Table of Contents

What are Databases?

A Database is a shared collection of logically related data and description of these data, designed to meet the information needs of an organization.

➢ Data Storage: A database is used to store large amounts of structured data, making it easily accessible, searchable, and retrievable.

- Data Analysis: A database can be used to perform complex data analysis, generate reports, and provide insights into the data.

- Record Keeping: A database is often used to keep track of important records, such as financial transactions, customer information, and inventory levels.

- Web Applications: Databases are an essential component of many web applications, providing dynamic content and user management.

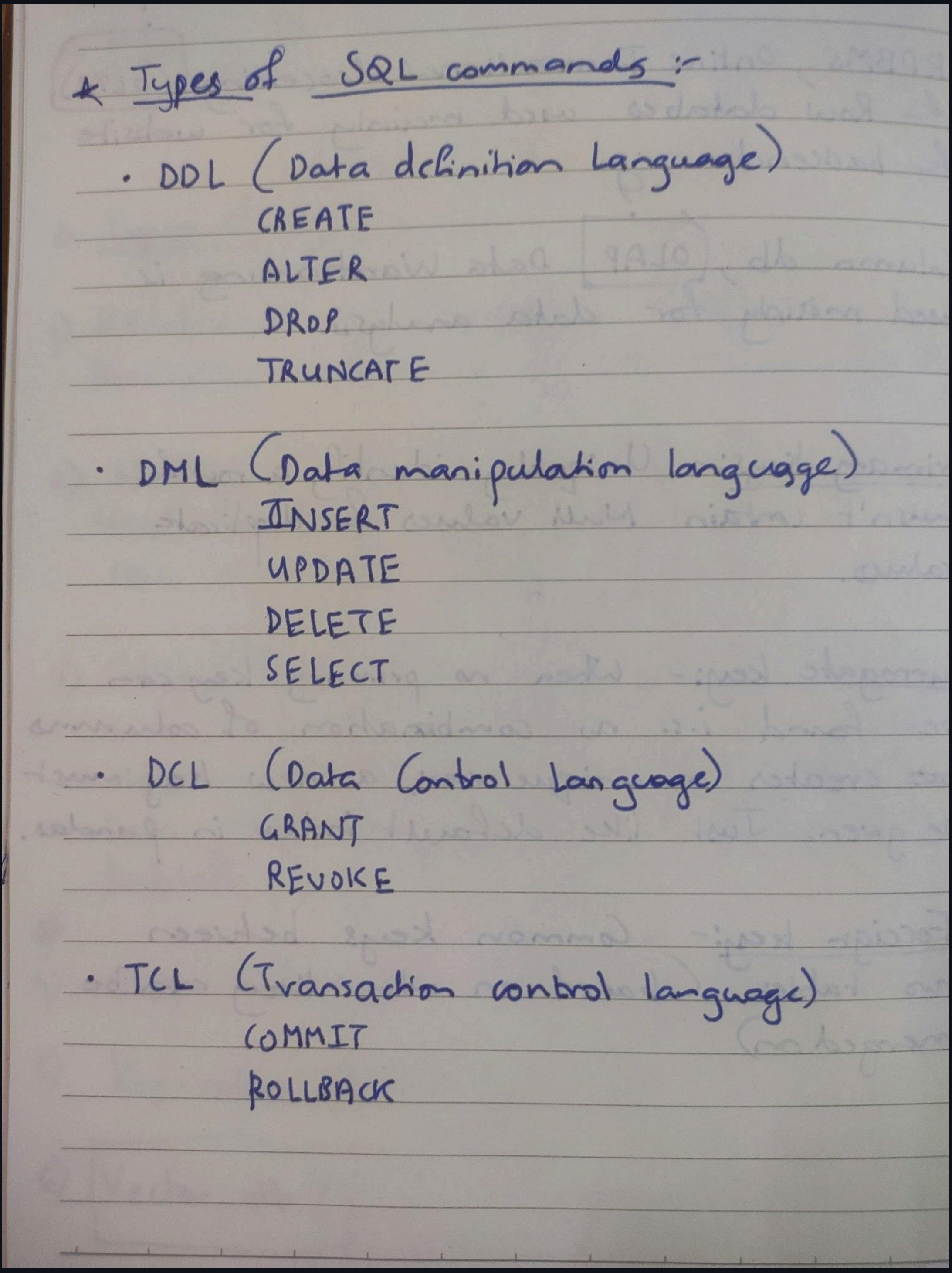
Properties of an Ideal Database:

1. Integrity
2. Availability
3. Security
4. Independent of Application
5. Concurrency

Types of Databases:

Note:- RDBMS, Online Transactional Processing (OLTP), Row databases are mainly used for websites and backend. Whereas, Column databases, OLAP, Data Warehousing is used mainly for data analysis.

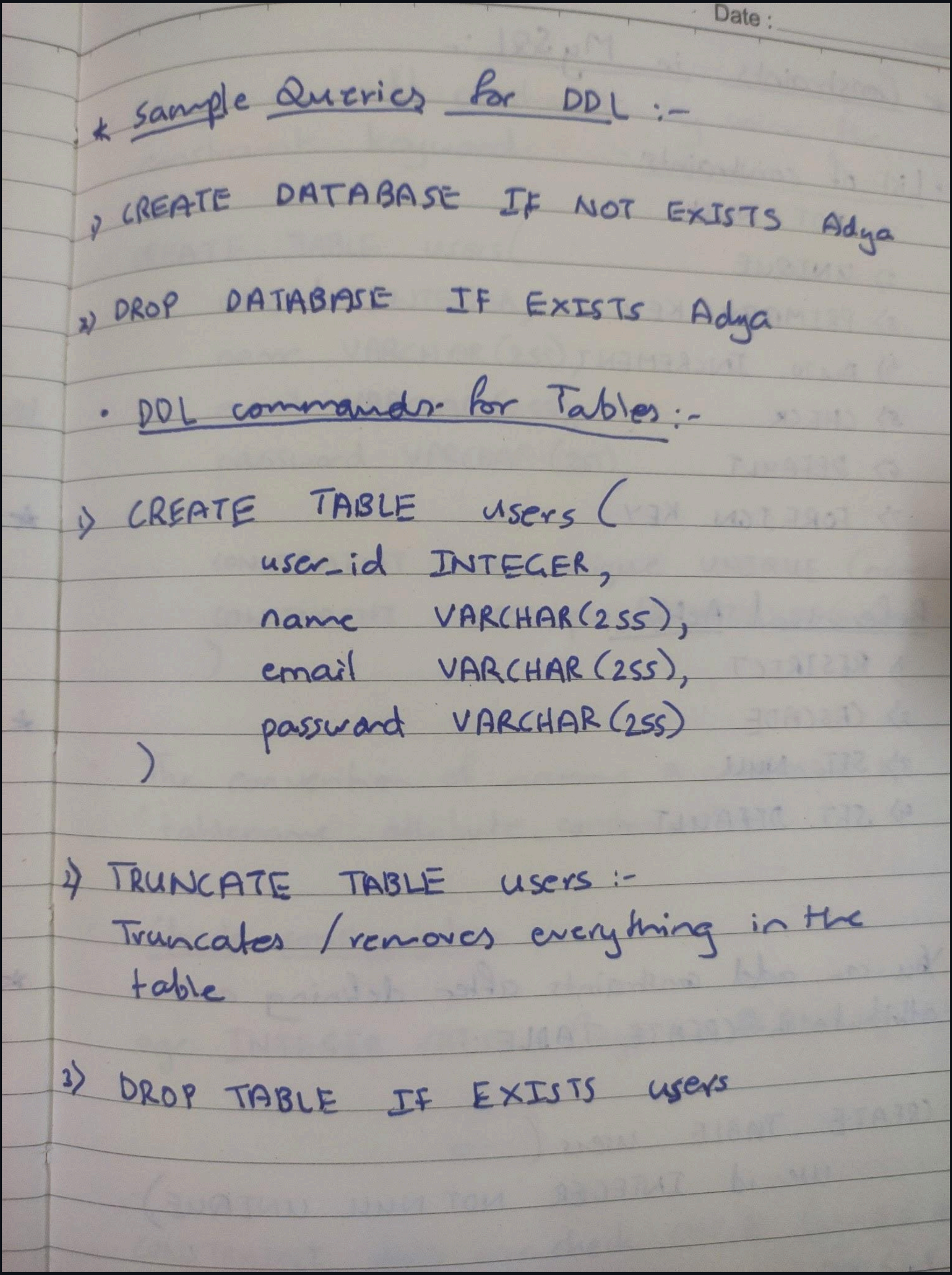
Types of SQL statements / languages:



* Types of SQL commands :-

- DDL (Data definition Language)
  - CREATE
  - ALTER
  - DROP
  - TRUNCATE
- DML (Data manipulation language)
  - INSERT
  - UPDATE
  - DELETE
  - SELECT
- DCL (Data Control Language)
  - GRANT
  - REVOKE
- TCL (Transaction control language)
  - COMMIT
  - ROLLBACK

Data Definition Language:



Date :

* Sample Queries for DDL :-

1) CREATE DATABASE IF NOT EXISTS Adya

2) DROP DATABASE IF EXISTS Adya

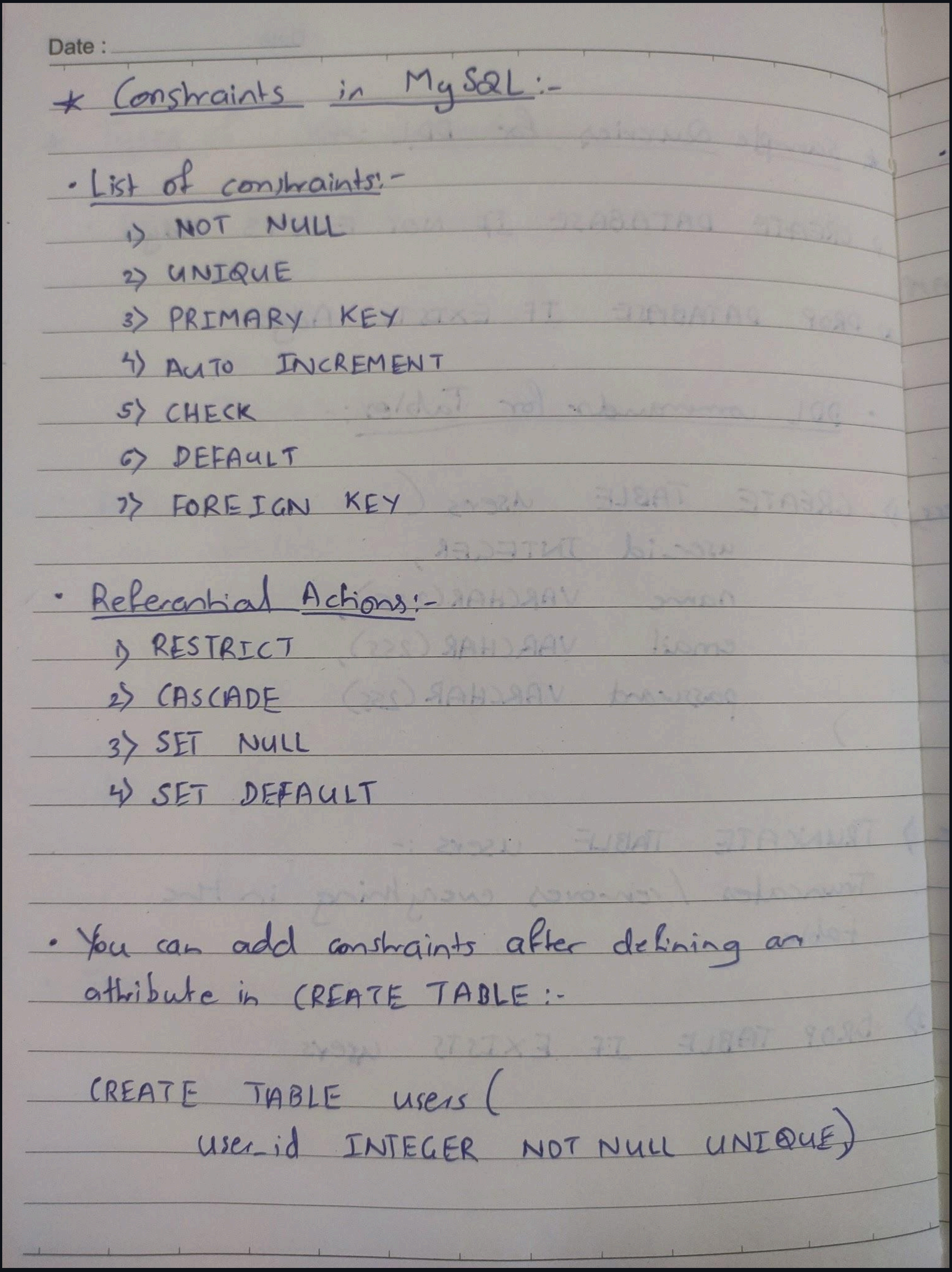
• DDL commands for Tables :-

```
1) CREATE TABLE users (
        user_id   INTEGER,
        name      VARCHAR(255),
        email     VARCHAR(255),
        password  VARCHAR(255)
   )
```

