The Complete SQL HandBook

Comprehensive Notes and Tips

HandWritten By
Bhavana

Contents

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.

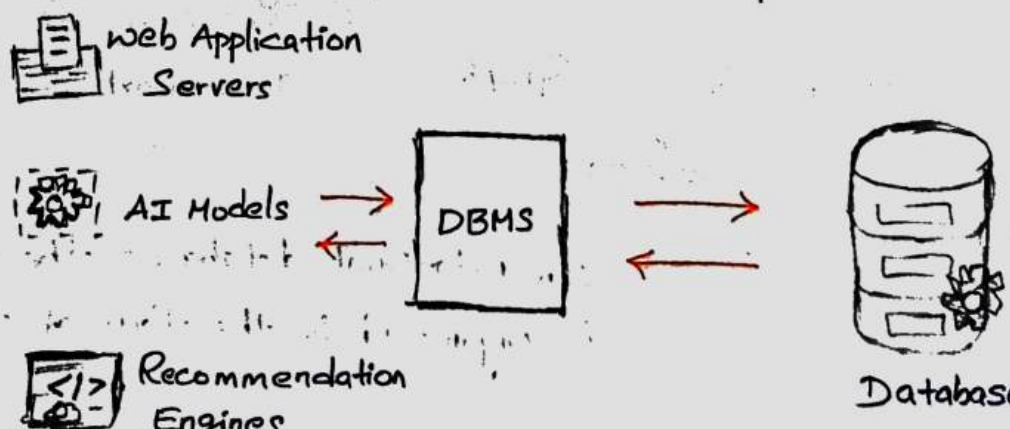
Examples :- 1. Messages & multimedia on whatsApp.
2. products and order on Amazon
3. Contact details in telephone directory, etc.

⇒ 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



- 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.

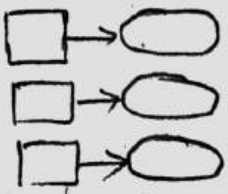
* performance:- Quickly accessible to all the clients (applications and stakeholders).

⇒ Types of Databases:-

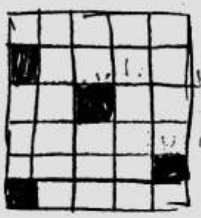
→ There are different types of databases based on how we organize the data.

Relational

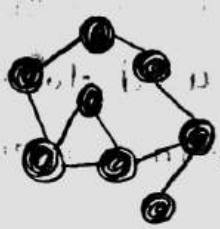
Analytical



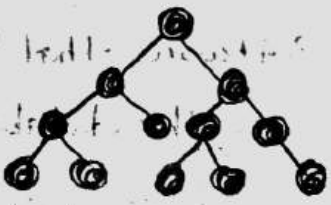
key value



Column family

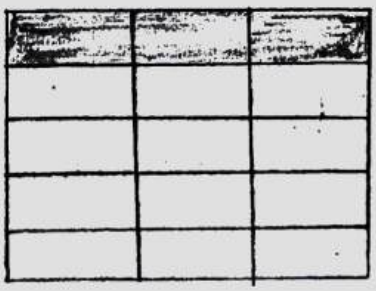


Graph



Document

- Relational Database:-



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

Non-Relational Database

→ Graph, key value, Column family, Document.

→ These four types are commonly reffered 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 respectvely. Datatypes that are supported in SQL are mentioned below.

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 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 specific 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

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 paranthesis () 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 output:-

If the datatype of the input value doesn't match with the datatype of column, SQLite 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 tournment. 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:-

name	age
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 retrive 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:-

name	age	score
Virat	32	50
Rakesh	39	35
Sai	47	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:-

name	age	score
Sai	47	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 1-

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 peopleprice. 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
Strawberry 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 <> "Food".
```

output:-

name	category	price	brand	rating
Blue shirt	clothing	750	Denim	3.8
Blue jeans	clothing	800	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 (>=), less than (<), less than or equal to (<=) 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, chocos...
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 equals 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".
Contains	WHERE name LIKE "%mobiles%"	Retrives 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;
```

Examples:-

1. 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 tabe

```
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	reamle	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	750	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 condi-
NOT	Used to negate a condion in the WHERE clause

Syntax:-

```
SELECT
  *
FROM
  table_name
WHRE
   condition1 ,
   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 determains the sequence of operations.

