What is a data base?

- It is a collection of data and a method for us to access and manipulate that data.
- Data is very valuable, we produce data every day. ~ 2.5 quintillion bytes of data daily.

data base

→ It is a computer

→ The cylinder shape is due to the drum memory where data was stored before SSD.

Data scientists → analyze data

Data engeneers → setup databases

Query:

A call to data-base to get desired information. SQL is used to call the data-base.

Database Management Systems: (DBMS)

It is a software program to manage the database.

Relational DBMS: (Most used type)

Same thing as DBMS but in a relational way between entities.

Type of database model:

Relational

Document (mongoDB), (firebase)

Key value (DyamoDB), (Radice)

Graph

Wide Columnar

What is SQL?

→ declarative language, case-insensitive
→ set-based processing

A programming language to talk to the database to get output. It is a declarative language.

wildcard (all information)

$\underbrace{\text{SELECT} * \text{FROM USERS}}_{\text{SQL statement also called Query}}$ → (this table)

```
SELECT  NAME
FROM    USERS
WHERE   ROLE = 'MANAGER' ;
```

identifier

expression

SQL statement

WHERE: keyword

ROLE = 'MANAGER': condition

Relational Database:

- **Tables:**

Representation of an object containing all it's attributes.

(Table) → primary key foreign key

ID	firstName	LastName	Sex	DOB	M ID
1	Buchan	BMW	other	--	m1
2	Cheema	Ronaldo	Male	--	m2
3	Zayn	B section	Male	--	m3
4	Hanzala	Lesbo	Female	--	m1

tuple/rows

columns/attributes

Degree: collection of colums

Domain: The type of data that can be stored in a column. i.e for date of birth column, only dates of format/type date can be stored. It is also called constraint.

Tuple: Tuple is a single row of data. It is also reffered as row.

Cardinality: Collection of rows/tuples

Primary key: Something that uniquely identify your data.

Foreign key: Give reference of primary key.

OLTP vs OLAP

Feature	OLTP	OLAP
Full form	Online transaction processing	Online Analytical processing
Purpose	Handles real-time transactions	Supports analytical queries and decision-making.
Users	Front-end users (cashiers, app users)	Data analysts
Operations	CRUD operations	Complex SELECT queries with aggregations.
Data Type	Higly normalized (relational)	De-normalized, star/snowflake schema
Query Speed	Fast for simple queries	Fast for complex queries
Data Volume	Handles small transactions	Deals with large volume of historical data.
Database Design	Transactional DBs (MySQL, PostgresQL)	Data warehouses (snowflake, Redshift)
Response Time	Milliseconds to seconds	Seconds to minutes
Backup and Recovery	Frequent due to real time data	Periodic, based on batch ETL jobs
Data Integrity	Very high (ACID compliance)	Not always ACID

SQL Commands

- **DDL** — data definition language
  → helps in setting up database
  - Create
  - Alter
  - Drop
  - RENAME
  - TRUNCATE
  - COMMENT
- **DQL** — data Query language
  - SELECT
    ↳ used to retrieve data
- **DML** — data modification language
  - INSERT
  - UPDATE
  - DELETE
  - MERGE
  - CALL
  - EXPLAIN PLAN
  - LOCK TABLE
- **DCL** — Data control lang
  → manages access
  - GRANT
  - REVOKE

Aggregate Functions:

→ They give a single output from large data input.

AVG() → calculates the average of a set of values

COUNT() → counts rows in a specified table

MIN() → get minimum value

MAX() → get maximum value

SUM() → calculates sum of values

Operator Precedence:

Parentheses

↓

Multiplication/division

↓

Subtraction/~~division~~ addition

↓

Not

↓

AND

↓

OR

Checking for Null values:

Null = Null -- Null

SELECT NULL = NULL - returns Null

→ doing any operation with null will return Null.

→ every field is Nullable by default.

→ It might be optional or required

→ Always check Nulls. ↳ future data

→ Nulls can be detected using 'IS' operator.

'IS' operator

- ↳ IS NULL
- ↳ IS NOT NULL
- ↳ IS TRUE
- ↳ Is FALSE

SELECT * FROM users WHERE age = 20 IS FALSE ;

NULL VALUE SUBSTITUTION:

→ Ability to replace NULL values to operate on the data.