2) TRUNCATE TABLE users :-
Truncates / removes everything in the table

3) DROP TABLE IF EXISTS users

Constraints in MySQL:



Date :

★ Constraints in MySQL :-

- List of constraints :-
  1) NOT NULL
  2) UNIQUE
  3) PRIMARY KEY
  4) AUTO INCREMENT
  5) CHECK
  6) DEFAULT
  7) FOREIGN KEY

- Referential Actions :-
  1) RESTRICT
  2) CASCADE
  3) SET NULL
  4) SET DEFAULT

- You can add constraints after defining an attribute in CREATE TABLE :-

```
CREATE TABLE users (
        user_id INTEGER NOT NULL UNIQUE)
```

Date :

- OR you can add constraints by using the constraint keyword:

```
CREATE TABLE users(
      userid INTEGER,
      name VARCHAR(255),
      email VARCHAR(255),
      password VARCHAR (255),

      CONSTRAINT users_unique UNIQUE (name,email),
      CONSTRAINT users_pk PRIMARY KEY (name,email)
     )
```

- The convention of naming a constraint is 'tablename_attribute_constraint'

- <u>Check</u> <u>constraint</u>:-

```
age INTEGER CHECK (age > 6 ~~&~~ AND age < 25)
```

or

```
CONSTRAINT table_age-check CHECK (age > 6 AND
                                  age < 25)
```

Date :

- There are some built in functions that provide you default values. For example, in DATETIME CONSTRAINT:

```
order_date DATETIME DEFAULT CURRENT_TIMESTAMP
```

- <u>Foreign Key constraint</u>:-

Consider two tables namely orders & customers, where orders has order_id as primary key & cid, order_date as other attributes. customers has cid as primary key and name, email as other attributes.

Now, notice that in orders table, some others table's primary key is present as an attribute. orders has cid which is customer table's primary key. Thus, cid is foreign key in orders table. We can give that restraint by:

```
CONSTRAINT orders_fk FOREIGN KEY (cid) ——
— REFERENCES customers (cid)
```

Date :

Now, since we have added the foreign key restraint on cid in orders, we have basically made the orders table dependent on customers table. Whatever we do in the customers table will affect the orders table, but converse is NOT True.

Now, since orders depend on customers, we can define additional things that tell how orders table will be affected with changes to customers table. we do this by specifying referential actions (Restrict, cascade, set null, set default). Eg:-

```
CONSTRAINT orders_fk FOREIGN KEY (cid) REFERENCES—
— customers (cid) ↵
ON DELETE CASCADE ↵
ON UPDATE CASCADE
```

- Cascade works by imitating the changes made to the independent table (customers) in the dependent table (orders). For example, if a customer with specific cid is deleted from

Date :

customers table, then all instances of that customer in the order table will be deleted. If we change the cid of a customer in customers table, the cid will also be changed in orders.

- By default, the referential action is RESTRICT. So, it throws an error if we try to change customers because orders is dependent on it.

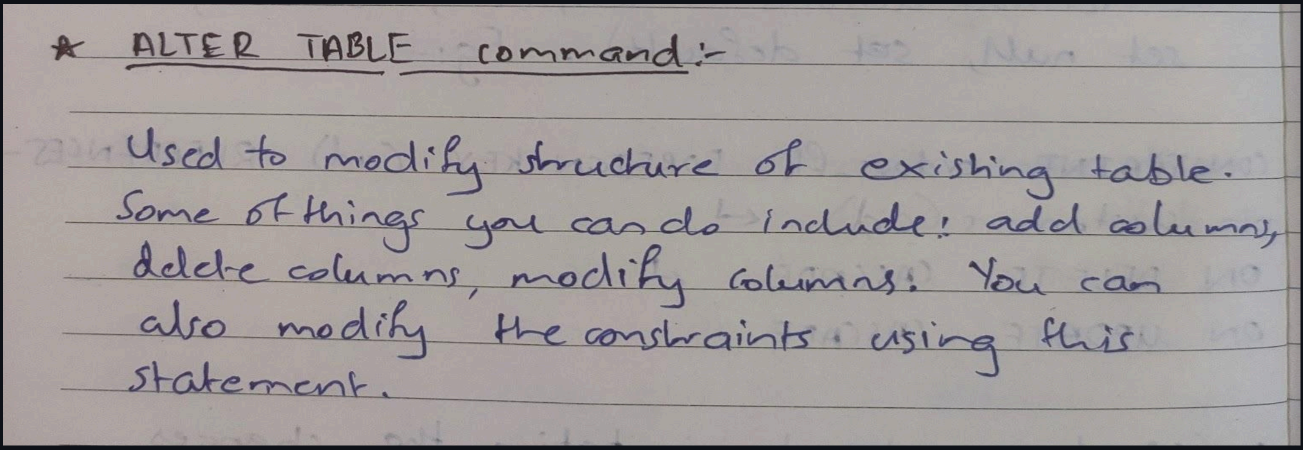
- SET NULL & SET DEFAULT work as you think they would.

★ <u>ALTER TABLE command</u>:-

Used to modify structure of existing table. Some of things you can do include: add columns, delete columns, modify columns. You can also modify the constraints using this statement.

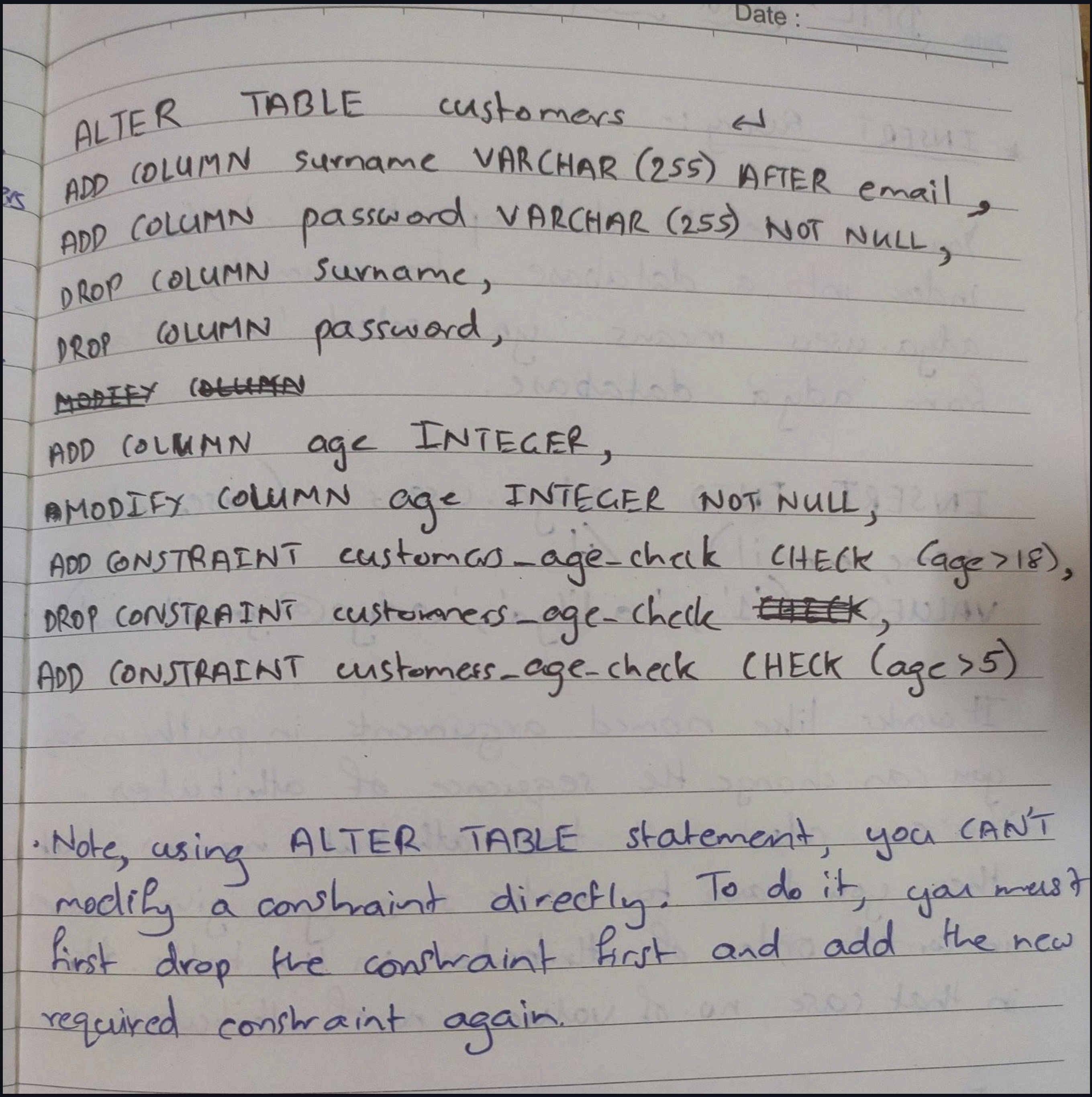
↵

ALTER table command:



★ ALTER TABLE command:-

Used to modify structure of existing table. Some of things you can do include: add columns, delete columns, modify columns. You can also modify the constraints using this statement.



Date :

```
ALTER TABLE customers ↵
ADD COLUMN surname VARCHAR (255) AFTER email,
ADD COLUMN password VARCHAR (255) NOT NULL,
DROP COLUMN surname,
DROP COLUMN password,
~~MODIFY COLUMN~~
ADD COLUMN age INTEGER,
MODIFY COLUMN age INTEGER NOT NULL,
ADD CONSTRAINT customers_age-check CHECK (age>18),
DROP CONSTRAINT customers_age-check ~~CHECK~~,
ADD CONSTRAINT customers_age-check CHECK (age>5)
```

• Note, using ALTER TABLE statement, you CAN'T modify a constraint directly. To do it, you must first drop the constraint first and add the new required constraint again.

Creating a table with same features, constraints, layout, attributes, etc of another table:

```
CREATE TABLE main.t1 LIKE main.t2
```

Data Manipulation Language:

INSERT Query, IN and NOT IN, UPDATE and DELETE, SQL functions:

Date : # DML - Data Manipulation Language

* **INSERT Query :-**

You can use the dot operator '.' to index into a database. For example, adya.users means you select 'users' table from 'adya' database.

```
INSERT INTO adya.users (user_id, —
— name, email) ↵
VALUES ('1', 'aditya', 'adya@gmail.com')
```

It works like named arguments in python. So, you can change the sequence of attributes. You can also omit the attribute names, but then you have to make sure you give values in correct order of attributes in the table. Also, in that case, no. of values = no. of attributes.

- Inserting multiple rows:-

```
INSERT INTO ~~[illegible]~~ adya.users VALUES
( - , - , - ),
( - , - , - ),
( - , - , - );
```

Date :

★ **IN and NOT IN:-**

```
brand_name IN ('samsung', 'apple')
```

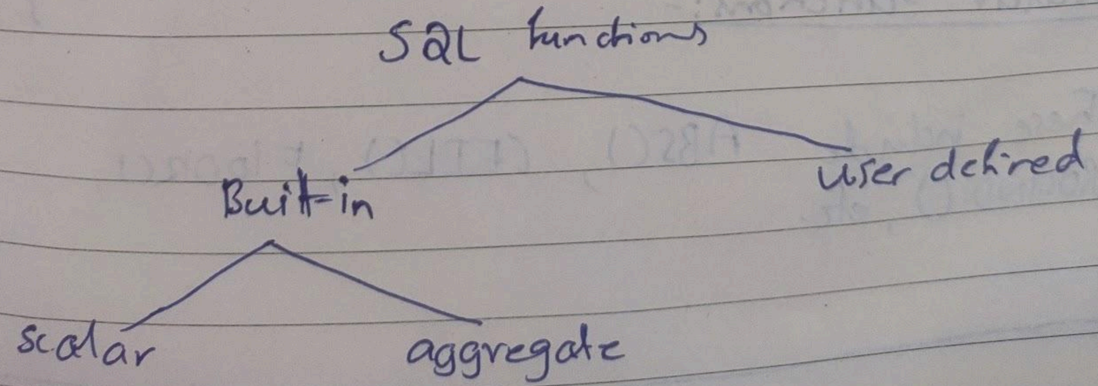
★ **UPDATE:-**

```
UPDATE aditya.smartphone SET
processor_brand = 'dimensity' WHERE
processor_brand = 'mediatek'
```

★ **DELETE:-**

```
DELETE FROM aditya.smarphones WHERE
price > 200000
```

★ **SQL Functions:-**



Date :

★ Aggregation functions :-

```
SELECT  MAX(price)  FROM  aditya.smartphones
-- will return max value of price
```

```
SELECT  AVG(rating)  FROM  aditya.smartphone
WHERE  brand_name = 'apple' ;
```

```
SELECT  COUNT (*)  FROM  aditya.smartphones
WHERE  brand_name  =  'samsung' ;
```

```
SELECT  COUNT(DISTINCT(brand_name))  FROM
aditya.smartphones. ;
```

★ Scalar functions :-

These include ABS(), CEIL(), FlOOR(), ROUND(), etc.

