C<sub>1</sub>, C<sub>2</sub>

FROM

table\_name - 2;

⇒ Basic rules when combining two SQL queries using set operators.

- Each SELECT statement must have the same number of columns.
- The columns must have similar data types
- The columns in each SELECT statement must be in the same order.

Example:- Get ids of actors who acted in both Sherlock Holmes (id = 6) and Avengers Endgame (id = 15)?

```
SELECT  
    actor_id  
FROM  
    cast  
WHERE  
    movie_id = 6
```

INTERSECT

```
SELECT  
    actor_id  
FROM  
    cast  
WHERE  
    movie_id = 15;
```

Output:-

actor\_id

6

## ORDER BY Clause in Set Operations

- ORDER BY clause can appear only once at the end of the query containing multiple SELECT statements.
- While using set operators, individual SELECT statements cannot have ORDER BY clause. Additionally, sorting can be done based on the columns that appear in the first SELECT query, for this reason, it is recommended to sort this kind of queries using column positions.

Example:-

Get distinct ids of actors who acted in Sherlock Holmes (id = 6) or Avengers Endgame (id = 15). Sort ids in the descending order.

```
SELECT
    actor_id
FROM
    cast
WHERE
    movie_id = 6
UNION
SELECT
    actor_id
FROM
    cast
WHERE
    movie_id = 15
ORDER BY
```

1 DESE :

## Pagination in Set Operations

Similar to ORDER BY clause, LIMIT and OFFSET clauses are used at the end of the list of queries.

### Examples:-

Get the first 5 id's of actors who acted in sherlock holmes (id=6) or Avengers Endgame (id=15). Sort id's in the descending order.

**SELECT**

actor\_id

**FROM**

cast

**WHERE**

movie\_id = 6

**UNION**

**SELECT**

actor\_id

**FROM**

cast

**WHERE**

movie\_id = 15

**ORDER BY**

1 DESC

**LIMIT**

5;

# Modelling Databases

SKBW  
Date \_\_\_\_\_  
Page \_\_\_\_\_

## \* Modelling Databases:-

### Core Concepts in ER Model :-

#### Entity:-

→ Real world object/concepts are called entities in ER Model.

Ex:- John, Emma, Apple, Google.

#### Attributes of an Entity:-

→ Properties of real world objects/ concepts are represented as attributes of an entity in ER Model

Ex: ① name: John      ② name: Emma  
      age: 29                  age: 25

#### key Attribute:-

→ The attribute that uniquely identifies each entity is called key attribute.

Ex:- Aadhar no: XXXX XXXX  
      age: 29                  name: Emma  
      name: John                age: 29

#### Entity Type:-

→ Entity Type is a collection of entities, where the same attributes (not values)

Ex:-

1   Aadhar no: xxxx	2   Aadhar no: xxx
1   named John	2   name: Emma
1   age: 29	2   age: 25 → person

## \* Relationships :-

Associating (Association) among the entities is called a relationship.

Example:

- \* person has a passport

- \* person can have many cars

- \* Each student can register for many courses, and a course can have many students.

## Types of Relationships:-

### ① One-to-one Relationship:-

→ An entity is related to only one entity, and vice versa.

→ An entity is related to only one entity, and vice versa.

Example:- \* A person can have only one passport

\* Similarly, a passport belongs to one and only one person

### ② One-to-Many Relationship:-

→ An entity is related to many other entities

Ex:- \* A person can have many cars. But a car belongs to only one person

### ③ Many-to-Many Relationship

→ Multiple entities are related to multiple entities.

Ex:-

- \* Each student can register to multiple courses.
- \* Similarly, each course is taken by multiple students.

### Cardinality Ratio:-

→ Cardinality in DBMS defines the maximum number of times an instance in one entity can relate to instances of another entity.

→ One-to-one (1:1)

→ One-to-many (1:m)

→ Many-to-one (M:1)

→ Many-to-many (m:n)

### \* Applying ER Model Concepts:

Let's build an ER model for a real-world scenario.

Customer → Order → OrderLine → Product

E-Commerce Applications involving Customer → Order → OrderLine → Product

In a typical e-commerce application, the following

\* Customer has only one cart & a cart belongs to only one customer.

\* Customer can add products to cart.

\* Cart contains multiple products.

\* Customer can save multiple addresses in the application for further use like shipping/delivery address.