This can be done using the keyword:

Coalese (<column>, 'EMPTY')

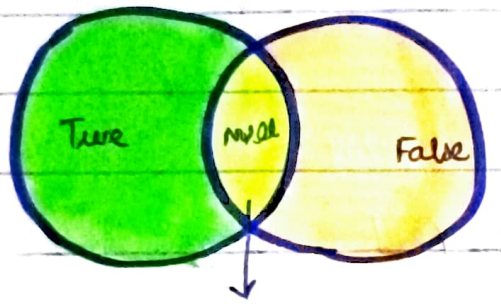
↳ when Null replace with

↳ It returns first Null value in a list

→ We can give multiple colums to coalesce();

Three-Value Logic:

→ Besides True and false, the result of logical expressions can also be unknown.



unknown → It could be anything in SQL

(NULL = 1) OR (1 = 1) -- TRUE

(NULL = 1) AND (0 = 1) -- NULL (always)

BETWEEN AND:

↳ shorthand to match Against a range of values.

--WHERE <column> BETWEEN X AND Y

↳ It is sensitive to the order of arguments

IN :

↳ Filtering Multiple values without many AND statements.

↳ Check if a value maches any values in a list of values.

```
WHERE emp_no IN(100001, 100006, 11008);
```

Partial Lookups:

↳ What if you don't know exactly what are you searching for?

↳ Can be done using **LIKE**

```
WHERE first_name LIKE 'M%';
```

↳ gives all names that starts with M.

Pattern wildcards	Meaning
%	Any number of characters
_	1 character

Use cases	Meaning
LIKE '%2'	Fields that end with 2
LIKE '%2%'	have 2 anywhere in the value
LIKE '_00%'	have 2 0's as second and third char
LIKE '%200%'	have 200 anywhere in value
LIKE '2_%_%'	starts with 2 but atleast 3 char long
LIKE '2___3'	five-digit number that starts with 2 and end with 3

⇒ In postgres 'LIKE' only does TEXT comparison so we must CAST the thing into text.

⇒ Casting can be done in two ways

① CAST (Salary AS text);
② Salary :: text

LIKE keyword can be used for case insensitive matching.

name ILIKE 'M%';

Date Filtering:

↳ data dealing is tuck due to GMT and UTC. We should always use UTC.

→ sets timezone to UTC for current session only

```
SET TIME ZONE 'UTC';
```

We can see our current time zone by:

```
SHOW TIMEZONE;
```

postgres uses ISO-8601

YYYY-MM-DDTHH:MM:SS

→ date

Time stamps = (Time + timezone info)

Date Operators:

To get current date:

1. `SELECT NOW()::date;`
2. `SELECT CURRENT_DATE;`

To get current date in custom format:

```
SELECT TO_CHAR(CURRENT_DATE, 'dd/mm/yyyy');
```

⇒ Refer to official documentation for more formating options.

To get difference between dates:

```
SELECT NOW() - '1800/01/01';
```

We can calculate Age from date

```
SELECT AGE( date '1800-01-01');
```

↓ casting string to date

```
SELECT AGE(
    date '1992/11/13', date '1800/01/01');
```

↓ returns yyyy, mm, dd between these two dates

Extracting information:

↳ Using EXTRACT keyword.

```
SELECT EXTRACT(DAY FROM date '1992/11/13') AS DAY
```

→ similar statement to get year and month.

To round the date to a specific time:

```
SELECT DATE_TRUNC('Year', date '1992/11/13');
```

① For 'year'; it will change month and day to lowest.

② For 'month'; it will change only day to lowest

③ For 'day'; it will not change anything unless it is a timestamp.

DISTINCT keyword:

↳ It removes duplicates

↳ keeps single row for same data.

```
SELECT DISTINCT salaries FROM salaries;
```

Sorting Data:

↳ ORDER BY (keyword)

```
SELECT * FROM customers
ORDER BY <column> [ASC/DESC]
```

Also, we can sort by length of string

```
SELECT * FROM Employees
ORDER BY LENGTH(firstName) DESC;
```

Join:

↳ It allows us to select multiple Tables by linking two tables.

INNER JOIN	Natural Join
SELF JOIN	Conditional Join
RIGHT JOIN	Outer Join
LEFT JOIN	Cross Join

Natural Join: (INNER JOIN)

↳ cross product + condition

⇒ check table name from which result is expected.

→ ⇒ used when some attribute is common and equalence is to be checked.

Natural join vs inner join

→ always prefer inner join

→ Do the same work in some cases

→ Inner join returns the intersection of two tables.

Table A			Table B	
emp_no	first_name	last_name	emp_no	salary
....	....	....	....	....
....	....	....	....	....

→ Inner join can be used using 'join' keyword or WHERE Filter but prefer JOIN.

→ All results are returned unsorted.

↳ use ORDER BY to sort.

```
SELECT a.emp_no,
       CONCAT(a.first_name, a.last_name) as "name",
       b.salary FROM employees as a
       INNER JOIN salaries as b ON
       b.emp_no = a.emp_no;
```

→ You have to decide from which you are trying to get and what are you trying to join.

SELF-JOIN:

↳ Joining a table with itself.