SELECT query for data retrieval:

```sql
SELECT * FROM adya.smartphones WHERE 1;

/* Here, the 'SELECT' query is used to choose columns specified after it.
The * symbol denotes all columns.
WHERE is a keyword used for filtering rows. So, WHERE 1 means select all rows
because 1 is always True.
By default, WHERE is 1, so you don't need to type it while selecting all the rows
in the table */
```

NOTE:- Every derived table (subquery that is a SELECT query) needs to have an alias after it.
All we need to do to resolve this error is to add an alias after the closing bracket.

Aliasing:

```sql
-- ALIASING

SELECT phone_model, os AS 'Operating system', battery_capacity AS 'mAh' FROM
adya.smartphones;

/* You can also give temporary alias names to tables and columns */

FROM adya.employess emp
```

Mathematical expressions and built in functions:

```sql
/* Mathematical expressions and built in functions */

SELECT phone_model, os,
SQRT(width*width + height*height) AS pixel_per_inches FROM adya.smartphones;
```

Constants:

```sql
SELECT phone_model, 'smartphone' AS 'type' FROM adya.smartphones;

/* This will create another column 'type' with all values as 'smartphone'
This is sort of a constant */
```

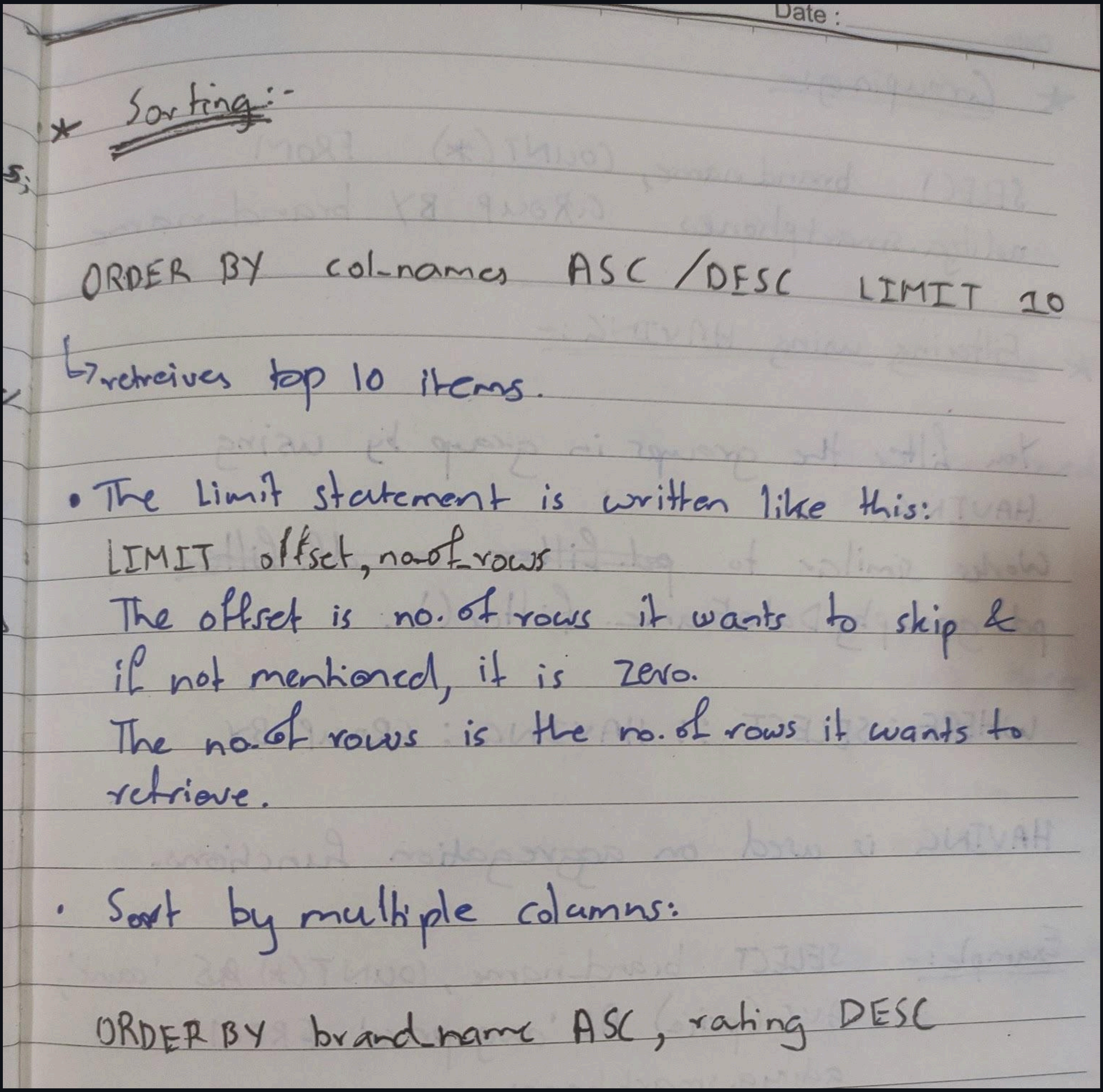
Unique values using DISTINCT:

```sql
SELECT DISTINCT(brand_name) FROM adya.smartphones;

/* You can also get unique combinations of columns */

SELECT DISTINCT brand_name,os FROM adya.smartphones;
```

Order of Query Execution:

FROM → JOIN → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT

Sorting using ORDER BY:



Date :

★ Sorting :-

ORDER BY col-names ASC /DESC LIMIT 10

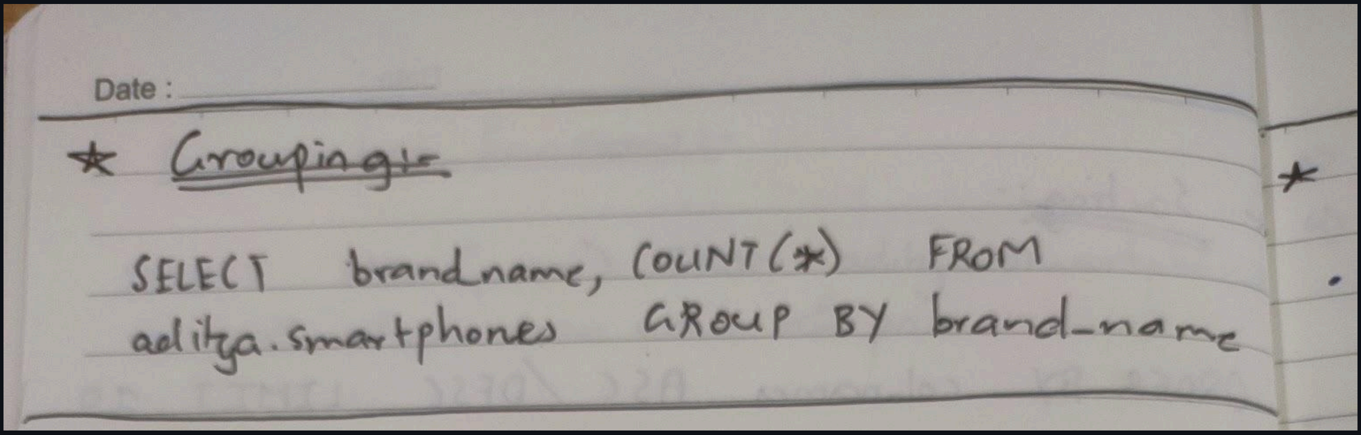
↳ retreives top 10 items.

- The limit statement is written like this:
  LIMIT offset, no.of_rows
  The offset is no. of rows it wants to skip & if not mentioned, it is zero.
  The no. of rows is the no. of rows it wants to retrieve.

- Sort by multiple columns:

ORDER BY brand_name ASC, rating DESC

Grouping using GROUP BY:

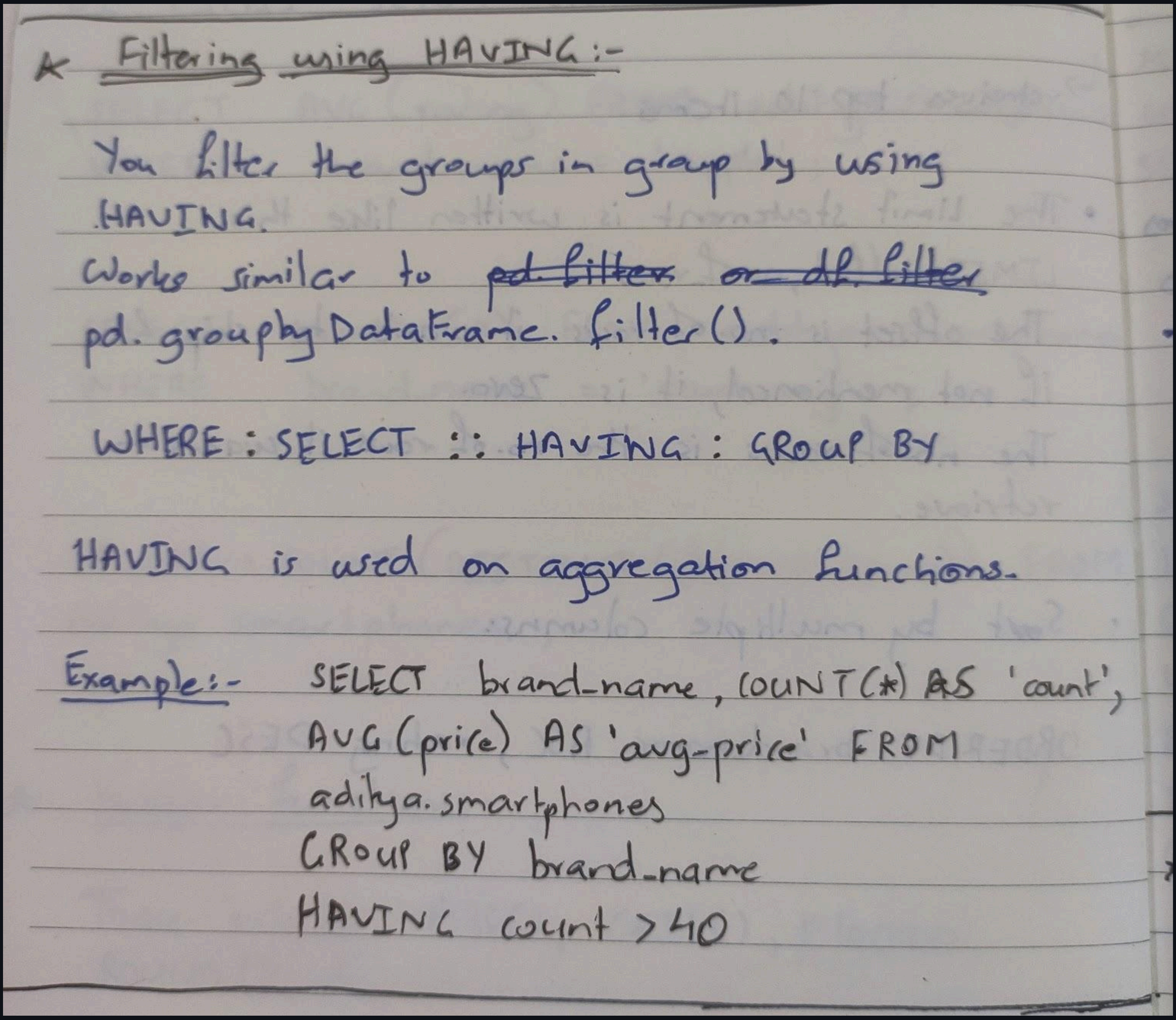


Date :

★ Grouping:-

SELECT brand_name, COUNT(*) FROM aditya.smartphones GROUP BY brand_name

Filtering groups using HAVING:



* Filtering using HAVING :-

You filter the groups in group by using HAVING.
Works similar to ~~pd.filter~~ ~~or df.filter~~
pd.groupby DataFrame. filter() .