→ Let's apply the concepts of ER Model to this e-commerce scenario.

### Entity types

- \* Customer
- \* product
- \* Cart
- \* Address

### Relationships

→ Relation Between Cart and Customer

- \* A customer has only one cart.
- \* A cart is related to only one customer.
- \* Hence, the relation between customer and cart entities is One-to-One relation.

→ Relation Between Cart and products

- \* A cart can have many products.
- \* A product can be in many carts.
- \* Therefore, the relation between cart and product is Many-to-Many Relation.

→ Relation Between Customer and Address

- \* A customer can have multiple addresses.
- \* An address is related to only one customer.
- \* Hence, the relation between customer and address is one-to-many relation.

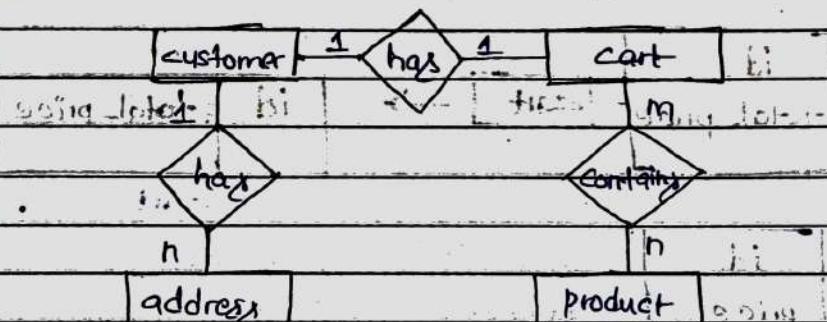
## Attributes:

Following are the attributes for the entity types in the e-commerce scenario:

- Here, attributes like id, product\_id etc. are key attributes as they uniquely identify each entity in the entity.

<u>Customer</u>	<u>product</u>	<u>address</u>	<u>cart</u>
<u>id</u>	<u>product id</u>	<u>id</u>	<u>cart id</u>
<u>name</u>	<u>price</u>	<u>pin code</u>	<u>total price</u>
<u>age</u>	<u>name</u>	<u>door no</u>	<u>;</u>
	<u>brand</u>	<u>city</u>	<u>state</u>
anushka	shopping	hi	category
			laptop
			on sale

## ER-Model of e-commerce application



## ER Model to Relational Database

## Entity Type to Table

Entity Types → Tables

Attributes → columns

key Attribute  $\rightarrow$  key, primary key.

**primary key :-** A minimal set of attributes (columns) in a table that uniquely identifies rows in a table.

→ In the following tables, all the ids are primary keys as they uniquely identify each row in the table.

id						
name	customer	→	id	name	age	
age						customer

id						
pin code	address	→	id	pin code	door no	city
door no						
city						address

id						
total price	cart	→	id	total price		
						cart

id						
price	product	→	id	name	price	brand
name						
brand						product
category						

## Relationships

→ Relationship Between customer and Address -

One-to-Many Relationship

- \* A customer can have multiple address.
- \* An address is related to only one customer.

→ We store the primary key of a customer in the address table to denote that the addresses are related to a particular customer thus establishing relationship.

→ This new column(s) in the table that refer to the primary key of another table is called Foreign key.

id	pin-code	door-no	...	customer_id
address				
1	560001	PK		
2	560002	FK		
3	560003	Unique FK		

Here, customer\_id is the foreign key that stores id (primary key) of customers.

→ Relation Between Cart and Customer - one-to-one Relationship

- \* A customer has only one cart.
- \* A cart is related to only one customer.

This is similar to one-to-many relationship. But, we need to ensure that only one Cart is associated to a customer.

<u>id</u>	<u>total_price</u>	<u>customer_id</u>

cart

→ Relation Between Cart and products - Many to Many Relationship.

- \* A cart can have many products.

- \* A product can be in many carts.

Here, we cannot store either the primary key of a product in the cart table or vice versa.

→ To store the relationship between the cart and product tables, we use a Junction Table.

The diagram illustrates the relationship between three tables: 'cart', 'product', and 'cart\_product'. The 'cart' table has columns 'id' and 'name'. The 'product' table has columns 'id' and 'name'. The 'cart\_product' junction table has columns 'cart\_id' and 'product\_id'. Foreign key arrows point from 'cart\_id' in 'cart\_product' to 'id' in 'cart', and from 'product\_id' to 'id' in 'product'.