→ It is usually done when a table has a foreign key referencing its primary key.

id	Name	SDate	superID
1	Inventix	2000/01/13	1
2	Ali	1995/02/7	1

→ select from the same table twice.

```
SELECT a.id, a.name as "employee",
b.name as "supervisor name"
FROM employee as a, employee as b
WHERE a.supervisorID = b.id;
```

→ It returns the supervisor name for each employee.

OUTER JOIN:

↳ Returns the Rows that don't match

There are two types of outer join.

LEFT JOIN → Adds data from table that is from "FROM" table.

RIGHT JOIN → Adds data from the right table.

```
SELECT *
FROM <table A> As a
LEFT [OUTER] JOIN <table B> AS b
ON a.id = b.id
```

→ It returns the intersection same as INNER JOIN and the data from table a.

→ Some rows may return NULL where there is no data available at b for a.

→ Any value that does not match is made to be NULL.

CROSS JOIN:

↳ creates a combination of every row.

→ performs cartesian product on tables rows.

It is not that common in bussiness cases

→ It returns a very large dataset.

```
SELECT * FROM <table a>
CROSS JOIN <table b> ;
```

FULL OUTER JOIN:

↳ Returns results from both tables whether they match or not.

It is a less common type of JOIN.

```
SELECT * FROM <table a> AS a
FULL JOIN <table b> AS b
ON a.id = b.id ;
```

→ Things that doesn't match, returns NULL on both tables.

USING keyword:

↳ simplifying the join syntax

→ Use when you are joining same columns from different tables.

```
SELECT e.emp_no, e.first_name, de.dept_no
FROM employees AS e
INNER JOIN dept_emp AS de USING(emp_no);
```

GROUP BY → strict rules

↳ Summarizing or aggregating data by groups

→ It splits data into groups or chunks so we can apply functions against the group rather than the entire table.

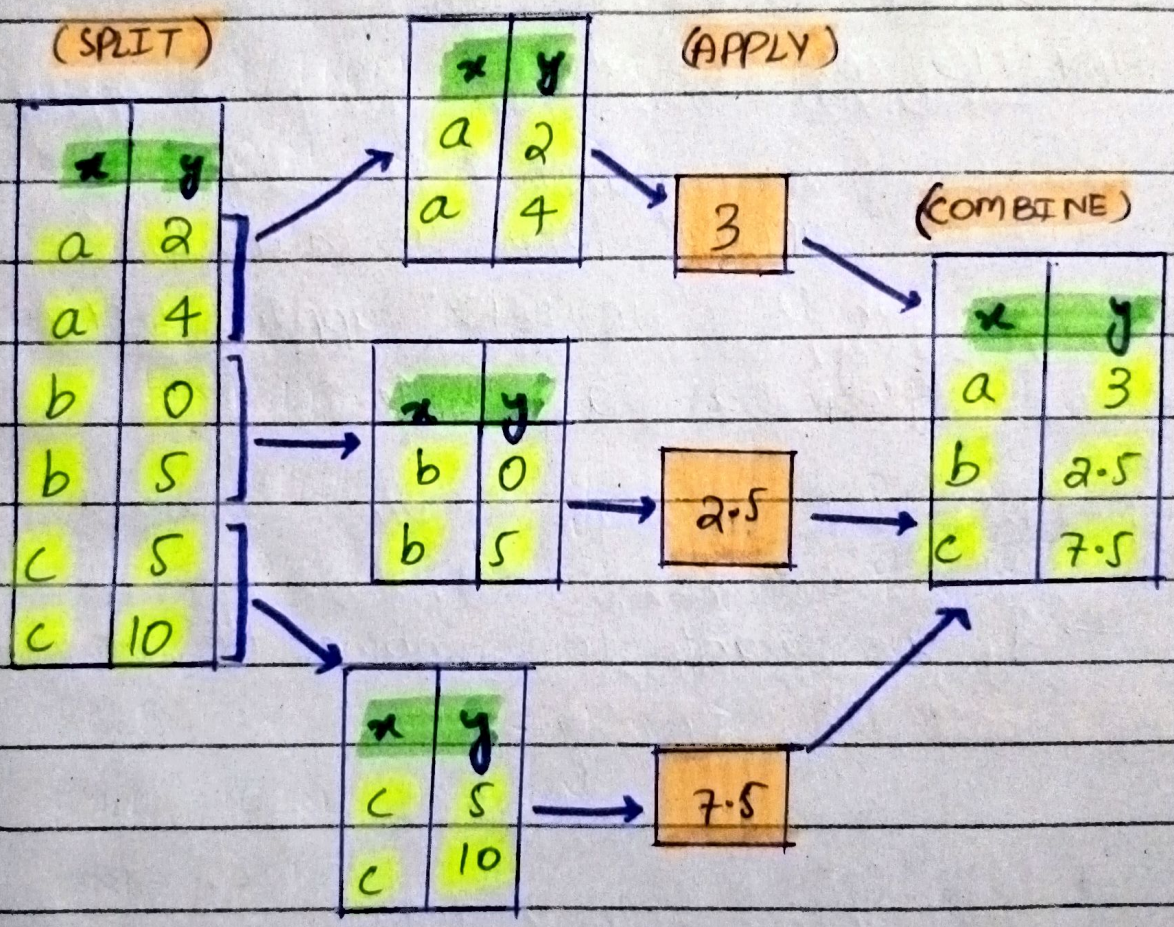
```
SELECT dept_no
FROM dept_emp
GROUP BY dept_no;
```

→ We use group by almost exclusively with aggregate functions.