WHERE : SELECT :: HAVING : GROUP BY

HAVING is used on aggregation functions.

Example:-

```
SELECT brand_name, COUNT(*) AS 'count',
AVG(price) AS 'avg_price' FROM
aditya.smartphones
GROUP BY brand_name
HAVING count > 40
```

ANY and ALL in SQL:

Works just like ‘any’ and ‘all’ in python.

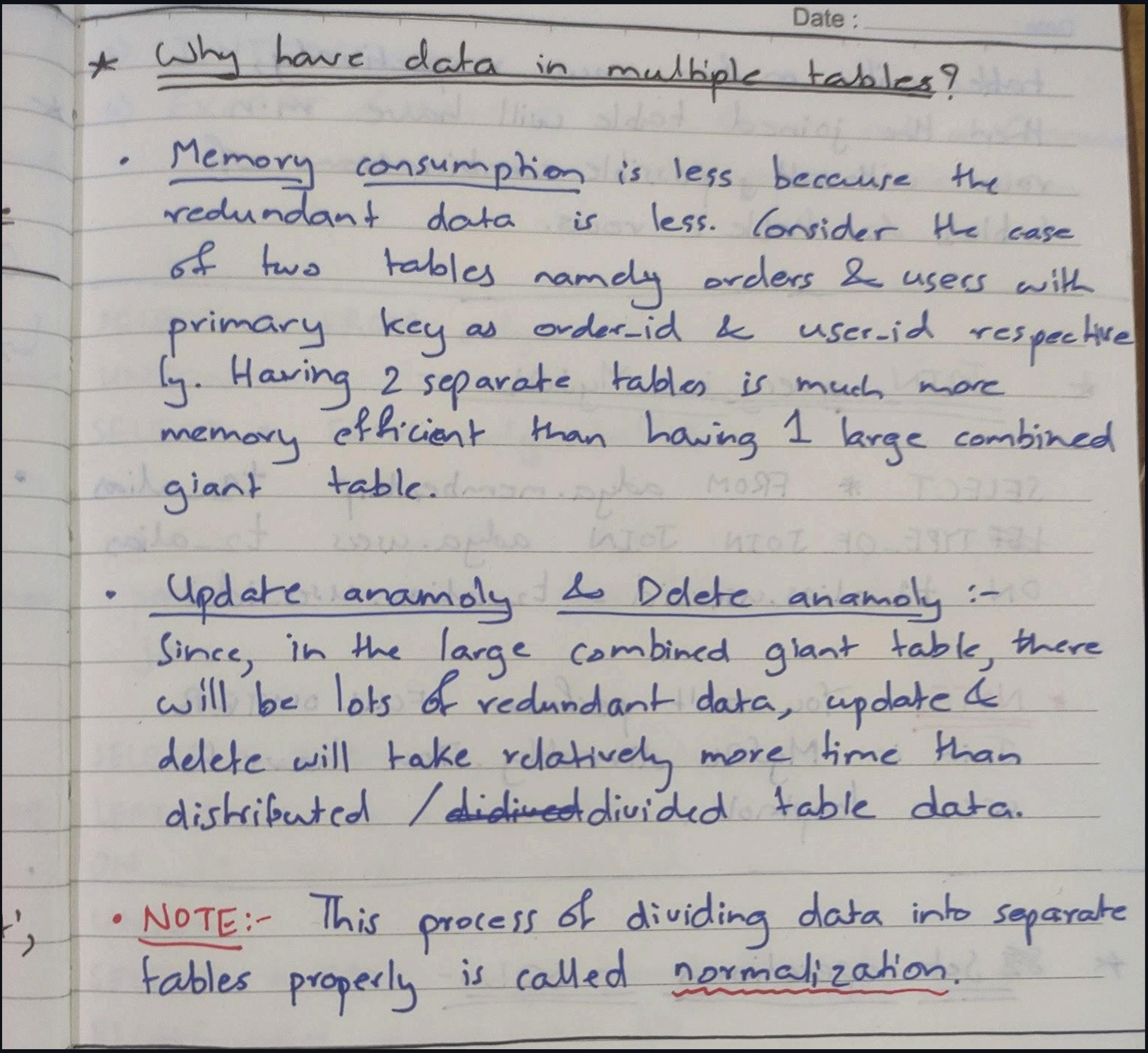
```
SELECT students FROM database WHERE marks > ALL(subquery that returns a column)
```

Extracting metadata of a table / database:

MySQL itself maintains a database that has metadata of all other databases and tables.

SQL Joins

Why have data in multiple tables?



Date :

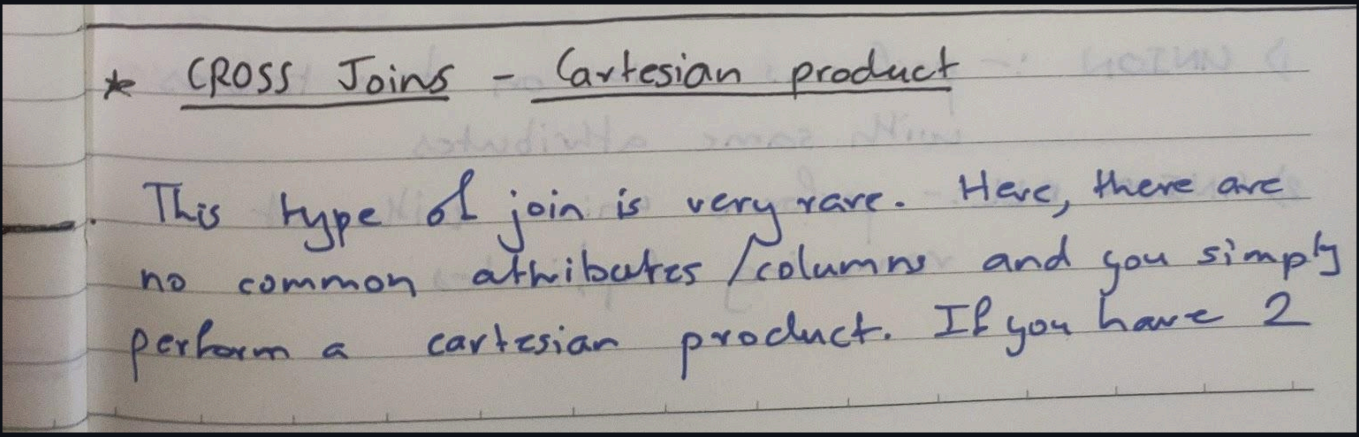
★ Why have data in multiple tables?

- Memory consumption is less because the redundant data is less. Consider the case of two tables namely orders & users with primary key as order_id & user_id respectively. Having 2 separate tables is much more memory efficient than having 1 large combined giant table.

- Update anamoly & Delete anamoly :- Since, in the large combined giant table, there will be lots of redundant data, update & delete will take relatively more time than distributed / ~~didived~~ divided table data.

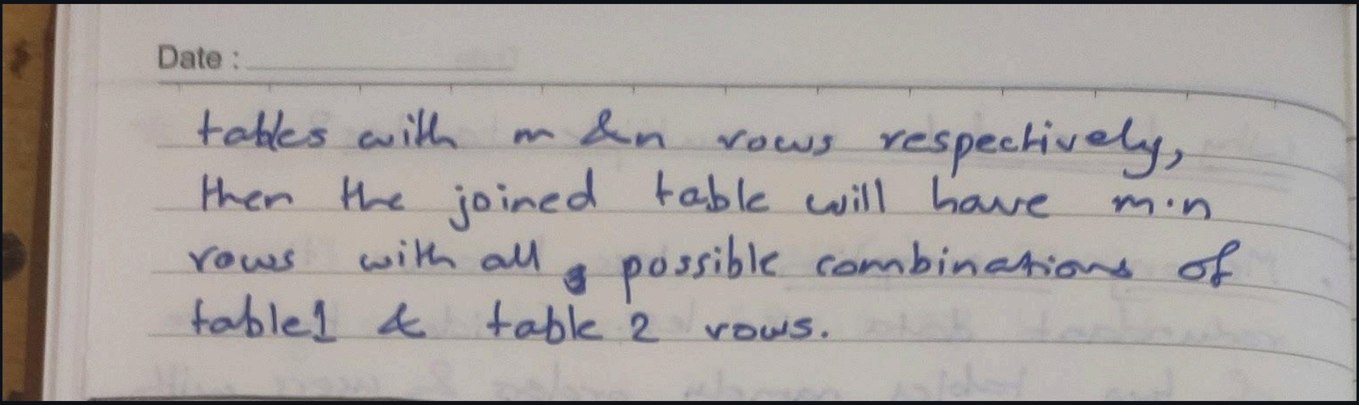
- NOTE :- This process of dividing data into separate tables properly is called normalization.

Cross Joins - Cartesian Product



★ CROSS Joins - Cartesian product

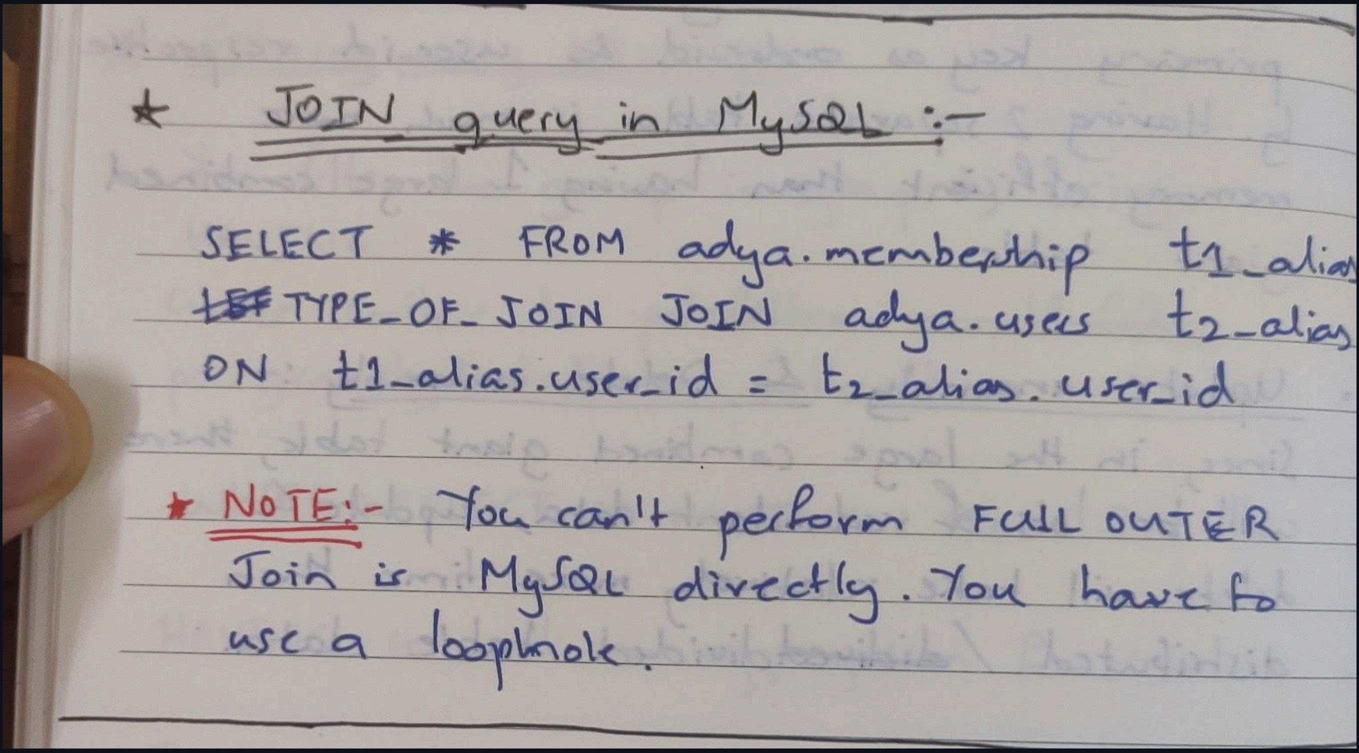
This type of join is very rare. Here, there are no common attributes /columns and you simply perform a cartesian product. If you have 2



Date :

tables with m & n rows respectively, then the joined table will have m·n rows with all ~~g~~ possible combinations of table1 & table 2 rows.