NOT
AND
OR

high ↓ low

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;

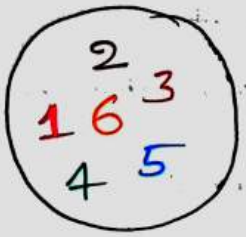
```
SELECT
  *
FROM
  product
WHERE
  (brand = "Redmi"
  AND rating > 4)
  OR brand = "Oneplus";
```

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 (v1, v2,..)



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", "Mufti", "Levi's", "Lee" or "Denim".

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

output :-

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

⇒ BETWEEN Operator :-

→ Retrieves 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	Ballow
Balloow	cow
cow	dog
Apple	Element
dog	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
    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
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 Jeans	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.

4 Aggregations and Group By :-

⇒ We perform aggregations is 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	count 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
   agregate_function (c1)
   agregate_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 wheather 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, data, time, datetime. etc., SQLite DBMS returns 0.0 and postgreSQL DBMS returns none.

⇒ Aggregate functions on strings and their outputs :-

Aggregate functions	Output
MIN, MAX	Based on texicographic 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

Aggregate functions	output
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_mateh_details;
```

output:-

player_name
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:-

name	total_score
David	105
Joseph	116
Lokesh	186
....	...

GROUP BY with WHERE

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

Syntax:-

```
SELECT
    c1,
    aggregate_function(c2)
FROM
    table_name
WHERE
        c3 = v1
GROUP BY c1;
```

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:-

name	half_centuries
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:-

name	half_centuries
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 different between collection and budget.

```
SELECT
    id, name, (collection_in_cr - budget_in_cr) as profit
FROM
    movie;
```

output:-

id	name	profit
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 || " _ " || gere 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) >= 50;
```

output:-

id	name	genre	budget_in_cr	collection_in_cr	rating	release
2	Inception	Action	16.0	83.68	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	1066.7	8.5	2010-06-25
...	...	...	...	...	...	...

⇒ Using Expressions in UPDATE Clause.

Example:- Scale docon rating 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:-

format	description	function
%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 move title and year for every movie from the database

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

output:-

name	strftime('%Y', release_date)
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;
```

Airthmetic functions:-

→ Airthmetic 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:-

```
FLOOR(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:-

```
CEIL(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	Behaviour
UPPER()	converts a string to upper case
LOWER()	Converts a string to lowercase

Example:-

```
SELECT
    name
FROM
    movie
WHERE
    UPPER(name) LIKE UPPER ("%avengers%");
```

output:-

name
Avengers : Age of ultron
Avengers : Endgame

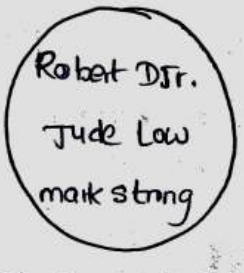
Note:- Usually, UPPER() AND LOWER() functions can help us to perform case-insensitive searches.

SQL Set Operations :-

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

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

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

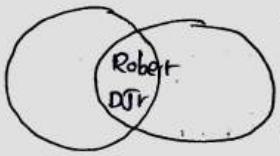


Sherlock Holmes.

Avengers Endgame

Common Set operators :

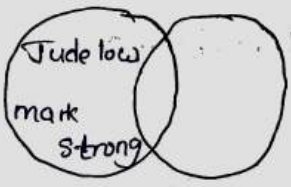
INTERSECT



Actors who acted in both Sherlock Homes and Avengers Endgame.

Result :- Robert DJr.

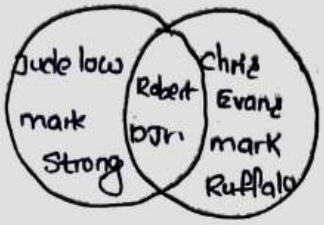
MINUS



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 Avengers Endgame

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

UNION ALL

Doesn't eliminate duplicate results

Result: Jude Law, Mark Strong, Robert DJr, 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
    c1, c2
FROM
    table_name_1
SET_OPERATOR
SELECT
    c1, c2
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 Holmesi (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 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 decending 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.

Example:-

Get the first 5 ids of actors who acted in sherlock Holmes (id = 6) or Avengers Endgame (id = 15). Sort ids in the decending order.