→ Every column not in the GROUP-BY clause must apply a function

→ Group-BY reduces all records found for the matching "GROUP" to a single record.

→ It uses "split-apply-combine" strategy.



HAVING :

↳ If we want to filter groups

```
SELECT col1, COUNT(col2)
FROM <table>
WHERE col2 > X
GROUP BY col1
HAVING col1 === Y;
```

→ "WHERE" applies filter to individual rows while "HAVING" applies filters to a group as a whole.

GROUPING SETS:

↳ If we want to combine the results of multiple Groupings.

→ **UNION**

↳ SELECT Multiple SELECT statements.

```
SELECT col1, SUM(col2) FROM <table>
GROUP BY col1
UNION
SELECT SUM(col2) FROM <table> ;
```

→ **UNION ALL:**

↳ Difference between "UNION" and "UNION ALL" is that 'UNION ALL' does not remove duplicate records.

→ **GROUPING SETS ( ).**

↳ Works same as union but with single select statement.

→ Should be written after GROUP-BY.

ROLLUP:

↳ It simplifies the grouping sets by listing all combinitions from the given group.

Window Functions:

↳ creates a new column based on functions performed on a subset or "WINDOW" of the data.

```
window_function (arg1, arg2, ...) OVER (
[PARTITION BY partition_expression]
[ORDER BY sort_expression [ASC|DESC]
[NULLS {FIRST | LAST}]
```

→ OVER() is a function that defines the window data. It can be influenced by partition and ORDER BY.

→ Large window data results in longer execution time.

→ **Partition BY:**

↳ Divide rows into groups to apply the function against (optional).

AVG(s.salary) OVER() as "average global salary"

→ ORDER BY changes frame of window function. It behaves differently in windows function.

FRAME CLAUSE:

↳ When using a frame clause in a window function we can create a sub-range or frame.

KEY	MEANING
Rows or Range	Whether you want to use a range or rows as a frame
Preceding	Rows before the current one
Following	Rows After the current one
Unbounded preceding or following	Returns all before or after
Current Row	Your current Row

PARTITION BY category ORDER BY price RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

→ Without ORDER BY, the default framing is usually all partition Rows.

CONDITIONALS :

↳ When you want to select something when a certain criteria is met.

→ CASE STATEMENT :

```
SELECT a,
        CASE WHEN a = 1 THEN 'one'
             WHEN a = 2 THEN 'two'
             ELSE 'other'
        END
FROM test;
```

→ case statements can be used in multiple places in a Query.

→ Each return must be a single output.

Filtering in a WHERE :

```
SELECT
    o.orderid,
    o.customerid,
    o.netamount
FROM orders as O  WHERE  CASE
WHEN o.customerid >10 THEN o.netamount < 100
ELSE o.netamount > 100  END
ORDER BY o.custer id;
```

→ In an aggregate function:

```
SELECT
    SUM (
        CASE
            WHEN o.netamount < 100
            THEN -100
            ELSE o.netamount
        END
    ) as "returns",
    SUM(o.netamounts) as "normal total"
    FROM orders as o;
```

NULL IF :

↳ If you want to return NULL if a condition is met.

```
NULLIF(val_1 , val_2)
```

val_1 = Val_2

```
NULLIF (0, 0)          -- NULL
NULLIF ('ABC', 'DEF')  -- ABC
```

↳ left value is returned

Use: Fill to avoide division by zero.

VIEWS :

↳ If you want to store the results of a Query.

→ If you want to Query the results of a query.

→ Views are the output of Query we ran.

There are two types of views

Materialized

Non-materialized

→ NON-materialized views :

↳ Query gets re-run each time the view is called on

→ Reference to the Query.

→ Materialized views :

↳ Stores the data physically on disk and periodically updates it when tables change.

VIEW Syntax:

To create a view:

CREATE VIEW <name> AS query;

→ Views act like tables, so we can also Query them.

→ Views take very little space to store. We only store the definition of a view not all of the data that it returns.

↳ It is for non-materialized only.