JOIN query in MySQL:

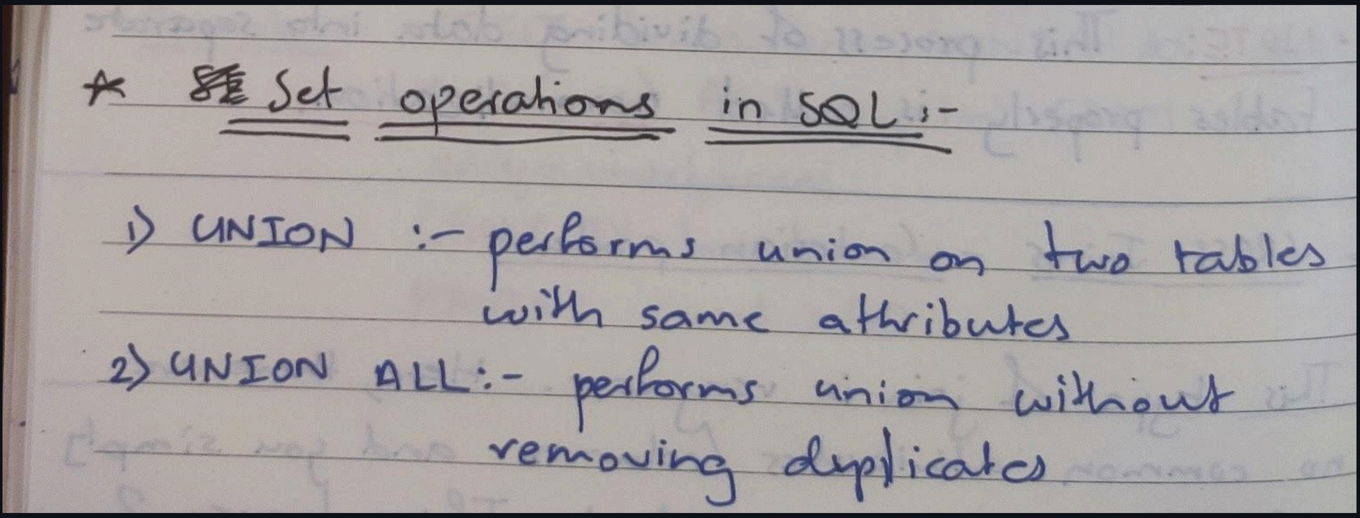


★ JOIN query in MySQL :-

```
SELECT * FROM adya.membership t1_alias
~~LEF~~ TYPE_OF_JOIN JOIN adya.users t2_alias
ON t1_alias.user_id = t2_alias.user_id
```

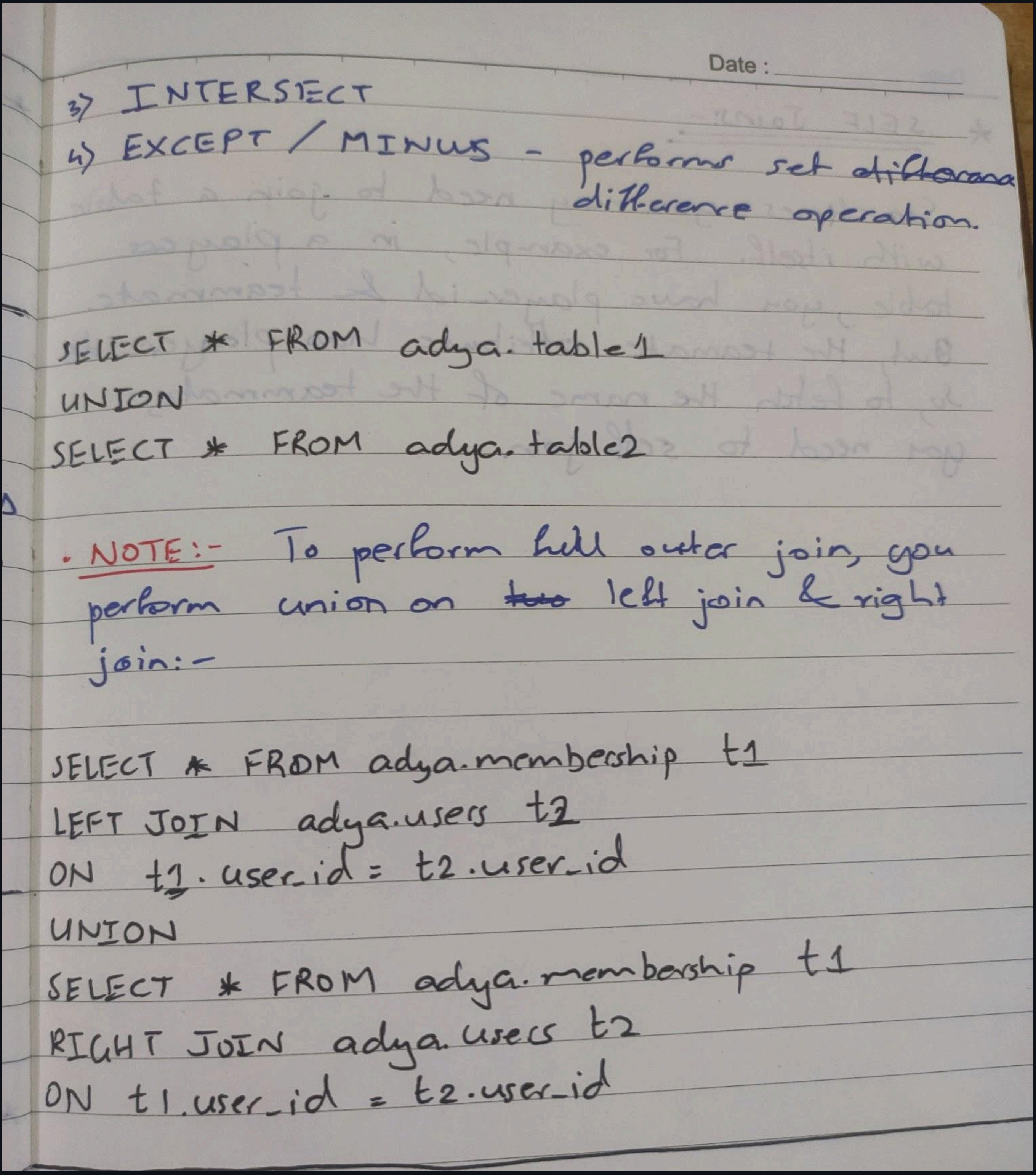
★ NOTE:- You can't perform FULL OUTER Join in MySQL directly. You have to use a loophole.

Set operations:



* ~~[illegible]~~ Set operations in SQL :-

1) UNION :- performs union on two tables with same attributes
2) UNION ALL :- performs union without removing duplicates



Date :

3> INTERSECT
4> EXCEPT / MINUS - performs set ~~difference~~ difference operation.

```
SELECT * FROM adya.table1
UNION
SELECT * FROM adya.table2
```

• NOTE:- To perform full outer join, you perform union on ~~two~~ left join & right join:-

```
SELECT * FROM adya.membership t1
LEFT JOIN adya.users t2
ON t1.user_id = t2.user_id
UNION
SELECT * FROM adya.membership t1
RIGHT JOIN adya.users t2
ON t1.user_id = t2.user_id
```

Join on multiple columns:

You can join on multiple columns using AND logic.

```
JOIN x ON y AND x ON z
```

Join multiple tables:

```
SELECT * FROM database.table
JOIN x
ON y
JOIN x
ON z
```

Non equi Joins:

In a non equi join, the join condition is based on operators other than equality. Specifically, the join condition can use operators such as greater than, less than, or not equal to, among others. Non equi joins are useful when you need to join tables on columns with similar but not identical data, or when you need to join tables based on a range of values rather than an exact match.
For example, if you have Name and player_id in a table, and you want to get all possible matches, then you could do not equal to join, `ON t1.player_id != t2.player_id`

Natural Joins:

A natural join is a type of join in SQL where two tables are joined based on the columns with the same name and data type. In other words, it is a join where the join condition is implicitly based on the column names that exist in both tables, and it eliminates the duplicate columns from the result set.
If you perform natural joins even if there are no matching column names, a simple cross join i.e. cartesian product will be performed.

Anti Joins:

An excluding join, also known as an anti-join, is a type of join operation in SQL that returns only the rows from one table that do not have any matching rows in another table. In other words, it returns the rows that are not included in the result set of an inner join between the two tables.

- Left excluding join - same as A - B set operation
- Right excluding join - same as B - A set operation
- Full excluding join - same as (A-B)U(B-A) or (AUB - A_intersect_B)

You can find more here: https://learnsql.com/blog/sql-joins/

SQL Subqueries:

Types of subqueries based on return data:

1. **Scalar values:** When the subquery returns a single value
2. **Row data:** when subquery returns a single column with multiple rows
3. **Table data:** when the subquery returns a table

Types of subqueries based on working of subquery:

1. **Independent:** when the inner query is independent of the outer query
2. **Correlated**: when the inner query is dependent on the outer query and can't be executed independently

Where can you use subqueries?

- INSERT
- UPDATE
- DELETE
- SELECT
- WHERE
- FROM
- HAVING

USE statement:

You can do `USE db_name;` to use that db going on forwards in the SQL script.

WITH statement:

Note:- In MySQL, you can't use LIMIT in certain versions, so you can use WITH statement to create a temporary db and you can store the subquery which uses LIMIT in that temp db.

```
WITH top_directors AS (
      SELECT director FROM movies GROUP BY director ORDER BY SUM(gross) DESC LIMIT 3
)
```

```
SELECT * FROM movies WHERE director IN (
      SELECT * FROM top_directors
)
```

Check multiple memberships using IN:

You can check multiple columns in a list of lists basically. It's basically equivalent to item in list in python.

```
WHERE (name, brand) IN (subquery that returns a n by 2 table)
```

Query optimization:-

Conventionally, searching operation is faster than sorting, right?
In MySQL, the columns are generally indexed (new concept), so sorting is as fast as searching.

Correlated subquery:

In this type, the inner query requires the outer query to be executed.
For example, you have to find all movies that have a rating higher than the average rating of movies in the same genre. Now, the thing here is that the genre changes for every movie, so you can't hardcore the average value. So, you calculate the average for each row (genre) in a loop of sorts.

```
SELECT * FROM movies m1
WHERE score > (
      SELECT AVG(score) FROM movies m2 WHERE m2.genre = m1.genre
)
```

Another example: Find the favourite food of each customer. So, each customer orders a certain food each time they order. For each customer, you have to find the most ordered food.

```
WITH favourites AS (
      SELECT COUNT(*) AS 'count', us.user_id, food.f_name FROM orders od
      JOIN order_details odd ON od.order_id = odd.order_id
      JOIN users us ON us.user_id = od.user_id
      JOIN food ON food.f_id = odd.f_id
      GROUP BY us.user_id, food.f_name
)
SELECT * FROM favourites f1
WHERE count = (
      SELECT MAX(count) FROM favourites f2 WHERE f1.user_id = f2.user_id
)
```

```sql
/* Notice that the query is calculating the MAX(count) for each and every row,
even if the user_id has repeated. So, instead, we store the info about max count
in another table and use that table to retrieve the favourite food. */

WITH favourites AS (
      SELECT COUNT(*) AS 'count', us.user_id, food.f_name FROM orders od
      JOIN order_details odd ON od.order_id = odd.order_id
      JOIN users us ON us.user_id = od.user_id
      JOIN food ON food.f_id = odd.f_id
      GROUP BY us.user_id, food.f_name
),

fav_count AS (
      SELECT user_id, MAX(count) AS 'num' FROM favourites GROUP BY user_id
)

SELECT * FROM favourites f1
WHERE count = (
      SELECT num FROM fav_count f2 WHERE f1.user_id = f2.user_id
)
```

Inserting a table in another table using subqueries:

```sql
-- You don't write VALUES when inserting like this

INSERT INTO zomato.loyal_users
(user_id, name)
SELECT us.user_id, us.name
FROM orders odd
JOIN users us ON us.user_id = odd.user_id
GROUP BY user_id, name
HAVING COUNT(*) > 3
```

Subqueries cannot manipulate their results internally:

Therefore ORDER BY clause cannot be added to a subquery. You can use an ORDER BY clause in the main SELECT statement (outer query) which will be the last clause.

Updating a table using subquery:

```sql
UPDATE loyal_users
SET money = (
      SELECT SUM(amount) FROM orders WHERE loyal_users.user_id = orders.user_id
)
```

Window Functions:

Window functions in SQL are a type of analytical function that perform calculations across a set of rows that are related to the current row, called a "window". A window function calculates a value for each row in the result set based on a subset of the rows that are defined by a window specification.

