__/__/____

DATABASE TECHNOLOGIES.

Database Types:

1. Relational / Sequential - MySQL, Oracle, IBM DB2, Microsoft SQL.
2. NoSQL - MongoDB, Cassandra, CouchDB, HBase, NoSQL DB.
3. Graph DB - Neo4j, Graph DB
4. Memory DB - MemDB, VoltDB
5. Disk Based DB - Local computer storage - structured

SQL Keys:

1. Simple Primary key - Minimal number of columns (One in this case) that identifies the row uniquely + not null.
2. Composite Primary key - More than one column, being used as primary key.

3. Candidate key - All possible combinations which are capable of becoming a Primary key.
4. Unique key - All values of this column are unique but also consist of zero or more NULL values.

5. Foreign key - One column referencing another column of same or different table for correctness of data.

PL-SQL : Procedural Language with features like loops, functions, conditional statements, etc.

DQL	- Query	- SELECT
DML	- Manipulation	- INSERT, UPDATE, DELETE
DDL	- Definition	- CREATE, ALTER, TRUNCATE, DROP
TCL	- Transaction Control	- ROLLBACK, COMMIT, SAVEPOINT
OCL	- Control	- GRANT, REVOKE

AUTO COMMIT → STATEMENTS

__/__/____

★ Operators used inside WHERE clause :

1. Relational :- = > < >= <= !=
2. Logical :- AND OR NOT
3. [NOT] IN :- WHERE city IN ('Pune', 'Mumbai');
   (replaces ~~AND~~ OR)
   WHERE sal IN (3000, 4000, 5000);
4. [NOT] BETWEEN ... AND
   (replaces ~~●~~ AND) :- WHERE price BETWEEN 10 AND 50;
   WHERE date BETWEEN '1996-07-01' AND '1996-07-09';
5. [NOT] LIKE :- % zero or multiple characters
   _ one single character
   WHERE LIKE pattern;

'a%' '%a' '%a%' 'a%e' 'a-e%' '-r%'
Start End Between Start--End

6. IS [NOT] NULL :- WHERE sal IS NOT NULL;

★ Database:

1. SHOW DATABASES;
2. USE database-name;
3. CREATE DATABASE IF NOT EXISTS db-name;
4. DROP DATABASE IF EXISTS db-name;

★ Clear Screen

mysql> \! cls

__/__/____

★ Tables:

1. SHOW TABLES;
2. DESC table-name;
3. DESC table-name col-name;
4. CREATE TABLE IF NOT EXISTS tb-name (col1 datatype,
   col2 datatype ...);

5. Copy existing table data into new table with same structure.
   CREATE TABLE tb2 AS SELECT col1, col2, ...
   FROM existing-table WHERE ... ;
   (Only data is copied & not the constraints)
6. DROP TABLE (IF EXISTS) tb-name;
7. TRUNCATE TABLE tb-name; ← Removes table data.

★ Constraints of a Table:

CREATE TABLE tb-name(col1 datatype constraints,
col2 — — ,...);

1. NOT NULL
2. UNIQUE - UNIQUE (Name) ①
3. PRIMARY KEY - PRIMARY KEY (ID) ① One or multiple
4. FOREIGN KEY - FOREIGN KEY (PID) REFERENCES Persons (PID)
   (Variable; col name)
5. DEFAULT - City varchar(200) DEFAULT 'Mumbai'
6. CHECK - CHECK (Age >= 18) ①
7. AUTO-INCREMENT - AUTO-INCREMENT = 100 (Starts from 100)
   INSERT INTO tb-name VALUES (default, —, —, —);

Q How to name a constraint explicitly?
CONSTRAINT Person-Pk PRIMARY KEY (ID, LastName)
↑ Name for our composite primary key.

__/__/____

★ ALTER

ALTER TABLE table-name

Default Position is at END

1) ADD col-name datatype constraints [FIRST | AFTER col-name];
2) DROP COLUMN col-name;
3) MODIFY COLUMN col-name datatype;
4) RENAME COLUMN old-name To new-name;
5) RENAME TO new-table-name;
6) ALTER OrderDate SET DEFAULT '2022-03-24';
7) ALTER OrderDate DROP DEFAULT;
8) ADD CONSTRAINT const PRIMARY KEY (col2);

★ NULL Functions:

1. IFNULL() - Replaces NULL values.

```
SELECT ProductName, Price * (Stock + IFNULL (Order, 0))
FROM Products;
```

2. COALESCE() - Returns 1st non-NULL value, from input list.

```
SELECT ProductName, Price * (Stock + IFNULL/COALESCE
FROM Products;                      (Order, 0))
```

3. ISNULL () - Returns 0 or 1

```
SELECT ISNULL (NULL);      ← 1
       ISNULL ("suraj");   ← 0
```

★ INSERT

1) INSERT INTO tb-name (col1, col2, ..)
   VALUES (val1, val2, --) } Multiple rows
   (val1, val2, --);

2) INSERT INTO first-tb (col1, col2, col3)
   SELECT col1, col2, col3 FROM second-tb
   WHERE condition;

__/__/____

★ UPDATE

```
UPDATE table-name
SET col1 = ___ , col2 = ___ ...
WHERE condition;
```

← If Where clause is skipped, it will update all records.

★ DELETE

```
DELETE FROM tb-name WHERE condition;
```

★ SELECT

1. `SELECT * FROM tb_name;`
2. `SELECT col1, col2, ... FROM tb-name;`
3. `SELECT DISTINCT col1 FROM tb-name;`

★ Aggregate Functions:

1. `SELECT MIN (col1) FROM tb-name;`
2. MAX
3. AVG
4. SUM
5. `SELECT COUNT(*) FROM tb-name;` ← counts NULL values

   `COUNT(Sal)` ← IGNORES NULL VALUES

6. `SELECT AVG (IFNULL (sal, 0)) FROM tb-name;`

   AVG ← unexpected

   IFNULL ← Replace NULL values with 0.

Gives average ~~NULL~~ if any one row is null.

__/__/____

★ GROUP BY , ORDER BY , HAVING Clause.

```
SELECT col-name(s)  FROM tb-name
WHERE condition                ①*
GROUP BY col-name(s)
HAVING condition  ← used on aggregate functions
ORDER BY col-names(s)  ASC | DESC ;
```

Note:

1.* Column name in GROUP BY clause and in SELECT query must be same, with an exception that SELECT query can have aggregate function operations on other columns as well.

Eg.

```
SELECT deptno , sum(sal) , avg(sal) , count(*)
FROM emp
WHERE condition  ← If Any
GROUP BY deptno
HAVING count(*) >= 4;
```

★ LIMIT & OFFSET

```
SELECT * FROM tb-name
LIMIT [row-count]  OFFSET [offset-val];
```

LIMIT 1 ← top 1 row

↑ skips this no. of rows.

LIMIT 2,1 ← top 1 row starting from row 2

★ Numeric Functions:

1.	Round (12.34567 , 2)	→	12.35	Round off
2.	Truncate (12.34567 , 2)	→	~~12~~ 12.34	Chop off
3.	Floor (12.7765)	→	12	Previous
4.	Ceil (12.3356)	→	13	Next
5.	Abs (-34)	→	34	Modulus

__/__/____

★ String Functions:

1. Upper(str) Lower(str)
2. Substr ('HelloSQL', 3, 3) ← start (inclusive) → Start index is 1
   ↑ length
3. concat (str1, str2, str3);
4. Lpad (old-str, 10, new-str); ← concatenation with repetition
   Rpad (———)
5. Ltrim, Rtrim, Trim ← Removes white spaces
6. length (str) or char_length(str)
7. Reverse (str)
8. Format (123656.321698, 2) ⇒ 123,656.32
   (123.61, 4) ⇒ 123.6100
9. Instr (str, substr)
   ↑ Returns 1st occurance position
10. Insert (str, pos, len, newstr)
    ↑ start point ↑ characters to be replaced
    Eg. insert ('WelcomeToSQL', 4, 4, 'Hello')
    ↳ Wel HelloToSQL
11. Left (str, len) ← Substrings → Right (str, len)

——— × ——— × ———

★ Difference Between Delete & Truncate:

TRUNCATE TABLE tb-name;

	Delete	Truncate
1.	DML Query.	DDL Query.
2.	Rollback is possible.	Rollback is not possible.
3.	Where condition can be used.	Where condition is unavailable.