<u>id</u>	<u>name</u>	<u>cart id</u>	<u>product id</u>	<u>id</u>	<u>name</u>
1	T-shirt	1	1	1	T-shirt
2	Jean	1	2	2	Jean
cart		1	3	3	mobile
		2	1		

cart\_product

FK to cart      FK to product

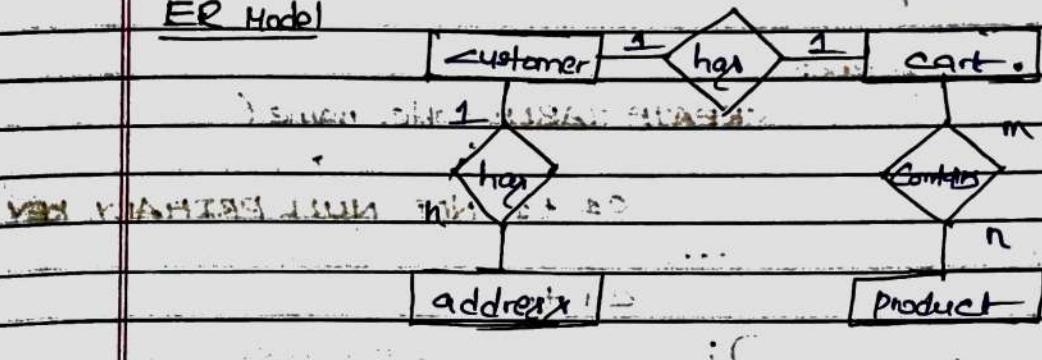
Note: what is the need of junction table?

We store the properties related to many-to-many relationship in the junction table. For example quantity of each product in the cart should be stored in the junction table 'cart\_product'.

## E-Commerce Usecase : ER Model to Relational Database :-

Following ER Model is represented as the below table in the relational database.

## ER Model



## Relational Database

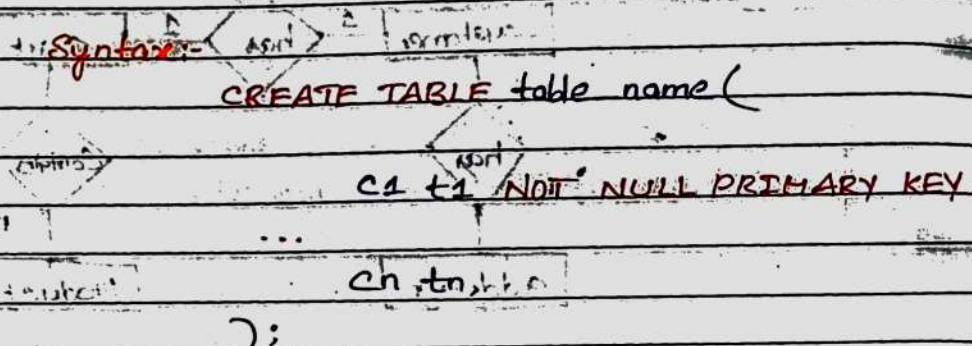
<u>id</u>	<u>totalPrice</u>	<u>Customer-id</u>	<u>..</u>	<u>..</u>	<u>id</u>	<u>name</u>	<u>price</u>	<u>brand</u>	<u>category</u>
1	100	1	2	3	1	MacBook Pro	1200	Apple	Laptops

id	cart_id	product_id	quantity
----	---------	------------	----------

## \* Creating a Relational Database

### primary key

following syntax creates a table with c1 as the primary key



### Foreign key

In case of foreign key, we just create a foreign key constraint.

#### Syntax:-

**CREATE TABLE table(**

**c1, t1 NOT NULL PRIMARY KEY,**

**FOREIGN KEY(c2) REFERENCES table(G)**  
**ON DELETE CASCADE**

**);**

#### Understanding

**FOREIGN KEY(c2) REFERENCES table1(c3)**

Above part of the foreign key constraint ensure that foreign key can only contain values that are in

the referenced primary key.

### ON DELETE CASCADE

Ensure that if a row in a table is deleted, then all its related rows in other tables will also be deleted.

Note:- To enable foreign key constraints in sqlite, use PRAGMA foreign\_keys = ON; by default it is enabled in our platform.