The window specification is defined using the OVER() clause in SQL, which specifies the partitioning and ordering of the rows that the window function will operate on. The partitioning divides the rows into groups based on a specific column or expression, while the ordering defines the order in which the rows are processed within each group.

```sql
-- Find all the students who have marks higher than the avg marks of their
respective branch

SELECT * FROM (
      SELECT *,AVG(marks) OVER(PARTITION BY branch) AS 'branch_avg' FROM marks
) t1
WHERE t1.marks > t1.branch_avg;
```

```sql
-- Rank each student in their respective branch based on marks

SELECT * ,
RANK() OVER(PARTITION BY branch ORDER BY marks DESC),
DENSE_RANK() OVER(PARTITION BY branch ORDER BY marks DESC)
FROM marks;
```

```sql
-- Find 2 most paying customers of each month

SELECT *
FROM (
      SELECT name, MONTHNAME(date) AS 'month', SUM(amount) AS 'total',
    DENSE_RANK() OVER(partition by MONTHNAME(date) order by SUM(amount)) AS 'rank'
    FROM orders od
      JOIN users us ON od.user_id = us.user_id
      GROUP BY name, month
) t1
WHERE t1.rank IN (1,2)
ORDER BY month
```

FRAMES:

A frame in a window function is a subset of rows within the partition that determines the scope of the window function calculation. The frame is defined using a combination of two clauses in the window function: ROWS and BETWEEN.

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

The BETWEEN clause specifies the boundaries of the frame.

Examples:

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

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

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

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

NOTE:- You define these clauses in the OVER() clause at the end.

Aliasing window clauses:

```
WINDOW window_name AS (
PARTITION BY temp ORDER BY temp ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING
)
```

Calculating Quantiles using WITHIN:

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

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

```
--Q1. Find the median marks of all the students

SELECT *,
PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY marks) OVER() AS 'median_marks'
-- Here, the 0.5 means 50th percentile or median.
-- To calculate the percentile, the data must be sorted by some parameter (marks).
-- That's why we use WITHIN GROUP(ORDER BY marks)
FROM marks
```

```
-- Q2. Find branch wise median of student marks

SELECT *,
PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY marks) OVER(PARTITION BY branch) AS
'median_marks'
-- Here, percentile_cont is used. That 'cont' means continuous. It interpolates
the marks and returns a value that may not be originally present in the marks
column.
-- percentile_disc is discrete and returns a value that is present in the original
dataset
FROM marks
```

```
-- Q3. Removing outliers using IQR (Inter quartile range)

SELECT * FROM (
      SELECT *,
      PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY marks) OVER(PARTITION BY
branch) AS 'quartile1',
      PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY marks) OVER(PARTITION BY
branch) AS 'quartile3'
      FROM marks
) t1
WHERE t1.marks > t1.quartile1 - (1.5*(t1.quartile3 - t1.quartile1)) AND
            t1.marks < t1.quartile3 + (1.5*(t1.quartile3 - t1.quartile1))
-- Here, we are selecting the data which excludes the outliers.
```

Segmentation using NTILE():

Segmentation using NTILE is a technique in SQL for dividing a dataset into equal-sized groups based on some criteria or conditions, and then performing calculations or analysis on each group separately using window functions.
It essentially creates equal sized bins or buckets.

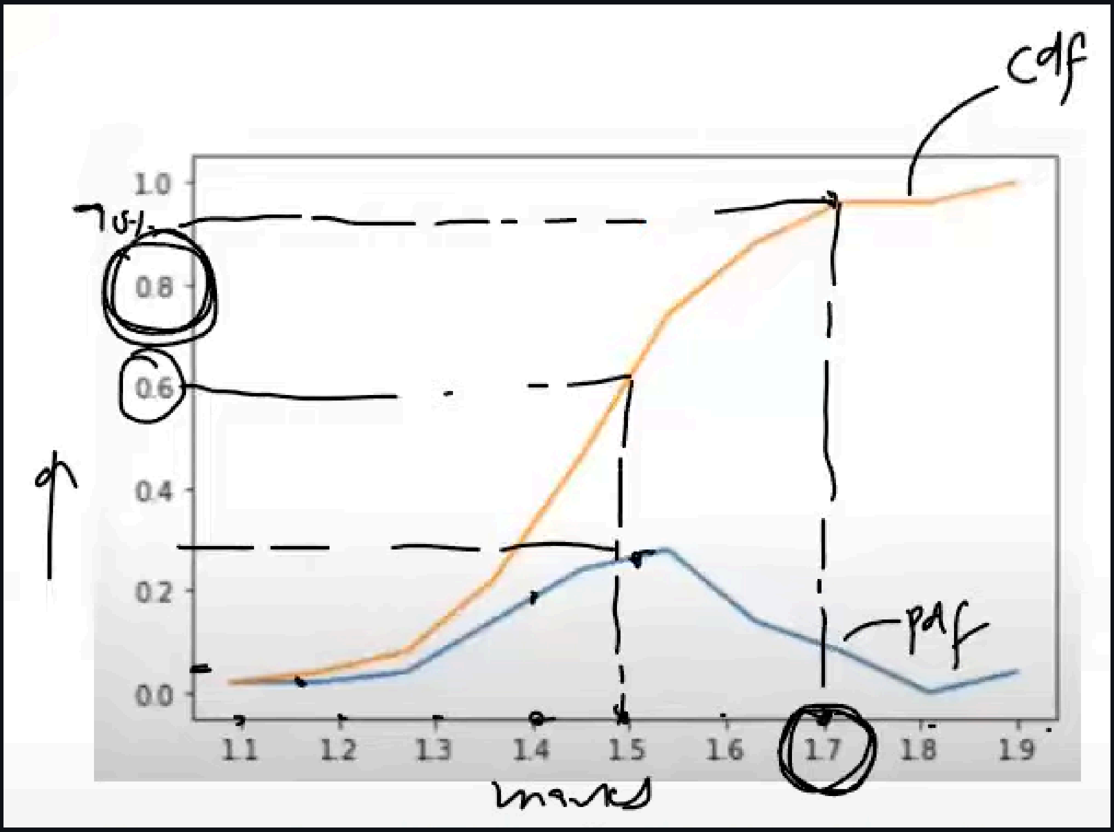
```
SELECT *,
NTILE(3) OVER(ORDER BY marks DESC) AS 'buckets'
FROM zomato.marks
-- What NTILE(n) essentially does is divides the data into n groups.
-- For n=3, you get 3 buckets for 33th, 66th and 99th percentile.
-- For n=4, you get 4 buckets for 25th, 50th, 75th and 100th percentiles.
-- Each bin ideally has equal no. of instances.
-- In the not rare scenario where 20 instances are to be divided in 3 buckets,
obviously 1 or 2 buckets will have extra instance(s).
```

IF ELSE / WHEN THEN / CASE in MySQL:

Here, if else is replaced by when then and the whole thing can be wrapped in a case.

```
-- For example, when you want to segment the smartphone dataset and assign a phone
type based on the price to each model.

SELECT *,
CASE
      WHEN bucket = 1 THEN 'premium'
      WHEN bucket = 2 THEN 'mid-range'
      WHEN bucket = 3 THEN 'budget'
END AS 'phone_type'
FROM (
      SELECT brand_name, model, price,
      NTILE(3) OVER(ORDER BY price DESC) AS 'bucket'
      FROM aditya.smartphones
) t1
```

Cummulative Distribution using CUME_DIST():



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

It answers the question: "What percentage of the rows in the data set have a value less than or equal to the current row?"

For example, in the marks dataset, assuming that each row contains 1 student and 1 student is only represented by 1 row, then if we use CUME_DIST() on marks, then for each row, the function looks at the marks in the row, finds where it lies in the quartile range distribution or basically finds which percentile those marks represent and returns it (in this case percent of students that have marks less than the given marks)

```
SELECT * FROM (
      SELECT *,
      CUME_DIST() OVER(ORDER BY marks) AS 'percentile_score'
    FROM zomato.marks
) t1
```

Database engine:

- Component of a DBMS software
- Has query optimizer

- Has transaction manager which ensure transactions are executed correctly and consistently
- Has storage manager that handles physical storage of data
- Has buffer manager which is responsible for managing cache of data in memory to optimize query performance

Famous MySQL DB engines:

InnoDB	**MyISAM**
Supports transactions	Doesn't support transactions
Has row level locking, analogous to thread lock in Global Interpreter lock in python	Has table level locking
Write operations are faster	Read operations are faster, because searching is much faster because of indexing
Highly scalable	Slower for very big databases

Components of DBMS:

1. DB engine
2. Security and access control - to manage user access and permissions and rights
3. Backup and Recovery - create backups and data recovery
4. Data dictionary - used to store metadata of the database (schema, column names, constraints, key columns, etc)
5. User interface

Collation:

- Collation refers to the rules and algorithms used to compare and sort characters in a database. It determines how character strings are compared and sorted, including order of characters, the treatment of case sensitivity, and handling of accent marks or other special characters.
- Collation is important in DBM because it affects the way queries are executed and results are returned. If the collation of a database is not set correctly, queries may return incorrect results or the database may not sort data properly
- Types of collation:
    a. Binary - compares strings byte by byte. Case sensitive and accent sensitive
    b. Case insensitive
    c. Accent insensitive

    d. Case and accent insensitive
    e. Unicode

COUNT(*) vs COUNT(col) vs COUNT(1):

- COUNT(*) counts all the number of rows
- COUNT(col_name) counts all non null values in the specified column
- COUNT(1)

Dealing with NULL values:

- Any comparison, algebraic operations with NULL values result in NULL values.
    - WHERE col_name = NULL doesn't work (returns single NULL row). You have to use IS NULL or IS NOT NULL
- When sorting / ordering by a column containing NULL values, NULL values have least value, i.e. they will be on top when sorting in ascending order.
- When grouping by a column containing NULL values, the NULL values are assigned a separate group.

- Aggregate operations:- When performing agg ops in MySQL, NULL values are treated differently depending on whether or not the group by clause is used.
    - Without GROUP BY:-
        - If agg funcs are sum, avg, max, min, count then NULL values are obviously ignored and not included in ops.
        - If the agg func is GROUP_CONCAT or CONCAT, NULL values are included in the result, but a NULL value is returned if all the values being concatenated are NULL
    - With GROUP BY:-
        - If agg func is COUNT then NULL values are not included in the count for each group. However, if you use COUNT(*) instead of COUNT(col_name), then NULL values are included in the count.
        - If the agg func is sum, avg, max, min then NULL values are obviously ignored. If a group contains only NULL values, then the result for that group will be NULL
        - If the agg func is GROUP_CONCAT or CONCAT, NULL values are included in the result for each group, but a NULL value is returned if all the values being concatenated are NULL

COALESCE(col_name, fill_value):

Basically returns the specified column, but the NULL values will be replaced with specified fill_value. You can use this in SELECT

DELETE vs TRUNCATE:-

- DELETE is a Data Manipulation Language (DML) statement, whereas TRUNCATE is a Data Definition Language (DDL) statement. This means that TRUNCATE requires the ALTER TABLE privilege, whereas DELETE requires the DELETE privilege on the table.
- DELETE can be rolled back using a transaction log, which means that you can undo the changes made by DELETE if necessary. TRUNCATE, on the other hand, cannot be rolled back because it does not generate a transaction log.
- DELETE is slower than TRUNCATE because it generates transaction log entries for each deleted row. If you need to delete a large number of rows, TRUNCATE may be a better option for performance reasons.
- If you use foreign key constraints in your database, DELETE can cause integrity issues if you delete rows that are referenced by other tables. In this case, you should use TRUNCATE or disable the foreign key constraints before using DELETE.

Data Types in MySQL:

String Data Types:

CHAR

This data type is used to store fixed-length strings. The length of the string is specified when the table is created, and the field will always use that amount of space, regardless of whether the string stored in it is shorter or longer. For example, if you define a CHAR(IO) field and store the string "hello" in it, MySQL will pad the string with spaces so that it takes up IO characters. CHAR fields are useful when you have a field that always contains the same length of data, such as a state abbreviation or a phone number.

VARCHAR

This data type is used to store variable-length strings. The length of the string can be up to a specified maximum, but the field will only use as much space as it needs to store the actual data. For example, if you define a VARCHAR(IO) field and store the string "hello" in it, MySQL will only use 5 characters to store the data. VARCHAR fields are useful when you have a field that can contain varying amounts of data, such as a user\'s name or address.