To update a view:

CREATE OR REPLACE <name> AS query;

To Rename a view:

ALTER VIEW <name> RENAME TO <name2>;

To delete a view:

DROP VIEW [IF EXISTS] <name> ;

Using VIEWS:

↳ (eg) getting the most recent salary of an employee.

```
SELECT * FROM salaries
JOIN last_salary_change AS l USING(emp_no)
WHERE from_date = l.max ORDER BY emp_no;
```

INDEXES:

↳ An index is a construct to improve querying performance.

→ It is a pointer to data in a table.

* Think it like table of contents.

→ It speed up Queries

→ It slows down data insertion and updates.

→ Poor indexing kills the performance.

Some types of indexes

(i) Single-column

(ii) Multi-column

(iii) Unique

(iv) partial

(v) Implicit Indexes

To create an index

CREATE [UNIQUE] INDEX <name>
on <table> (colm1, colm2, ...);

To delete index
DROP INDEX <name>

→ **When to use?**
→ Index foreign keys
→ Index primary keys and unique colums
→ Index on columns that end up in the order BY/WHERE caluse often.

→ **When not to use?**
→ Don't add an index just to add an index
→ Don't use indexes on small tables
→ Don't use on tables that are updated frequently
→ Don't use on colums that can contain NULL.
→ Don't use on columns that have large values.

→ **Single-column Indexes**

↳ used for most frequently used columns in a Query.

→ **Multi-column Indexes**

↳ used for more frequently used columns in a Query

→ Single-column Index retrives data that satisfies ~~multiple~~ single conditions.

→ Multi-column Index retrives data that satisfies multiple conditions

→ **Unique Index:**

↳ for speed and integrity

→ use for columns with unique values like primary keys.

→ **Partial Index:**

Index over a subset of a table.

```
CREATE INDEX <name>
on <table>(<expression>);
```

→ **Implicit Indexes:**

↳ Automatically created by the database

- primary key
- unique key

INDEX Algorithms:

PostgreSQL provides several types of indexes:

(i) B-Tree

(ii) Hash

(iii) Gin

(iv) Gist

Every index type uses a different algorithm

```
CREATE [UNIQUE] INDEX <name>
ON <table> USING <method> (colm1, ...);
```

→ **B-Tree**

↳ default algorithm.

→ Best used for comparisons with

<, <=, =, >=, BETWEEN, IN, ISNULL

→ **Hash**

↳ can only check (handles) equality (=) operations

→ **Gin**

↳ (Generalized inverted Index)

→ Best used when multiple values are stored in a single field.

→ **Gist**

↳ (Generalized search Tree)

→ useful in indexing geometric data and full-text search.

SUBQuery:

↳ A construct that allows you to build extremely complex Queries

→ It is also called an inner Query or inner select.

→ A subQuery is most often found in the WHERE clause.

```
SELECT * FROM <table> WHERE <column>
<condition> ( SELECT <column>, .... FROM
            <table> [WHERE | GROUP BY | ORDER BY]
        ... ) ;
```

→ It cannot return multiple columns in WHERE caluse.

→ We can also use it in the ~~select~~ SELECT, FROM and HAVING clause.

→ When a subquery is used in a SELECT/HAVING clause, it must return a single record.

SUBQUERY VS JOIN

→ Both subquery and joins combine data from different tables

→ Subqueys are queries that can stand alone

→ Joins combine rows from one or more tables based on a match condition.

→ Subquery can return a single result or a row set.

→ Joins can only return a row set.

→ Subquery results are immediately used

→ A Joined table can be used in the outer query.

Guidelines:

→ A subquery must be enclosed in parentheses.

→ It should be (must) placed on the right side of the comparison operator.

→ We cannot manipulate their results internally. (ORDER BY ignored)

→ Use single-row operators with single-row subqueries.

→ Subqueries that return NULL may not return result.

→ Types of subQueries:

(i) Single Row

(ii) Multiple Row

(iii) Multiple Column

(iv) Correlated

(v) Nested

→ Single Row

↳ Returns zero or one row

```
SELECT name, salary
FROM salaries
WHERE salary =
    (SELECT AVG(salary) FROM salaries)
```

→ Multiple Row:

↳ Returns one or more Rows

```
SELECT title, price, category FROM products
WHERE category IN
(   SELECT category FROM categories
    WHERE categoryname IN ('Comedy', 'Family')
);
```

→ Multiple Column:

↳ Returns one or more columns

```
SELECT emp_no, salary, dea.avg
FROM salaries JOIN dept_emp USING(emp_no)
JOIN (
        SELECT dept_no, AVG(salary) FROM salaries
        JOIN dept_emp USING (emp_no)
        GROUP BY dept_no
        ) USING (dept_no)
WHERE salary > dea.avg ;
```