__/__/____

☆ DATE

1. NOW() → current date and time
2. DATE(NOW()) → only date
3. TIME(NOW()) → only time
4. CURDATE() → only date

5. DATE_FORMAT(CURDATE(), '%d-%m-%Y')
   ↑ separators can be changed according to requirement.

- %d - numeric
- %D - th, st after date
- %m - numeric month
- %M - month in word format
- %y - 2 digit date
- %Y - 4 digit date

6. DATEDIFF('2015-11-04', '2014-9-21')
   DATEDIFF(CURDATE(), hiredate)/365
   floor(DATEDIFF(CURDATE(), hiredate)/365)

7. Date after certain interval (Future Date):

(i) DATE_ADD('2015-06-30', INTERVAL 1 DAY)
↓
DAY | MONTH | WEEK | YEAR

(ii) DATE_ADD(('2015-09-01', INTERVAL 2 YEAR), INTERVAL 6 MONTH)

8. Date before certain interval (Past Date):
   DATE_SUB(_____, INTERVAL ___)

__/__/____

1. To fetch DAY, MONTH, QUARTER and YEAR of a date value.

```
SELECT  DAY('2000-12-31') day,          31
        MONTH ————————                  12
        QUARTER ————————                4
        YEAR ————————;                  2022
        WEEKDAY ————————;               6
        WEEK ————————;                  53
        WEEKOFYEAR ————————;            52
        MONTHNAME ————————;             December
        DAYNAME ————————;               FRIDAY
```

———— × ———— × ———— × ————

* SHOW CREATE TABLE tb-name; ← Gives all queries required to build this table.

* DCL Queries:

→ GRANT privileges-names ON object TO user;

[mysql user name]

[SELECT, INSERT, CREATE, DELETE, INDEX, ALTER, UPDATE, DROP, ALL, GRANT] — [database object / table]

→ GRANT EXECUTE ON [PROCEDURE | FUNCTION] object TO user;

→ REVOKE privileges ON object FROM user;

__/__/____

★ Foreign Key Constraints:

constraint pid-fk FOREIGN KEY (pid) REFRENCES employee (pid) ON DELETE CASCADE ON UPDATE CASCADE

1. CASCADE - If we Delete/Update rows from Parent table, it automatically Delets/Updates rows in Child table.

2. SET NULL - On performing Delete/Update, it sets Foreign key column/columns in child table to NULL.

Default →3. NO ACTION - MySQL server rejects Update/Delete operation for parent table if there is a related Foreign key value present in reference table.

4. SET DEFAULT - SET DEFAULT values for child table if UPDATE/DELETE is being performed on parent table.

5. RESTRICT - Rejects Update/Delete operation on parent table, if we have a related Foreign key.

★ ACID Properties:

A → Atomicity → Transaction as a single unit

C → Consistency → Transaction should be consistent with db

I → Isolation → Transactions run in isolated environment

D → Durability → Recovery from unexpected failures, backup.

__/__/____

★ Transaction Control Language.

```
SET autocommit = 0/1 ;
SELECT @@autocommit;
commit;
rollback;
Savepoint A;
Savepoint B;
rollback To A;
```

★ Nested Query:

Q1. Fall all employees with sal > Blake's salary.

→
```
select * from emp where sal >
        (select sal from emp where
                ename = 'BLAKE');
```

Q2. sal > average salary for dept 10.

→
```
Select * from emp where sal >
        (select avg(sal) From emp where
                deptno = 10);
```

Q3. sal > Smith's salary and <= blake's salary.

→
```
Select * from emp where sal BETWEEN
        ( select * sal from emp where ename = 'SMITH')
    AND ( select sal from emp where ename = 'BLAKE');
```

Q4. sal is either SMITH or BLAKE

→
```
Select * From emp where sal IN
    (select sal from emp where ename IN
            ('SMITH', 'BLAKE'));
```

__/__/____

Q5. Sal > average salary of their own department.

→
```
Select * from emp e where sal >
    (select avg(sal) from emp m where
        m.deptno = e.deptno );
```

Q6. Sal < avg salary of ALLENS department.

→
```
Select * from emp where sal <
    (select avg(sal) from emp where deptno =
        (select deptno from emp where
            ename = 'ALLEN') );
```

Q7. Sal > average salary of either dept 10 or 20.

→
```
Select * from emp where sal > ANY
    (select avg(sal) from emp where
        deptno IN (10,20));
```

★ EXISTS clause

Q8. Find all employees under whome some other employees are working.