```
SELECT
    actor_id
FROM
    cast
WHERE
    movie_id = 6
UNION
SELECT
    actor_id
FROM
    cast
WHERE
    movie_id = 15
ORDER BY
    1 DESE
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
age: 29

② name: Emma
age: 25

key Attribute:-

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

Ex:- aadhar no: xxx
age: 29
name: John

aadhar no: xxx
name: Emma
age: 29

Entity Type:-

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

SKBW
Date
Page

Ex:-

aadhar_no: xxxx	aadhar_no: xxx	→ person
name: John	name: Emma	
age: 29	age: 25	

* 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.

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

SKBW
Date
Page

③ Many-to-Many Relationship

→ Multiple entities are related to multiple entities.

Ex:-

* Each student can register to multiple courses.
* Similarly each cource 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.

Cardinality Ratio
- → 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.

E-Commerce Application

In a typical e-commerce application,

* 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 selecting 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

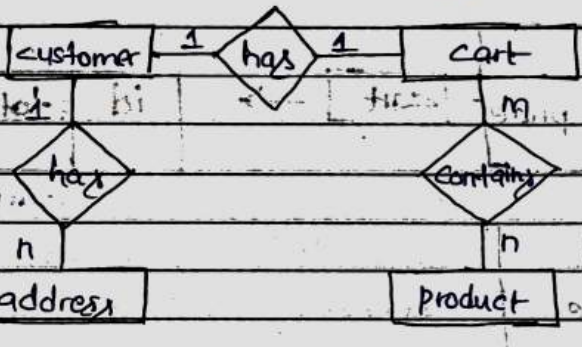
* A customer can have multiple addresses.
* An address is related to only one customer.
* Hence, the relation between customer and address is one-to-many relation.

Attributes:

Following are the attributes for the entity types in the e-commerce application. there, attributes like id, product id etc. are key attributes as they uniquely identify each entity in the entity.

Customer	product	address	cart
id	product id	id	cart id
name	price	pin code	total price
age	name	door no	
	brand	city	
	category		