→ **Correlated**

↳ Reference one or more columns in the outer statement

→ It runs against each row.

```
SELECT emp-no, salary, from_date
FROM salaries AS s
WHERE from_date = (
    SELECT max(s2.from_date) AS MAX
    FROM salaries AS s2
    WHERE s2.emp_no = s.emp_no
) ORDER BY emp_no;
```

→ **Nested Sub-Queries:**

↳ It contains subquery inside of a subquery

```
SELECT .....
    ( SELECT .....
        ( SELECT (....) ) );
```

Operators that can be applied on subqueries in WHERE clause

→ EXISTS

↳ check if the subquery returns any rows

→ IN / NOT IN

↳ check if the values is equal (not equal) to any of the rows in the Return (NULL yields NULL)

→ ANY / SOME

↳ check each row against the operator and if any comparison matches, Return TRUE.

→ ALL

↳ check each Row against the operator and if all comparisons match, Return TRUE

→ All single value operators, operates on single row.

* DataBase Management

There are two types of database
→ Regular (→ dB created based on template)
→ Template

→ When we create dB with postgress, three databases are created:
(i) postgres
(ii) Template 0
(iii) Template 1

To create postgress Database

~ psql -U <user> <database>
→ If we don't give database name, postgres will consider it same name as of user.
→ Always give database name

→ **Template 0:**
↳ This is the template that is used to create Template1 (like a backup)
→ Never change it.

→ **Template 1:**

↳ This is the template that is used to create new databases.

→ Any changes maked in template1 will apply those changes automatically to all new databases.

→ It is the default template.

→ If template1 is being accessed, no new databases can be created until the connection is closed.

* To check your connectivity status:

\conninfo (command)

→ We can also create our own templates to create other database.

```
CREATE DATABASE my;
CREATE TABLE <table>;
CREATE DATBASE <name> WITH
TEMPLATE my;
```

How to create a DataBase:

```
CREATE DATABASE <name>
  [ [WITH] [OWNER [=] user-name]
      [TEMPLATE [=] template]
      [ENCODING [=] encoding]
      [LC_COLLATE [=] lc-collate]
      [TABLESPACE [=] tablespace]
      [CONNECTION LIMIT [=] connlimit] ]
```

Setting	Default
TEMPLATE	template01
ENCODING	UTF8
CONNECTION_LIMIT	100
OWNER	current user

Database Organization:

↳ postgress offer the concept of "schemas". It is used to organize tables, views, indexes etc.

→ By default each database gets a "public" schema.

e.g

```
SELECT * FROM employees
SELECT * FROM public.employees
```

-- same

schema notation

To see all schemas in database, write

\dn (command)

→ It also show who is the owner of the schema.

→ Reasons to use schemas

→ To allow many users to use one database without interfering with each other.

→ To organize database objects into logical groups to make them more manageable.

→ Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

Roles:

↳ Roles are vital to any DBMS. They determine what is allowed.

→ Not everyone is allowed to create a database

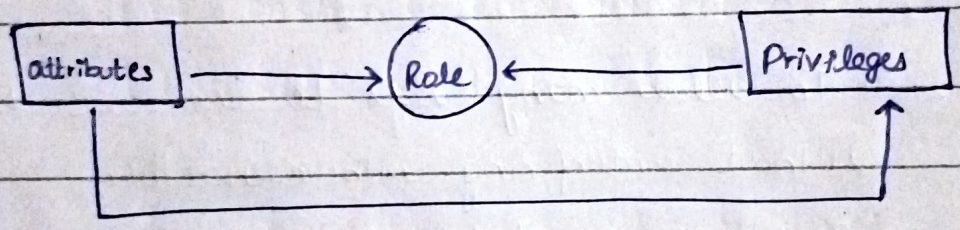
→ A role can be a user or a group. It depends on how you setup the role.

→ Roles have the ability to grant membership to another roles.

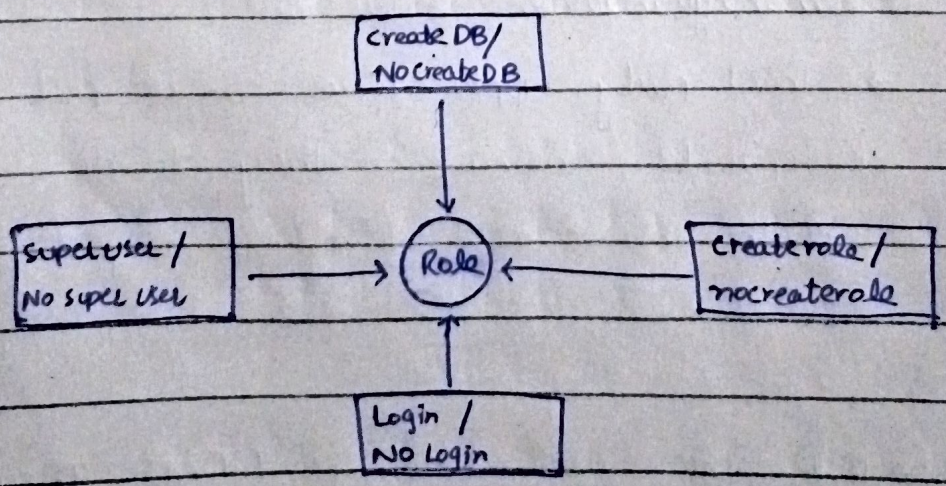
Role Attributes:

↳ When a role is created, it is given certain attributes.