→
```
Select * from emp e where EXISTS
    (select * from emp m where
        m.mgr = e.empno );
```

Q9. Employees that have not bought any vehicle

→
```
Select * from customer c where NOT EXISTS
    ( select * from vehicle v where
        v.custid = c.cid );
```

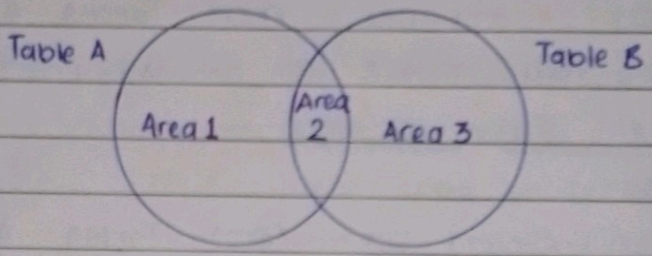
__/__/____

★ JOINS:

Joins should be used only when we require output from more than one table.

Note: Joins are memory inefficient and slow in terms of execution time, hence should be avoided in real life scenarios.

Types of Joins (Cheat Sheet):



1. Set A (1+2):

```
SELECT <select-list> FROM TableA A
LEFT JOIN TableB B ON A.key = B.key;
```

2. Set B (2+3):

```
—— " —— RIGHT JOIN Table B —— . ——;
```

3. Set (A-B) or (1):

```
—— " —— LEFT JOIN TableB B
ON A.key = B.key WHERE B.KEY IS NULL;
```

4. Set (B-A) or (3):

```
—— . — RIGHT JOIN TableB B
ON A.key = B.key WHERE A.key IS NULL;
```

5. Set (A∩B) or (2):

```
SELECT <slect-list> FROM TableA.A INNER JOIN TableB.B
ON A.key = B.key;
```

__/__/____

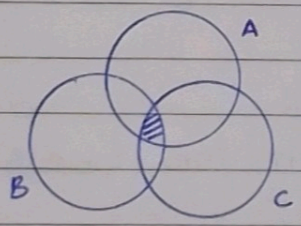
6. Set (AUB) or (1+2+3):

```
SELECT <select-list> FROM TableA A
LEFT JOIN TableB B ON A.key = B.key
UNION
SELECT <slect-list> FROM TableA A
RIGHT JOIN TableB B ON A.key = B.key;
```

7. Set (AUB - A∩B) or (1+3):

```
SELECT <slect-list > FROM TableA A
LEFT JOIN TableB B ON A.key = B.key
WHERE B.key IS NULL
UNION
SELECT <select-list> FROM TableA A
RIGHT JOIN TableB B ON A.key = B.key
WHERE A.key IS NULL ;
```

8.



Set (A∩B∩C):

```
SELECT <select-list> FROM TableA A
INNER JOIN TableB B ON A.key = B.key
INNER JOIN TableC C ON B.key = C.key;
```

__/__/____

★ VIEWS: Used to give restricted access to existing table.

1. A view is a virtual table based on the result set of a SQL statement and it does not occupy any extra space in memory.

2. Views are of two types:
   (i) Simple views - based on single table
   (ii) Complex views - based on more than one table.

3. We can perform DML operations on simple views with certain conditions:
   (i) The SELECT query used in creation of VIEW should not contain Group By or Order By clause.
   (ii) DISTINCT keyword must not be present.
   (iii) The VIEW should have all NOT NULL columns

4. Complex views are Read Only by default.

Find all VIEWS:

```
SHOW FULL TABLES IN <db-name>
WHERE TABLE-TYPE LIKE 'VIEW';
```

CREATE a VIEW:

```
CREATE VIEW <view-name> AS
SELECT ---- FROM --. WHERE --. ;
```

DROP a VIEW:

```
DROP VIEW <view-name>;
```

__/__/____

UPDATE a VIEW : Possible only when conditions specified in point 3 satisfy.

CREATE OR REPLACE VIEW <view-name> AS
SELECT ... FROM ... WHERE ... ;

ORACLE specific queries:

1. CREATE VIEW <view-name> AS
   (SELECT ---)
   WITH READ ONLY ; ← Imposing restrictions

2. CREATE MATERIALIZED VIEW <view-name> AS
   (SELECT ---); ↑
   Allocates physical memory for the result of the query (snapshot).

Why use VIEWS?

1. Restricts data access.
2. Hiding complexity of queries.
3. In case of materialized views, reduce cost of execution by storing results.

__/__/____

★ INDEXES

Indexes are set of keys made up of single or multiple columns, to help faster retrival of rows by using a pointer.