ER-Model of e-commerce application



ER Model to Relational Database

Entity Type to Table

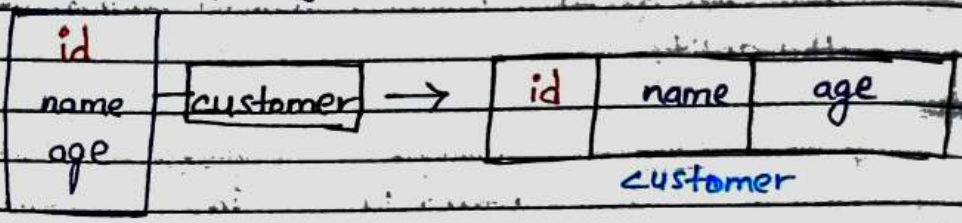
Entity Types ⟶ Tables

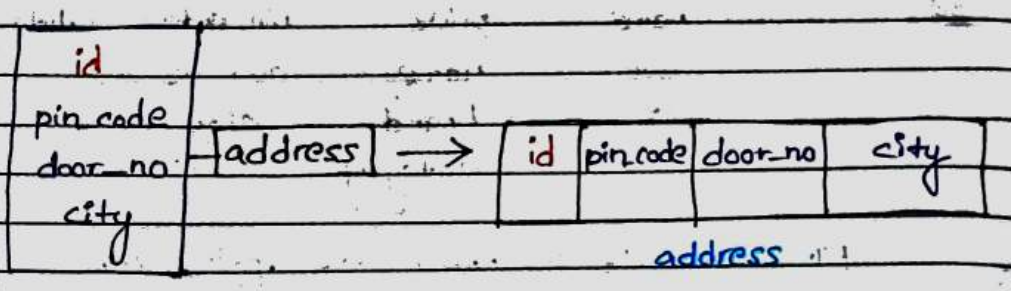
Attributes ⟶ columns

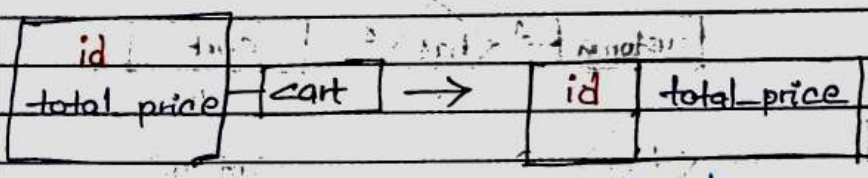
key Attribute ⟶ key primary key

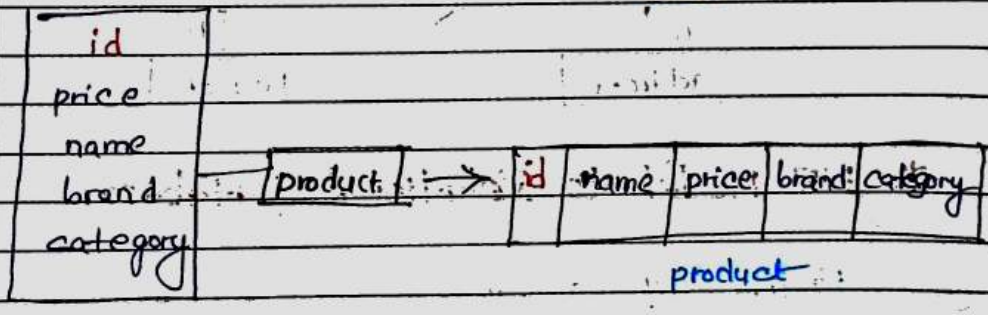
primary key :- A minimal set of attributes (columns) in a table that uniquely identifies rows in a table.

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









Notes by Bhavana

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 double that the addresses are related to a particular customer.
→ 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

PK
FK
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.

Notes by Bhavana

id	total_price	customer_id

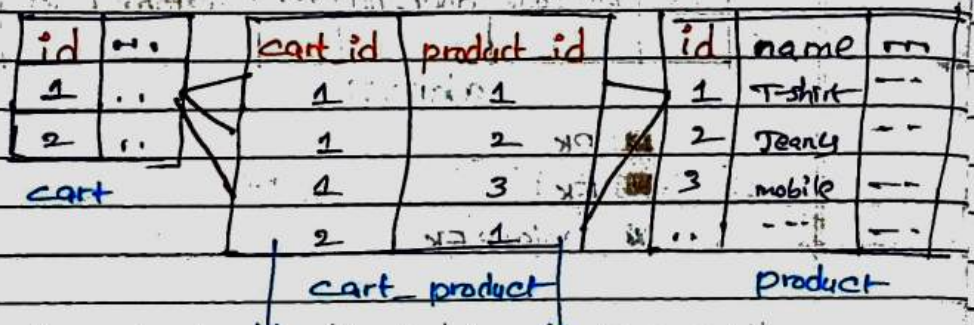
cart

→ 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.



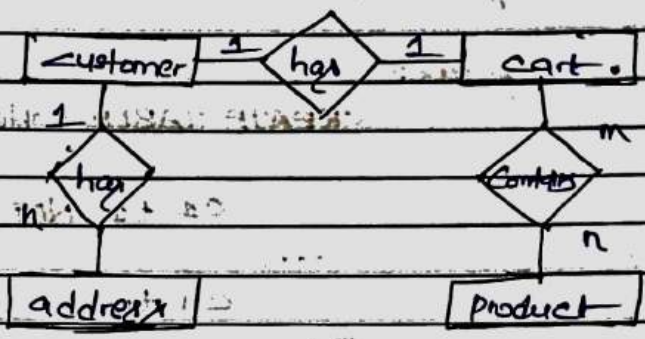
Note:

We store the properties related to a 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

id	name	age

customer

id	pincode	door no	city	customer_id

address

id	totalprice	customer_id

cart

id	name	price	brand	category

product

id	cart_id	product_id	quantity

cart product

SKBW
Date
Page

* Creating a Relational Database

primary key

Following syntax creates a table with c1 as the primary key.

Syntax:-

```
CREATE TABLE table name(
        c1 t1 NOT NULL PRIMARY KEY,
        ...
        cn tn
    );
```

Foreign key

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

Syntax:-

```
CREATE TABLE table2(
        c1 t1 NOT NULL PRIMARY KEY,
        c2, t2,
        FOREIGN KEY(c2) REFERENCES table1(c3)
        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 table1 is deleted, then all its related rows in table2 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),
    age INT
);
```

product Table

```
CREATE TABLE product(
    id INTEGER NOT NULL PRIMARY KEY,
    name VARCHAR(250),
    price INT,
    brand VARCHAR(250),
    category VARCHAR(250)
);
```

SKBW
Date
Page

Address Table

```
CREATE TABLE address(
    id INTEGER NOT NULL PRIMARY KEY,
    pin_code INTEGER,
    door_no VARCHAR(250),
    city VARCHAR(250),
    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,
    card_id INTEGER,
    product_id INTEGER,
    quantity INTEGER,
    FOREIGN KEY(card_id) REFERENCES cart(id)
    ON DELETE CASCADE,
    FOREIGN KEY(product_id) REFERENCES
    product(id) ON DELETE CASCADE
);
```