## Creating Tables in Relational Database:-

### Customer Table

```
CREATE TABLE customer(
```

```
    id INTEGER NOT NULL PRIMARY KEY,
```

```
    name VARCHAR(250),
```

```
    address TEXT,
```

```
);
```

### product Table

```
CREATE TABLE product(
```

```
    id INTEGER NOT NULL PRIMARY KEY,
```

```
    name VARCHAR(250),
```

```
    price INTEGER,
```

```
    brand VARCHAR(250),
```

```
    category VARCHAR(250),
```

```
    dimension TEXT,
```

```
    quantity INTEGER,
```

## Address Table

**CREATE TABLE address(**

**id INTEGER NOT NULL PRIMARY KEY,**

**pin\_code INTEGER,**

**door no VARCHAR(950),**

**city VARCHAR(950),**

**customer\_id INTEGER,**

**FOREIGN KEY(customer\_id) REFERENCES  
Customer(id) ON DELETE CASCADE**

## Cart Table

**CREATE TABLE cart(**

**id INTEGER NOT NULL PRIMARY KEY,**

**customer\_id INTEGER NOT NULL,**

**total price INTEGER,**

**FOREIGN KEY(customer\_id) REFERENCES  
Customer(id) ON DELETE CASCADE**

## Cart product Table (Junction Table)

**CREATE TABLE cart\_product(**

**id INTEGER NOT NULL PRIMARY KEY,**

**cart\_id INTEGER,**

**product\_id INTEGER,**

**quantity INTEGER,**

**FOREIGN KEY(cart\_id) REFERENCES cart(id)**

**ON DELETE CASCADE,**

**FOREIGN KEY(product\_id) REFERENCES  
product(id) ON DELETE CASCADE.**

2:

# JOins

SKIN

Date \_\_\_\_\_

Page \_\_\_\_\_

## Joins:-

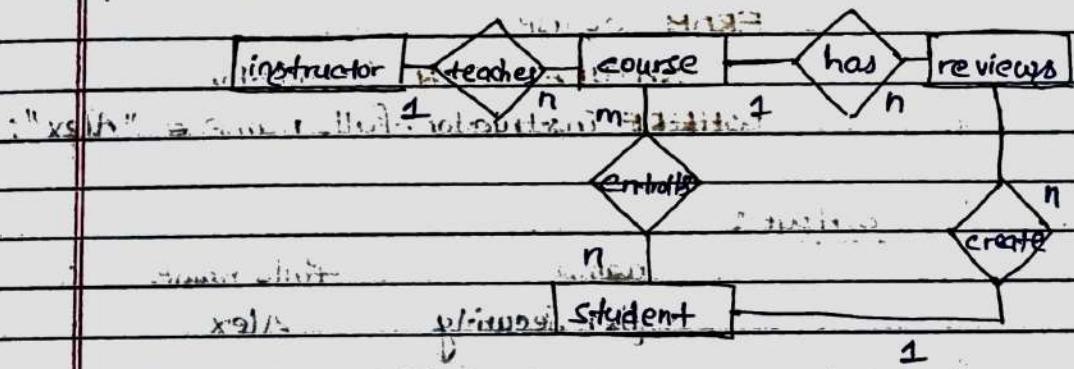
So far we have learnt to analyse the data that is present in a single table. But in the real-world scenarios, often, the data is distributed in multiple tables. To fetch meaningful insights, we have to bring the data together by 'combining' the tables.

→ We use JOIN clauses to combine rows from two or more tables, based on a related column between them. There are various types of joins, namely Natural join, Inner join, Full Join, Cross join, Left join, Right join.

→ Let's learn about them in detail using the following database, i.e., ~~student~~ students and ~~course~~ courses.

### Database:- ~~student~~ students and ~~course~~ courses

Here, the database stores the data of students, courses, course reviews, instructors, etc. in an e-learning platform.



Refer the tables in the code playground for a better understanding of the database.

## Natural Join

1. NATURAL JOIN combines the tables based on the common columns. It is a type of joining.

2. Syntax: `SELECT * FROM table1 NATURAL JOIN table2;`

3. `NATURAL JOIN table2;`

Example: The following query can be used to

1. fetch the details of courses that are being taught by "Alex".

Solving this problem involves querying on data stored in two tables, i.e., course & instructor. Both the tables have common column instructor\_id. Hence, we use natural join.