Two type of indexes:

1. Clustered index - Created on primary key by default. We can have only one clustered index for each table. They are physically stored in order (ie. in ascending or descending order).

2. Non-clustered index - Created by joining multiple columns. A table can have one or more non clustered indexes. Foreign keys are usually placed in non clustered indexes.

(i) SHOW ALL INDEXES:

```
SHOW INDEX FROM <table-name>;
```

(ii) CREATE AN INDEX:

```
CREATE [UNIQUE] INDEX <index-name> ON
<table-name>( col1, col2,--);
```

(iii) DROP AN INDEX:

```
DROP INDEX <index-name> ON <table-name>;
```

Note : 1. Indexes should be created on the columns which are used frequently in the WHERE clause

2. Indexes should not be created on columns that are frequently manipulated, as the corresponding indexes are updated as well.

__/__/____

★ PL/SQL

(1.) Procedures: A procedure is a series of SQL statements stored in database, that does not return any value. Procedures can be called from functions, triggers, other procedures or applications like Java.

Syntax:

(i) Create a Procedure:

```
DELIMITER $$
CREATE PROCEDURE GetCustomers ( param-list )   <- zero or more
BEGIN
    SELECT - - -  ;
END $$
DELIMITER ;

CALL GetCustomers ( );   <- argument-list
```

(ii) Drop Procedure:

```
DROP PROCEDURE [IF EXISTS] GetCustomers;
```

(iii) Display procedure code:

```
SHOW CREATE PROCEDURE GetCustomers;
```

(iv) Declaring a Variable: To be declared inside BEGIN ... END

```
DECLARE var-name datatype(size) [DEFAULT def-value];
```

↑ we can declare more than one variable at once. (var1, var2, ...)

(v) Initializing a Variable: Static initialization

```
SET var-name = 10;
```

__/__/____

(vi) Dynamic initialization of a variable :

```
BEGIN
DECLARE product-count INT DEFAULT 0;
SELECT count(*) INTO product-count
FROM products;
END $$
```

(vii)
1. The variables used in above examples were local variables and had scope within 'BEGIN-- END' block.
2. But we can declare session variables, that has its visibility for current session.
3. Such session variables does not require declaration.
4. Session variables are generally used with OUT parameters.

Ex.

```
SET @message = 'Current date and time is:'
SELECT CONCAT (@message, CURRENT-TIMESTAMP);
```

(viii) Procedure Parameters

1. IN : Default mode, Read only, Used to pass input.
2. OUT : Its value can be changed inside a procedure, they do not have any default/initial value & hence must be initialized within the procedure.
3. INOUT : Read and Write

Syntax:

```
[IN | OUT | INOUT ] parameter-name datatype (length)
```

Eg.

```
CREATE PROCEDURE GetOrders (
                    IN  OrderNum INT,
                    OUT total INT )
```

__/__/____

★ Conditional Statements

1. IF - THEN :

```
IF condition THEN
    statements ;
END IF ;
```

2. IF - THEN - ELSE :

```
IF condition THEN
    statements;
ELSE
    else-statements;
END IF ;
```

3. IF - THEN - ELSEIF - ELSE :

```
IF condition1 THEN
    statements;
ELSEIF cond2 THEN
    state.. ;
ELSE
    state--;
END IF;
```

4. CASE statements :

```
CASE case-value
    WHEN value1 THEN
        statement;
    WHEN value2 THEN
        statement;
    :
    ELSE
        statement;
END CASE;
```

★ LOOPS

1. WHILE : Top tested loop.

```
WHILE condition DO
    statements;
END WHILE ;
```

2. Repeat Loop : Bottom tested

```
REPEAT
    statements;
UNTIL condition
END REPEAT ;
```

3. LOOP: Infinite loop. Use leave statement to break the loop.

```
loop-label : LOOP
        IF condition THEN
                LEAVE loop-label;
        END IF;
        ...
    END LOOP;
```

Note: LEAVE statement can also be used to exit a procedure execution.

---

② Functions : A function always returns a single value, and hence a function can be used in a SQL statement.

Syntax:

(i) Create a function:

```
DELIMITER $$
CREATE FUNCTION funct-name (param1 , param2 , ..)
        RETURNS datatype [NOT] DETERMINISTIC
BEGIN
    statements;
    RETURN value;
END $$
DELIMITER ;
```

DETERMINISTIC : A deterministic function always returns result for same input parameters, on the other hand a non-deterministic function returns different results. Default value for a function is: 'NOT DETERMINISTIC'.