→ The privileges of a role are determined in part by its attributes



→ **Attributes:**



Common Attributes

Attributes	Purpose
LOGIN PRIVILEGE	A role with the Login attribute can be considered the same thing as a "database user".
SUPERUSER STATUS	A database superuser bypasses all permission checks
DATABASE CREATION	A role must be explicitly given permission to create databases.
ROLE CREATION	A role must be explicitly given permission to create ~~data~~ more roles.
PASSWORD	A password is only significant if you give LOGIN privilege.

→ **Creating Role:**

```
CREATE ROLE readonly WITH LOGIN
ENCRYPTED PASSWORD 'readonly';
```

→ Always encrypt when storing a role that can log in.

→ To view all the roles
\du (command)

→ By default, only the creator of the database or superuser has access to its objects.

Create Data: (Tables)

```
CREATE TABLE <name>(
    <col1> TYPE [CONSTRAINT],
    table_constraint [CONSTRAINT] )
    [INHERITS <existing_table>];
```

→ **Constraints:**

↳ they are tools to apply validation methods against data that will be inserted.

→ Column constraints

↳ A column constraint is defined as part of a column definition.

→ They are not applied to multiple columns but the specified one.

NOT NULL	cannot be NULL
PRIMARY KEY	colum will be primary key
UNIQUE	can only contain unique values + NULL
CHECK	Any special condition check
REFERENCES	Constrain the values of the column to only be values that exist in the column of another table (foreign Key)

→ Table constraints:

↳ A table constraint definition is not tied to a particular column, and it can encompass more than one column.

UNIQUE (column_list)

PRIMARY KEY (column_list)

CONSTRAINT pk_student_id PRIMARY KEY (student_id);

→ Every column constraint can be written as a table constraint.

Backup:

↳ Back ups are the copy of database kept to avoid any potential data desasters.

→ Alway backup your data base.

TYPE	PURPOSE
FULL BACKUP	backup all the data
INCREMENTAL	Bakup since last incremental
DIFFERENTIAL	Bakkup since last full backup.
TRANSACTION LOG	Backup of the database transactions.

→ Choosing the type totally depends on the size of database.

How can I make a back up?

You can make a back up on

(i) operating system (logical backup)

(ii) hard disk (physical clone)

(iii) cloud (AWS etc)

(iv) direct copy-paste

→ Frequency of backups depends upon database.

How to backup:

↳ Backup can be done by SQL Dump.

→ A SQL dump generates a text file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump.

Tool → pgBackRest

↳ used to create backup and perform restores.

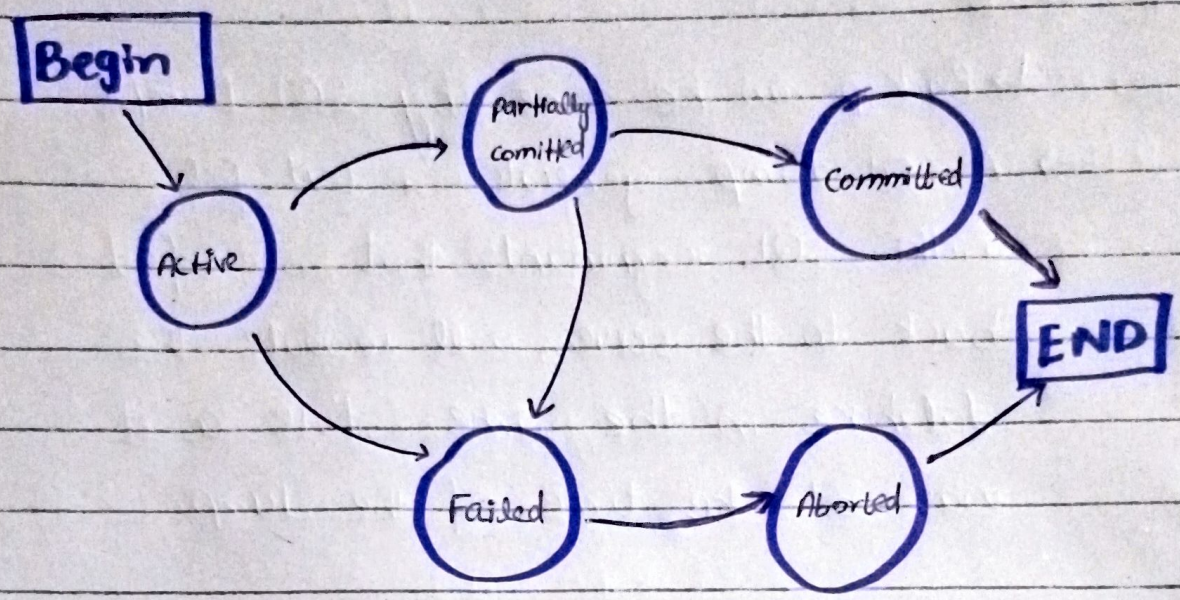
Transactions:

→ A database is a shared resource, so many users may access it concurrently

→ Transaction is a unit of instructions (Query for instance)

→ It is used to keep things consistent.

Lifecycle of transactions:



→ To enter the transaction manually, write 'BEGIN;' in terminal (in database)

→ If you don't want to commit changes, write 'ROLLBACK;'

→ To commit, write 'END;'

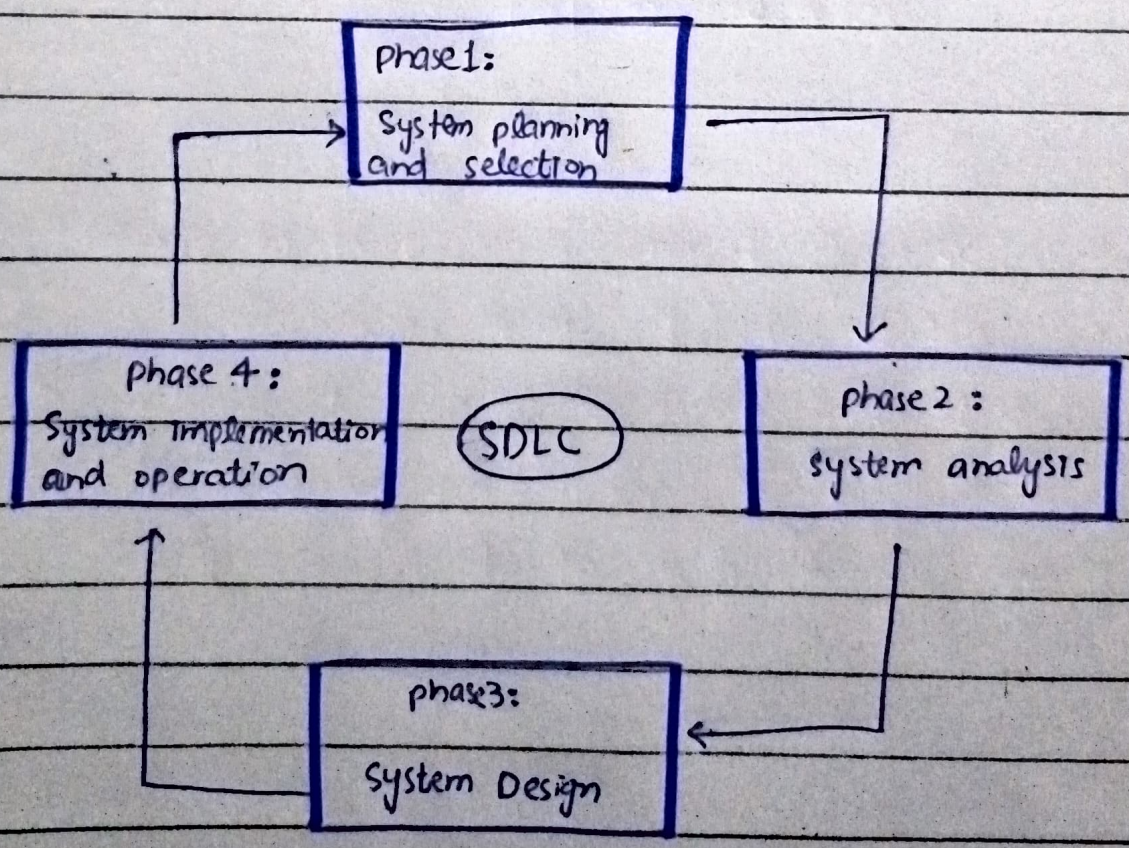
→ To maintain the integrity of a database, All transactions must obey ACID properties.

A (atomicity) → either execute entirely or not at all.

C (consistency) → Each transaction should leave DB in consist-ent state (commit or ROLLBACK)

I (Isolation) → Transactions should be isolated from other transaction

D (Durability) → After completion of a transaction, the changes in the database should presist.

* Data Base and system Design:

SDLC (Software development life cycle)