`SELECT course.name, course.id, instructor.instructor_id, instructor.full_name  
FROM course  
NATURAL JOIN instructor  
WHERE instructor.full_name = "Alex";`

Output:-

	name	full_name
1	Cyber Security	Alex

## INNER JOIN:-

It combines rows from both the tables if they meet a specified condition.

**Syntax:** - innermost part of query is it.

**SELECT \***

**FROM table1**

**INNER JOIN table2**

**ON table1.col = table2.col;**

D. older than

**Note:-** We can use any comparison operator in the condition.

### Example:-

Get the reviews of course "cyber security" (course with id=15).

**SELECT student.full\_name,**  
**review.content**

**FROM student** **INNER JOIN review**

**ON student.id = review.student\_id**  
**WHERE review.course\_id = 15;**

### Output:-

full_name	content	created_at
Ajay	Good explanation	2021-01-19
Ajay	cyber security is awesome	2021-01-20

## LEFT JOIN

In LEFT JOIN, for each row in the left table, matched rows from the right table are combined. If there is no match, NULL values are assigned to the right half of the rows in the temporary table.

Syntax:-

```
SELECT * FROM table-1
```

```
LEFT JOIN table-2
```

```
ON table1.c1 = table2.c2
```

### Example:-

Fetch the full name of students who ~~are~~ have not enrolled for any course.

```
SELECT student.full_name
```

```
FROM student
```

```
LEFT JOIN student_course
```

```
ON student.id = student.course.student_id
```

```
WHERE student.course_id IS NULL;
```

### Output:-

Full name
Afrin

## Joins on Multiple Tables :-

We can also perform join on a combined table.

Example :- fetch all the students who enrolled for the courses taught by the instructor "Arun" (id = 109).

```
SELECT T.course_name AS course_name, student.full_name
```

```
FROM COURSE
```

```
INNER JOIN student_course
```

```
ON course.id = student_course.course_id) AS T
```

```
INNER JOIN student
```

```
ON T.student_id = student.id
```

```
WHERE course.instructor_id = 109;
```

Output:

course_name	full_name
Machine Learning	Naren
Machine Learning	Sandhya

Note :- In the previous slide we studied,

Best practices

1. Use ATAS to name the combined table

2. Use alias : "table names" to refer for the columns in the combined table. MOST

## Using joins with other clauses

⇒ We can apply WHERE, ORDER BY, HAVING, GROUP BY, LIMIT and other clauses (which are used for retrieving data table as well).

### Example :-

Get the name of the student who scored highest in "Machine Learning" course.

```
SELECT student.full_name
FROM course
INNER JOIN student_course
ON course.id = student_course.course_id AS T
INNER JOIN student
ON T.student_id = student.id
WHERE course.name = "Machine Learning"
ORDER BY student_course.score DESC
LIMIT 1;
```

## Using joins with aggregations

⇒ We can apply aggregate functions such as SUM, Avg, COUNT, MAX, MIN and other to perform calculations on the temporary joined table as well.

### Example :- Get the highest score in each course.

```
SELECT
course.name AS course_name
MAX(score) AS highest_score
FROM
course LEFT JOIN student_course
ON course.id = student_course.course_id
GROUP BY
course.id;
```

## RIGHT JOIN :-

→ RIGHT JOIN or RIGHT OUTER JOIN is vice versa of LEFT JOIN i.e., in RIGHT JOIN, for each row in the right table, matched rows from the left table are combined. If there is no match, NULL values are assigned to the left half of the rows in the temporary table.

Syntax :-

```
SELECT *
FROM table1
RIGHT JOIN table2
ON table1.n1 = table2.n2;
```

Example :- perform RIGHT JOIN on course and instructor tables.

```
SELECT course.name,
instructor.full_name
FROM course
RIGHT JOIN instructor
ON course.instructor_id = instructor.instructor_id;
```

Note :- RIGHT JOIN is not supported in some DBMS (SQLite).

## FULL JOIN

⇒ FULL JOIN or FULL OUTER JOIN is the result of both RIGHT JOIN and LEFT JOIN.

Syntax:-

```
SELECT *
FROM table1
    FULL JOIN table2
```

```
ON table1.c1 = table2.c2 ;
```

Example :- perform ~~query~~ ~~performs~~ full join on course and instructor.

```
SELECT course.name,
```

```
instructor.full_name
```

```
FROM course
```

```
    FULL JOIN instructor
```

```
ON course.instructor_id = instructor.instructor_id ;
```

Note :-

FULL JOINS is not supported in some dbms  
(MySQL).