__/__/____

Note: A function definition must contain atleast one RETURN statement. The function execution is terminated as soon as RETURN statement is reached.

(ii) Calling a function: A function can be called from a select statement, from a procedure or from another function.

Eg

```
SELECT empno, ename, CalcSal (sal, comm)
FROM emp;
```

(iii) Drop a function:

```
DROP FUNCTION [IF EXISTS] function_name;
```

——— x ——— x ——— x ———

★ MySQL Cursor

With cursor we can iterate over a set of rows returned by a query or a process.

MySQL Cursors are Read Only, Non scrollable and Asensitive.

Read Only - Table data cannot be updated using cursor

Non scrollable - Cannot skip, jump or traverse rows in reverse order.

Asensitive - Modifications in actual table are not reflected in the cursor. It points to actual data.

__/__/____

Syntax:

```
DECLARE finished INT DEFAULT 0;
DECLARE cursor-name CURSOR FOR Select-statement;
DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET finished = 1;
OPEN cursor-name;
label1 : LOOP
        FETCH cursor-name INTO variable-list;
        IF finished = 1 THEN LEAVE label1;
        END IF;
END LOOP;
CLOSE cursor-name;
```

Note: The variable-list within FETCH statement must match the columns present in CURSOR's Select-statement.

★ TRIGGERS:

A trigger is a stored block of program that is automatically invoked in response to certain events.

MySQL supports triggers on INSERT, UPDATE or DELETE.

SQL defines various type of triggers like: Database level, table level / statement level, row-level. However MySQL supports only row-level triggers.

Row-level triggers / DML triggers are activated for each row that is inserted / updated / deleted.

__/__/____

(i) Show Triggers:

```
SHOW TRIGGERS FROM db-name;
```

(ii) Create Trigger:

```
CREATE TRIGGER trigger-name
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }
ON table-name FOR EACH ROW
BEGIN
    --
END $$
```

The trigger body can access the values of columns being affected using NEW and OLD modifiers to distinguish between column values BEFORE and AFTER the DML was fired.

(iii) Drop Trigger:

```
DROP TRIGGER [IF EXISTS] db-name.trigger-name;
```

★ Error Handling.

```
DECLARE Action HANDLER FOR condition statements;
```

Action accecpts two values:

1. CONTINUE - the execution of enclosing block continues
2. EXIT - terminates the execution of enclosing block, which contains the handler.

__/__/____

condition - It represents the particular scenario that activates the handler.

The condition value can accept one of the following:

1. A MySQL error code.
2. SQLSTATE Sqlstate-value
   ↳ 5-character string code.
3. NOT FOUND
4. SQLEXCEPTION

If the condition value matches, MySQL will execute the statement and continue/exit the code block.

★ Normalization:

Normalization is the process of organizing/structuring the data such that it reduces data redundancy and improves the data integrity. Without normalization we face issues such as:

Insertion anomaly = Cannot insert data into a table without the presence of other attribute.

Update anomaly = Partial updates and data inconsistency.

Deletion anomaly = Certain attributes are lost because of deletion of other attributes.

1st Normal Form: It tackals the problem of atomicity. ie. each cell should hold a single value.

Ex. A user having multiple phone numbers.

2nd Normal Form: All non-key attributes should be fully dependent on all primary key. The table should not contain any partial dependency.

__/__/____

Example for 2NF:

EmpID	Name	JobCode	Job	StateCode	HomeState
E001	Alice	J01	Chef	26	Michigan
E001	Alice	J02	Waiter	26	Michigan
E002	Bob	J02	Waiter	56	Wyoming
E002	Bob	J03	Bartender	56	Wyoming
E003	Alice	J01	Chef	56	Wyoming.

The above table is in 1NF.
EmpID and JobCode together form a composite PK.
Similarly EmpID & Job form a composite PK.

Name, HomeState and StateCode have a dependency on EmpID = Partial Dependency
↑(EmpID is a part of composite PK).

Similarly Job has a partial dependency on JobCode.

After applying 2NF:

employees table

EmpID	Name	StateCode	HomeState

jobs table

JobCode	Job

employee-jobs (relational-table).

EmpID	JobCode

Note: 2NF is required to be checked only if our table has a composite primary key.

__/__/____

3rd Normal Form: Our table should not have any transitive dependency for any non-prime attribute.
which means a non-prime attribute should not be dependent on any other non-prime attribute.