JOins

SKBW
Date
Page

* Joins:-

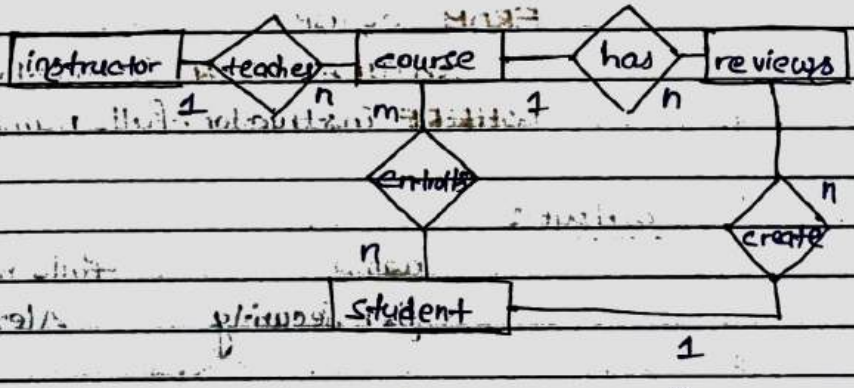
So far we have learnt to analyse the data that is present in a single table. But in the real-word scenarios, often, the data is distributed in multiple tables. To fetch meaningful insights, we have to bring the data together by 'combining' the tables.

→ We use JOIN clause 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.

Database:-

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



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

SKBW
Date
Page

Natural Join

NATURAL JOIN combines the tables based on the common columns.

Syntax:

```
SELECT *
FROM table1
NATURAL JOIN table2;
```

Example:

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,
       instructor.full_name
FROM course
NATURAL JOIN instructor
WHERE instructor.full_name = "Alex";
```

output:-

name	full_name
Cyber Security	Alex

SKBW
Date
Page

INNER JOIN :-

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

syntax :-

```
SELECT *
FROM table1
INNER JOIN table2
ON table1.c1 = table.c2;
```

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,
review.created_at
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

SKBW
Date
Page

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 table_1.c1 = table_2.c2;
```

Example:-

Fetch the full_name of students who 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

Notes by Bhavana

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 = 102).

```
SELECT T.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 = 102;
```

Output :-

course_name	full_name
machine learning	Varun
Machine learning	Sandya

Note :-

Best practices

1. Use AS/AS to name the combined table
2. Use alias table names to refer the columns in the combined table.

SKBW
Date
Page

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;
```

Notes by Bhavana

SKBW
Date
Page

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 :-

```
SELET *
FROM table1
  RIGHT JOIN table2
ON table1.c1 = table2.c2;
```

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 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 (sqlite).

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 JOIN

Syntax:-

```
SELECT *
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
Machine Learning	Alex
Machine Learning	Ann
Cyber Security	Alex

SKBW
Date
Page

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.c1;
```

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

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

```
SELECT sc1.student_id As student_id1,
    sc2.student_id As student_id2, sc1.cour
FROM
    student_course AS sc1
    INNER JOIN student_course sc2
    ON sc1.course_id = sc2.course_id
WHERE
    sc1.student_id < sc2.studen_id;
```

Output:-

student_id1	student_id2	course_id
1	3	11

SKBW
Date
Page

JOINS Summary :-

Join-type	use case
Natural join	joins based on common columns
Inner join	joins 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 :d, name, age, gender and pincode.

```
CREATE VIEW user_base_details AS
SELECT :d, 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 Availabe views

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

```
SELET
    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.
```

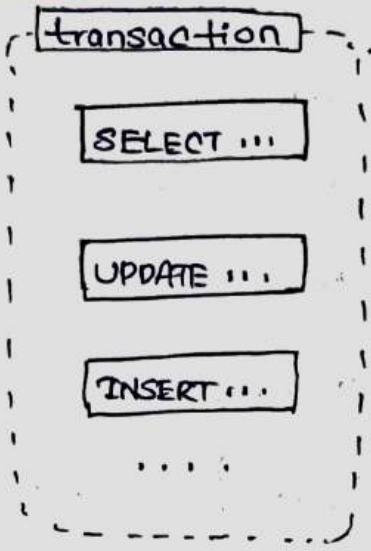
Advantages:-

⇒ 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.