## CROSS JOIN

- In CROSS JOIN, each row from the first table is combined with all rows in the second table.
- Cross join is also called as CARTESIAN PRODUCT.

Syntax:-

```
SELECT * FROM table1
```

```
FROM table1
```

```
CROSS JOIN table2;
```

Example:- perform CROSS JOIN on course and instructor

```
SELECT course.name AS course-name,  
instructor.full-name AS instructor-name  
FROM course
```

```
CROSS JOIN instructor;
```

Output:-

	course-name	instructor-name
i.	Machine Learning	Alex
ii.	Machine Learning	Arun
iii.	Cyber Security	Alex

## SELF JOIN

⇒ we can combine a table with itself. This kind of join is called SELF JOIN.

### Syntax:-

SELECT t1.c1,

t2.c2

FROM table1 AS t1

JOIN table1 AS t2

ON t1.c1 = t2.c2;

Note:- we can use any JOIN clause in self-join

Example:- Get student pair who registered for common course.

SELECT sc1.student\_id AS student\_id1,

sc2.student\_id AS student\_id2, sc1.course

FROM

student\_course AS sc1

INNER JOIN student\_course sc2

ON sc1.course\_id = sc2.course\_id

WHERE

sc1.student\_id < sc2.student\_id;

### Output:-

student_id1	student_id2	course_id
-------------	-------------	-----------

1

3

11

## JOINS Summary :-

Join-type	use case
Natural join	joining based on common columns
Inner join	:joining based on a given condition
Left join	All rows from left table and matched rows from right table.
Right join	All rows from right table and matched rows from left table
full join	All rows from both the tables
Cross join	All possible combinations.

# Views and Subqueries

## ⇒ Views :-

→ A view can simply be considered a name to a SQL query.

### Create View

To create a view in the database, use the CREATE VIEW Statement.

#### Example :-

Create user\_base\_details view with id, name, age, gender and pincode.

```
CREATE VIEW user_base_details AS  
SELECT id, name, age, gender, pincode  
FROM user;
```

#### Note :-

→ In general, views are read only.

→ We cannot perform write operations like updating, deleting and inserting rows in the base table through views.

### Querying Using View

We can use its name instead of writing the original query to get the data..

```
SELECT *
```

```
FROM user_base_details;
```

## List All Available Views

→ In SQLite, to list all the available views, we use the following query.

Syntax:-

```
SELECT  
    name  
FROM  
    sqlite_master  
WHERE  
    TYPE = 'view';
```

Output:-      name

order-with-products  
user\_base\_details

## DELETE View:-

To remove a view from a database, use the DROP VIEW statement.

Syntax:-

```
DROP VIEW View-name;
```

Example:- Delete user\_base\_details view from the database

```
DROP VIEW user_base_details.
```

## Advantages:-

⇒ Views are used to write complex queries that involves multiple joins, group by, etc., and can be used whenever needed.

⇒ Restrict access to the data such that a user can only see limited data instead of a complete table. **Notes by Bhavana**

# Transactions and Indexes

## → Transactions :-

**transaction**

→ A transaction is a logical group of one or more SQL statements.

**SELECT ...**

**UPDATE ...**

**INSERT ...**

→ Transactions are used in various scenarios such as banking, ecommerce, social networks, booking tickets, etc.

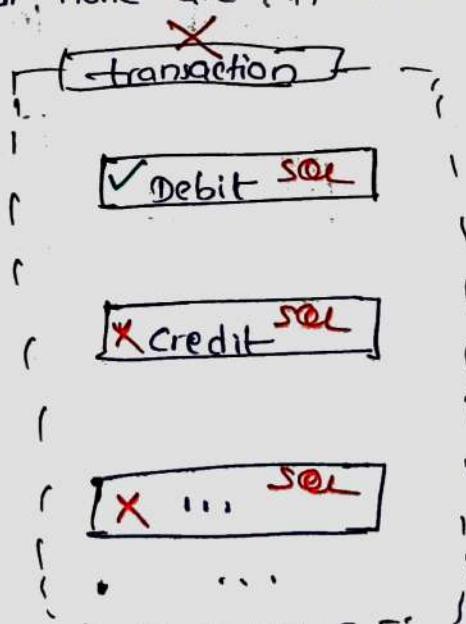
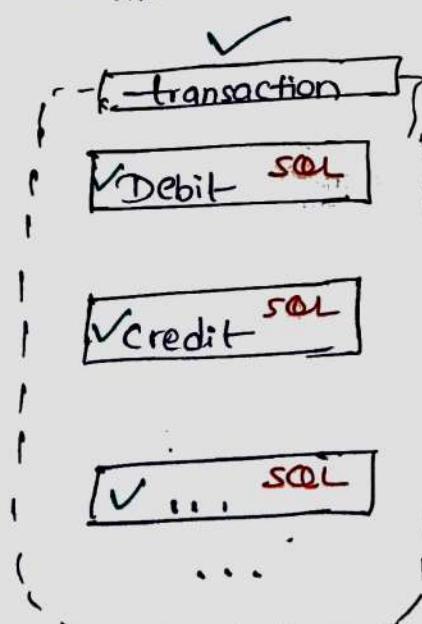
...