$X \rightarrow Z$ indirectly, by the virtue of:
$X \rightarrow Y$, $Y \rightarrow Z$
(prime attribute → non-prime → non-prime).

In the previous example after applying 2NF we got the following employees table.

EmpID	Name	StateCode	Homestate

In this table, EmpID determines HomeState via StateCode. That implies we have a transitive functional dependency, & hence does not satisfy 3NF.

After applying 3NF:

employees table

EmpID	Name	StateCode

homestates table

StateCode	HomeState

★ Denormalization: It is an optimization technique.
It is the process of adding redundant data to one or more tables to avoid costly join operations.
Example Courses and Teachers.

__/__/____

★ ACID Properties in DBMS: In order to maintain consistency in database, before and after the transaction, ACID properties must be followed.

Automicity = Entire transaction takes place as one single unit or does not happen at all.

Example: Transfer of funds between two bank accounts.

The process of withdrawal from first account & deposite in second account must take place as a single transaction.

Consistency = The database must be consistent before & after the transaction. It refers to the correctness of data. (Before & after totals should be consistent).

Example:

Before:	T_1	500	T_2 300	= 800
		Transaction		
		- 100	+ 100	
After:	T_1	400	T_2 400	- 800

Isolation: Ensures that multiple transactions can occur concurrently without leading to inconsistency in database state.

Changes occuring due to one transaction are not visible to other transaction happening in parallel, until the transaction has been committed.

Durability: The changes of a successful transaction are stored and written to a disk & they persist even if system failure occurs.

__/__/____

★ Horizontal and Vertical scaling:

Scaling = Altering the size of a system.

Vertical scaling = Adding more resources to existing system, by increasing its computational power (CPU, RAM).
It is a cheaper option as compared to Horizontal scaling.

Horizontal scaling = Adding new servers to existing system

★ Sharding

Sharding is horizontal partitioning the database, which means replicating the schema and dividing the data accross multiple servers to spread the load.

Every distributed table has exactly one shard key, which is used to determine in which partition a given row in the table is stored.

The shard key is computed based on a hash function executed over the values in the column.

Advantages: 1) Reduces the index size
2) Data of one single table can be distributed accross multiple servers
3) Data can be segmented geographically.

★ Data Abstraction in DBMS:

Abstraction is the process of hiding irrelevant details from the user, to make the system efficient in terms of retrival of data and to reduce the complexity in terms of usability of user.

__/__/____

There are 3 levels of abstraction:

1. Physical = This is the lowest level of abstraction. It tells us how the data is actually stored in memory.
   Information like the access methods being used are sequential or random, the organization of files are done using B+ trees or hashing, etc. can be achieved.

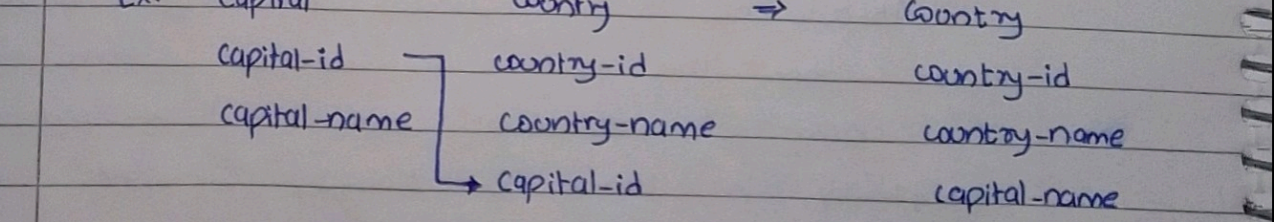
2. Logical = It comprises of the data which is stored in the database in form of tables or documents. It also gives information about the relationship between the entities.

3. View = Highest level of abstraction. Only a part of database is viewed by the user. User views the data in form of rows and columns.

★ Types of Entity Relationships:

1. A one-to-one relationship has one record on each side of the relationship. The primary key from one table relates to at most one entry from another table, making the foreign key unique.
   Note: Combining the two tables having one-to-one relationship does not break any normalization rule.

Ex.



__/__/____

2. In one-to-many relationship, a single entry on one side corresponds to one or more entries on the other end. In this case the foreign key is not unique column.

3. Many-to-many relationships in database have multiple records on both of the ends of relation. A association table with foreign keys from both sides is created.