TEXT

This data type is used to store larger amounts of variable-length string data than VARCHAR. It can store up to 65,535 characters. TEXT fields are useful when you need to store large amounts of text data, such as blog posts or comments.

MEDIUMTEXT

This data type is used to store even larger amounts of text data than TEXT. It can store up to 16,777,215 characters. MEDIUMTEXT fields are useful when you need to store very large amounts of text data, such as long-form articles or legal documents.

LONGTEXT

This data type is used to store the largest amounts of text data. It can store up to 4,294,967,295 characters. LONGTEXT fields are useful when you need to store extremely large amounts of text data, such as entire books or large collections of data.

ENUM

The ENUM data type is used to store a set of predefined values. You can specify a list of possible values for an ENUM column, and the column can only store one of these values. The ENUM data type can be used to ensure that only valid values are stored in a column, and it can also save storage space compared to storing string values. Example - gender

```sql
ALTER TABLE trial ADD COLUMN GENDER ENUM('male','female')
```

If you try to set value for gender any other than the defined, it would give you an error

SET

The SET data type is similar to ENUM, but it can store multiple values. You can specify a list of possible values for a SET column, and the column can store any combination of these values. The SET data type can be used to store sets of values, such as tags or categories, in a single column. Example - hobbies

ALTER TABLE trial ADD COLUMN hobby SET('sports','gaming')

```sql
INSERT INTO trial (hobby) VALUES ('sports')
INSERT INTO trial (hobby) VALUES ('sports', 'gaming')
INSERT INTO trial (hobby) VALUES ('gaming')

-- If you try to insert anything else, it will throw a warning and insert an EMPTY
STRING
INSERT INTO trial (hobby) VALUES ('swimming', 'gaming')
```

Numeric Data Types

INT

The INT data type is used to store integers with a maximum value of 2147483647 and a minimum value of -2147483648. Examples of data that can be stored in INT include employee IDs, order numbers, and product IDs.

TINYINT

The TINYINT data type is used to store integers with a maximum value of 127 and a minimum value of -128. Examples of data that can be stored in TINYINT include Boolean values, such as O for false and 1 for true.

```
CREATE TABLE trial(
      user_id TINYINT,
      course_id TINYINT UNSIGNED
)
```

Note:- If you try to insert a value which is bigger than that attribute's defined size constraint, MySQL by default will fill the max value of that constraint instead. For example, if you try to insert 200 in a TINYINT attribute, the value will be inserted as 127 which is TINYINT's max value. It will show a warning

SMALLINT

The SMALLINT data type is used to store integers with a maximum value of 32767 and a minimum value of -32768. Examples of data that can be stored in SMALLINT include quantities of items, such as the number of products sold in a transaction.

MEDIUMINT

The MEDIUMINT data type is used to store integers with a maximum value of 8388607 and a minimum value of -8388608. Examples of data that can be stored in MEDIUMINT include the number of visitors to a website or the number of followers on a social media platform.

BIGINT

The BIGINT data type is used to store integers with a maximum value of 9223372036854775807 and a minimum value of -9223372036854775808. Examples of data that can be stored in BIGINT include the total revenue generated by a company or the number of views on a YouTube video.

FLOAT

The FLOAT data type is used to store single-precision floating-point numbers, which are numbers with a decimal point. Examples of data that can be stored in FLOAT include the price of a product or the temperature of a room.

DOUBLE

The DOUBLE data type is used to store double-precision floating-point numbers, which are numbers with a decimal point that can store more digits than FLOAT. Examples of data that can be stored in DOUBLE include very large or very small numbers, such as the distance between planets in the solar system or the size of an atom.

DECIMAL

The DECIMAL data type is used to store exact decimal values with a fixed number of digits before and after the decimal point. Examples of data that can be stored in DECIMAL include financial values, such as the cost of an item or the total balance in a bank account.

```sql
ALTER TABLE dt_demo ADD COLUMN price DECIMAL(5,2)
-- DECIMAL(5,2) means there are total 5 digits and there are 2 digits after the decimal

UPDATE dt_demo
SET price = 4563.4;
```

Note:- If you try to set value 1234.5 for decimal(5,2), the value will be set to 999.99
If you try to set 123.456 to decimal(5,2), it will round off after 2 decimals to 123.46

Note:- There is a precision vs memory trade-off when using float, double vs decimal

BLOB Data Type

The BLOB (Binary Large Object) data type in MySQL is used to store large binary data, such as images, audio, video, or other multimedia content. In MySQL, there are four types of BLOB data types that can be used to store binary data with different maximum sizes:

TINYBLOB

Maximum length of 255 bytes. TINYBLOB is the smallest BLOB data type in MySQL. It can be used to store small binary data, such as icons, small images, or serialized objects.

BLOB

Maximum length of 65,535 bytes (64 KB).BLOB is a medium-sized BLOB data type that can be used to store larger binary data, such as images, audio, video, or other multimedia files.

MEDIUMBLOB

Maximum length of 16,777,215 bytes (16 MB).MEDIUMBLOB is a larger BLOB data type that can be used to store even larger binary data, such as high-resolution images or longer audio or video files.

LONGBLOB

Maximum length of 4,294,967,295 bytes (4 GB).LONGBLOB is the largest BLOB data type in MySQL, and it can be used to store very large binary data, such as very high-resolution images, long audio or video files, or even entire documents.

```sql
ALTER TABLE trial ADD COLUMN image MEDIUMBLOB;

-- you can directly convert to BLOB format using LOAD_FILE() function
-- You need to use forward slashes when mentioning path
INSERT INTO trial (image) VALUES (LOAD_FILE('C:/trial.png'))
```

Pros of storing files in BLOB columns:

- BLOB columns allow you to Store binary data directly in the database, without needing to store the file externally.
- Storing files in the database can simplify and restore procedures, as all the data is in one place.
- Access to BLOB data can be controlled through database user permissions.

Cons of storing files in BLOB columns:

- Storing large files in the database can slow down database performance and increase storage requirements.
- If you need to access the file outside of the database (e.g. to share it with another application or user), you'll need to extract it from the database.
- Some file types may not be well-suited for storage in BLOB columns, depending on their size, structure, and how they are accessed.

Spatial Data Types

GEOMETRY

The GEOMETRY data type is a generic spatial data type that can store any type of geometric data, including points, lines, and polygons.

```
ST_ASTEXT(), ST_X(), ST_Y()
```

JSON

```sql
ALTER TABLE trial ADD COLUMN description JSON;

INSERT INTO trial (description) VALUES ('{"os":"android","type":"smartphone"}');

-- You can use the JSON_EXTRACT() function to extract/load the data
-- you pass in the column name and then a string of format '$.column_name'
SELECT JSON_EXTRACT(description, '$.os') FROM trial;

SELECT JSON_EXTRACT(description, '$.type') FROM trial;
```

Data Normalization

Why can't a single table hold all the data?

Redundant information

For example, consider order_id, user_id, name and detailed info about thge user like address, phone number, etc. Then for every order placed, redundant data about user info will be added inflating the table size unnecessarily.

Insert Anamoly

For example, consider the orders, users, and restaurants dataset merged together (non-normalized data). suppose then a user registers on the website but doesn't place any order. Then, that user's details would be filled int the dataset, but other info about the orders corresponding to that user will be NULL.

Delete Anamoly

Consider the same example. But, this time you want to delete all orders of a certain person, but you don't want to delete the user's info. but, consider this, since the data is not normalized, all orders of that user are linked to that user's info. If you delete those orders, you will also loose the user's info.

Update Anamoly

Consider the same anomaly. But, this time you want to update some specific info about a user, let's say the user's address. To update it, you will need to update the address fields for all that user's orders. This will result in unnecessary updates. Also, if you fail to update address for some orders, then you will have an anomaly, where one user will have unintentional 2 addresses.

What is Data Normalization?

Database normalization is a process used to organize data in a database to reduce data redundancy and dependency. The goal of normalization is to ensure that each piece of data is stored in one place, in a structured way, to minimize the risk of inconsistencies and improve the overall efficiency and usability of the database, There are several levels Of database normalization, each with its own set Of rules and guidelines. The most commonly used levels of normalization are:

1. ***First Normal Form (lNF):*** This level requires that all data in a table is stored in a way that each column contains only atomic (indivisible) values, and there are no repeating groups or arrays.
2. ***Second Normal Form (2NF):*** This level requires that each non•key attribute in a table is dependent on the entire primary key, not just a part of it.
3. ***Third Normal Form (3NF):*** This level requires that each non-key attribute in a table is dependent only on the primary key and not on any other non.key attributes.

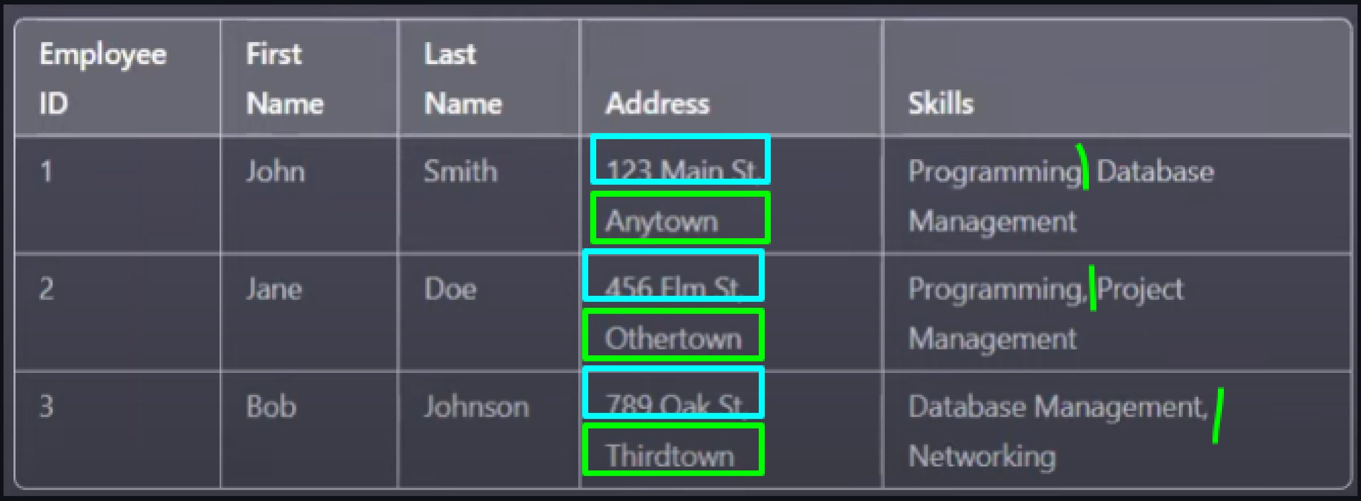
There are ***higher levels Of normalization***, such as **BCNF** (Also called **3.5NF**), Fourth Normal Form (**4NF**) and Fifth Normal Form (**5NF**), but they are less commonly used in practice.

Levels of Normalization (Normal Forms)

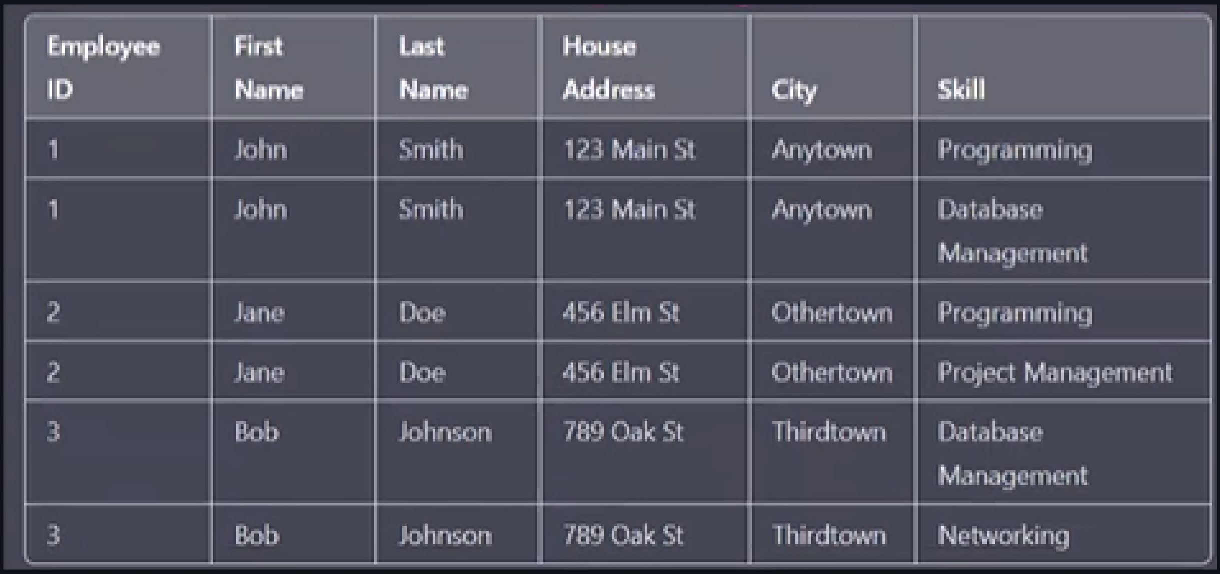
1st Normal Form (1NF)

A table is in 1 NF if:

- There are only Single Valued Attributes or each col should contain atomic values.
- Attribute Domain does not change data type should not change.
- There is a unique name for every Attribute/Column.
- The order in which data is stored does not matter.