→ A transaction has four important properties.

(1) Atomicity (2) Consistency (3) Isolation (4) Durability

## Atomicity

Either all SQL statements or, none are applied to database



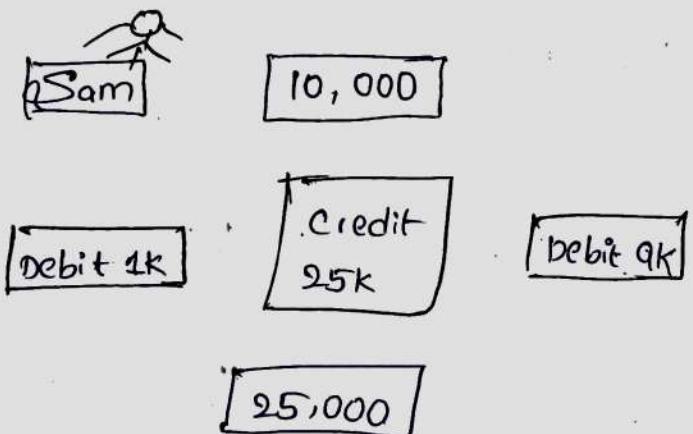
## Consistency:-

Transactions always leave the database in a consistent state

	Sam	David	
before	[10,000]	+ [5,000]	= 15,000
Success	[9,000]	+ [6,000]	= 15,000
Failure	[10,000]	+ [5,000]	= 15,000

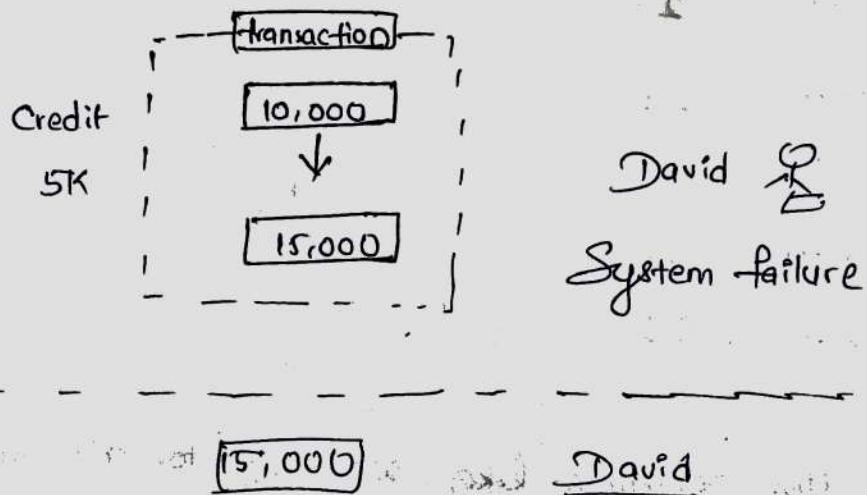
## Isolation:-

Multiple transactions can occur at the same time without adversely affecting the other.



## Durability:-

Changes of a successful transaction persist even after a system crash.



\* These four properties are commonly acronymed ACID

~~A~~tomicity Consistency ~~I~~solation Durable

## Indexes

A	ab...	02
	az...	23

B	ba...	24
	bz...	32

C	ca...	33
	c2...	43

In this Scenarios like, searching for a word in dictionary, we use index to easily search for the word. Similarly, in databases, we maintain indexes to speed up the search for data in a table.

---

---