Transactions and Indexes

→ Transactions :-



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

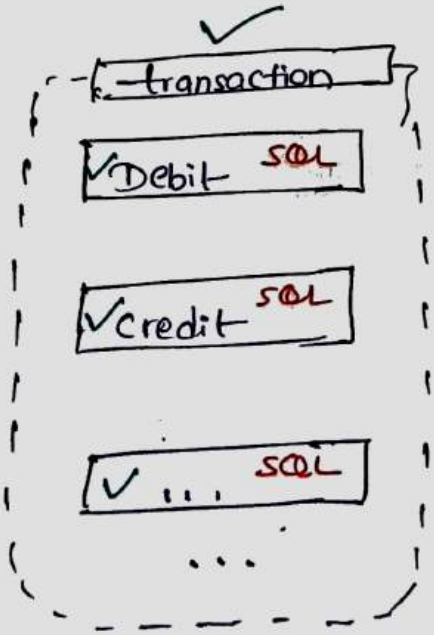
→ Transactions are used in various scenarios such as banking, ecommerce, social networks, booking tickets, etc.

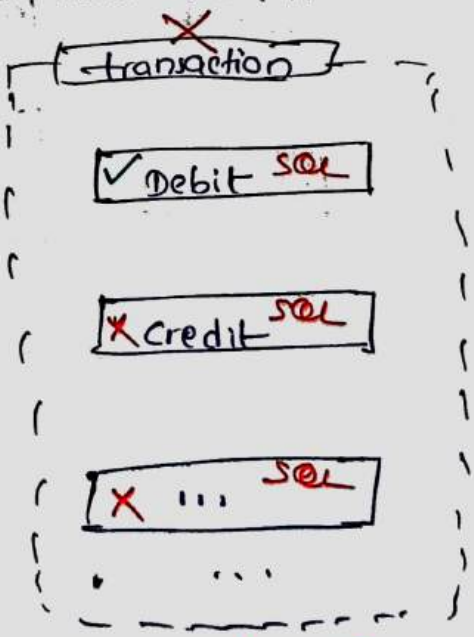
→ A transaction has four important properties.

(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	+	5000	= 15,000
Success	9,000	+	6,000	= 15,000
failure	10,000	+	5,000	= 15,000

Isolation:-

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

Sam

10,000

Debit 1k

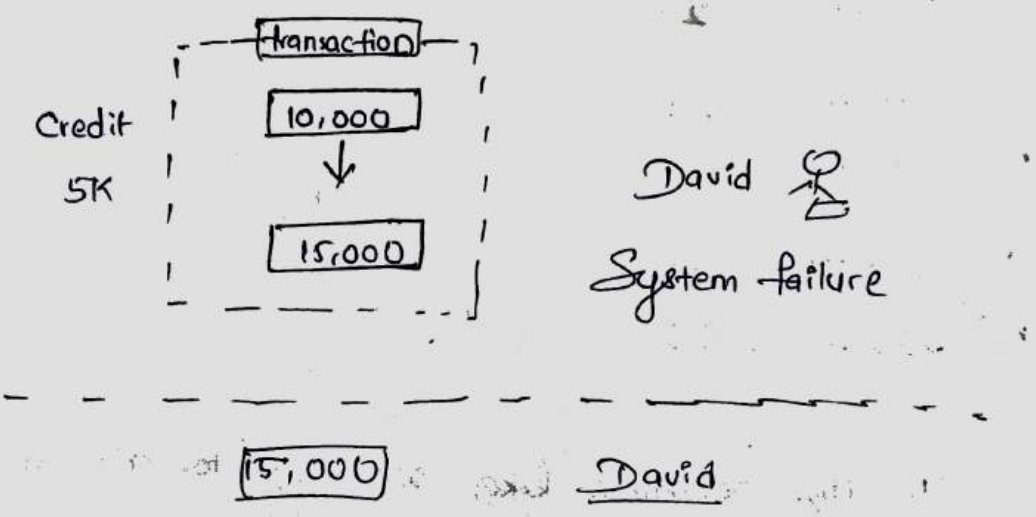
Credit 25k

Debit 9k

25,000

Durability :-

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



* These four properties are commonly acronymed ACID

Atomicity Consisteny Isolation Durable

Indexes

A	ab...	02
	az...	23
B	ba...	24
	bz...	32
C	ca...	33
	cz...	43

In this scenario 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.