Employee ID	First Name	Last Name	Address	Skills
1	John	Smith	123 Main St Anytown	Programming, Database Management
2	Jane	Doe	456 Elm St Othertown	Programming, Project Management
3	Bob	Johnson	789 Oak St Thirdtown	Database Management, Networking



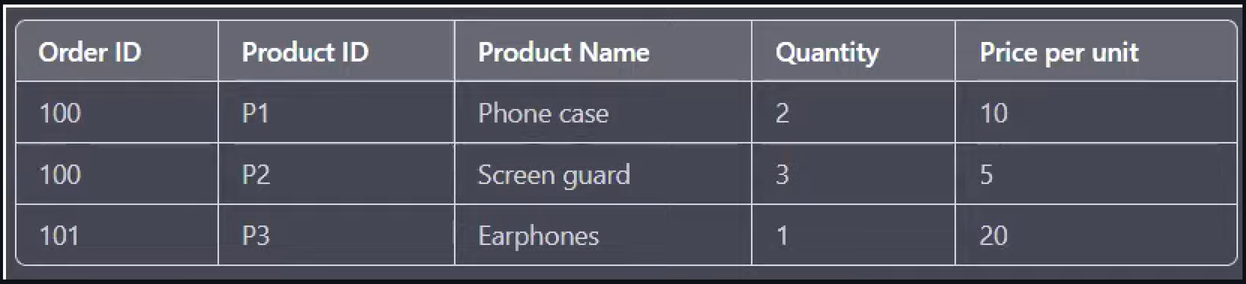
Employee ID	First Name	Last Name	House Address	City	Skill
1	John	Smith	123 Main St	Anytown	Programming
1	John	Smith	123 Main St	Anytown	Database Management
2	Jane	Doe	456 Elm St	Othertown	Programming
2	Jane	Doe	456 Elm St	Othertown	Project Management
3	Bob	Johnson	789 Oak St	Thirdtown	Database Management
3	Bob	Johnson	789 Oak St	Thirdtown	Networking

2nd Normal Form (2NF)

A table is in 2NF if:

- It is already in 1NF.
- It does not contain any partial dependency.

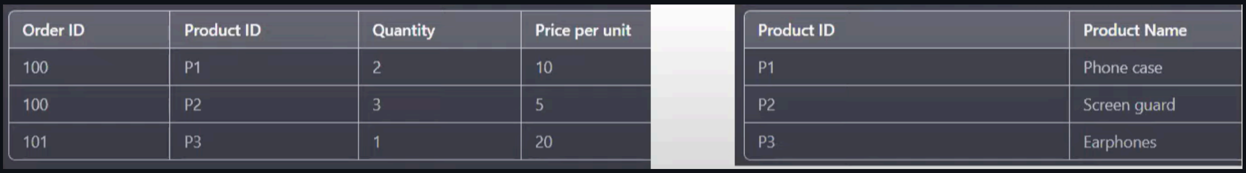
Partial dependency - occurs when a non-key attribute is dependent on only the part of the primary key instead of the entire key.



Order ID	Product ID	Product Name	Quantity	Price per unit
100	P1	Phone case	2	10
100	P2	Screen guard	3	5
101	P3	Earphones	1	20

Here, assume that Order ID and Product ID together form a composite primary key. Then, looking at other columns:

- The attribute 'quantity' depends on order ID and product ID
- The attribute 'price per unit' depending on who you ask, depends both on Order ID and Product ID (price may change for subsequent orders)
- But, the attribute 'Product Name' depends only on Product ID. Hence, it's a partial dependency. We need to change this to bring this to 3NF



Order ID	Product ID	Quantity	Price per unit
100	P1	2	10
100	P2	3	5
101	P3	1	20

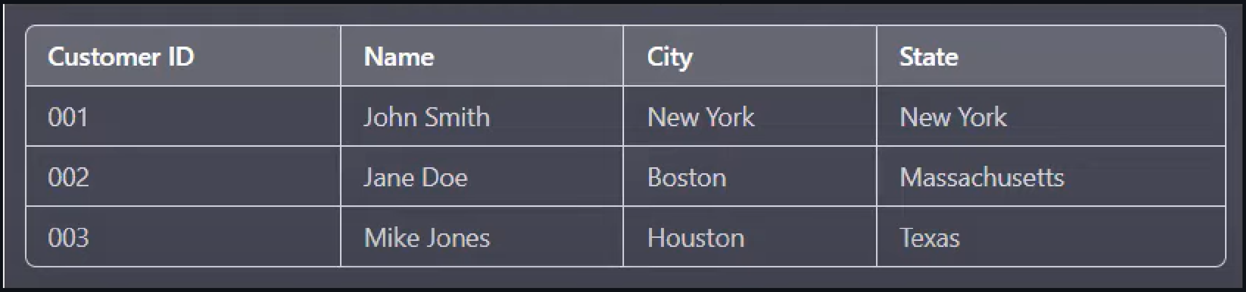
Product ID	Product Name
P1	Phone case
P2	Screen guard
P3	Earphones

3rd Normal Form (3NF)

A table is in 3NF if:

- It is already in 2NF.
- There is no transitive dependency.

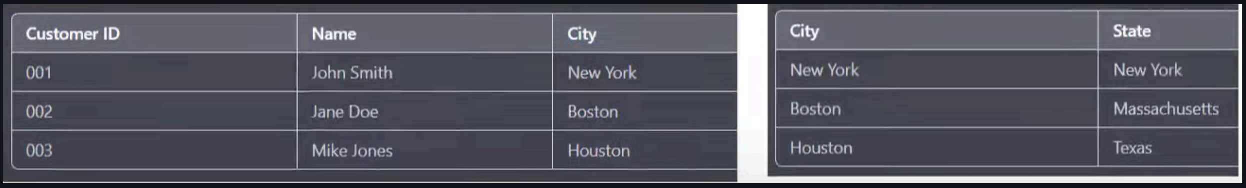
A ***transitive dependency*** exists, when a non-key attribute depends on another non-key attribute, which is not part of primary key



Customer ID	Name	City	State
001	John Smith	New York	New York
002	Jane Doe	Boston	Massachusetts
003	Mike Jones	Houston	Texas

Here, looking at the columns (assuming Customer ID is Primary key)

- The attribute 'Sate' depends on 'City'. Hence, you can see a transitive dependency here.



Customer ID	Name	City
001	John Smith	New York
002	Jane Doe	Boston
003	Mike Jones	Houston

City	State
New York	New York
Boston	Massachusetts
Houston	Texas

3.5th Normal Form (3.5NF) (BCNF)

4th Normal Form (4NF)

5th Normal Form (5NF)

6th Normal Form (6NF)

Views

In SQL, a view is a virtual table that does not store any data on its own but presents a customized view of one or more tables in a database. A view can be thought of as a pre-defined SELECT statement that retrieves data from one or more tables and returns a specific subset of data to the user.
So basically it is a logical table instead of a physical table Once a view is created, it can be used in the same way as a table in SQL queries, and any changes made to the underlying tables will be reflected in the view.

Simple Views - Created from 1 single table

Complex Views - Created from multiple tables with the help of joins, subquery etc.

Read only Vs Updatable Views

Read-only views: As the name suggests, read-only views are views that cannot be updated. They are used to simplify the process of querying data, but they cannot be used to modify or delete data in the underlying tables.

Updatable views: Updatable views are views that allow you to modify, insert or delete data in the underlying tables through the view. They behave like normal tables, but with restrictions.
To make a view updaNtable, certain conditions must be met. For example, the view must not contain any derived columns, subqueries, or aggregate functions. Additionally, the view must be based on a single table or a join of tables with a unique rdone-to-one relationship.

Materialized Views

A materialized view is a database Object in SQL that contains the results Of a query. Unlike regular views, which are just virtual tables that store SQL queries, materialized views are physical tables that store the results of a query. Materialized views are precomputed and stored on disk, which makes them much faster to access than regular views.

Benefit - Faster queries

Disadvantage - Need to manually update the view

Advantages of Views

- ***No physical storage***
- ***Make complex queries simple***
- ***Security*** - Consider an example where you are a DB admin and you have a table with sensitive info about your users. And, you want to give access of this table to your programmers/analysts without the sensitive info. You could do this by creating a view of that table which doesn't include the sensitive info and then give the programmers/analysts access to this view instead of the original table.

User Defined Functions

User-defined functions (UDFs) in SQL are functions that are created by users to perform specific tasks. These functions can be used just like built-in functions in SQL and can take parameters as input, perform some operations on them, and then return a value.

Syntax

```
DELIMITER $$

CREATE FUNCTION IF NOT EXISTS `function_name` (
      Parameter_1 Data_Type,
      Parameter_2 Data_Type,
      Parameter_n Data_Type,
)
RETURNS return_datatype
[NOT] DETERMINTSTIC
BEGIN
      Function Body
Return return_value
END $$

DELIMITER ;
```

Example Functions

Hello World

```
DELIMITER $$

CREATE FUNCTION IF NOT EXISTS `hello_world` ()
RETURNS VARCHAR(255)
-- [NOT] DETERMINTSTIC
BEGIN
Return "Hello World!";
END $$

DELIMITER ;

-- SELECT QUERY
SELECT hello_world() FROM table_name
```

Proper Name

```
DELIMITER $$

CREATE FUNCTION IF NOT EXISTS `proper_name` (
      name VARCHAR(255),
      gender VARCHAR(255),
      is_married VARCHAR(255),
)
RETURNS VARCHAR(255)
-- [NOT] DETERMINTSTIC
BEGIN
      DECLARE str VARCHAR(255);
      DECLARE prefix VARCHAR(255);

      SET str=TITLE(name);

      IF gender='M' THEN
            SET prefix='Mr';
      ELSE
            IF is_married='Y' THEN
                  SET prefix='Mrs';
            ELSE
                  SET prefix='Ms';
            END IF;
      END IF;
      str = CONCAT(prefix, " ", str)
Return str;
END $$

DELIMITER ;
```

```sql
-- SELECT QUERY
SELECT proper_name(name, gender, is_married) FROM table_name
```

Deterministic vs Non Deterministic Functions

Stored Procedures

A stored procedure is a named block of SQL statements and procedural logic that is stored in a database and can be executed by a user or application (like a website, mobile app, python program, etc).

Stored procedures are often used to encapsulate business logic and application logic, such as data validation, data processing, and database updates. By using stored procedures, developers can separate application logic from the presentation layer and simplify the application code.

Benefits of Stored Procedures

- ***Improved performance:*** Stored procedures are precompiled and optimized, which can improve performance and reduce network traffic.
- ***Enhanced security:*** Stored procedures can be granted specific permissions and access rights, which can improve security and limit access to sensitive data.
- ***Encapsulation of business logic:*** Stored procedures allow developers to encapsulate complex business logic and make it easier to maintain and update.
- ***Consistency:*** Stored procedures ensure that database operations are performed in a consistent manner, which can help to maintain data integrity.
- ***Reduced network traffic:*** By encapsulating data access and manipulation logic in stored procedures, developers can reduce the amount of data that needs to be transmitted over the network.

String Functions:

Wildcards (LIKE operator):

The LIKE operator in MySQL is used to match a string value against a pattern using wildcard characters. It is commonly used in SELECT, WHERE, and JOIN clauses to filter or join rows based on a pattern match. The LIKE operator uses two wildcard characters: the percent sign (%) and the underscore (_). The percent sign represents zero, one, or more characters, while the underscore represents a single character.

MISCELLANEOUS RESOURCES:

1. Recursive queries with Common Table expressions (CTEs)
   http://database-programmer.blogspot.com/2010/11/recursive-queries-with-common-table.html
2. SQL Joins https://learnsql.com/blog/sql-joins/
3. MySQL Tutorial org https://www.mysqltutorial.org/
4.