Complete SQL and Database Bootcamp

LEARNING ROADMAP (Course by ZTM)

1. History of Databases.
2. Database & SQL Basics.
3. Environment Setup.
4. SQL Deep dive.
5. Advanced SQL.
6. Database Management.
7. Database Design.
8. Solving data breach mystery.
9. Database Landscape
10. Data Engineering.
11. Redis

1. History and Story of Data

Data: Facts and statistics collected together for reference or analysis.

Database (DB): It is a computer system (hardware + software) used for collection and management of data effectively.

Database Management System (DBMS): Software used to mange data in a DB.

Relational DBMS (RDBMS): A subset of DBMS that follows a relational model to connect various data in tables.

SQL (Structured Query Language): A query language used as a medium to interact with DB.

Creating data means that data is being captured and stored in a DB.

Why so many DBMS? Due to different requirements by different organizations to store and mange data differently.

Why not use Excel or other files for storing and maintaining data?

- Risk of inaccurate information.
- Has certain data limit.
- Cells cannot be connected to one another.
- Cant manage visual content.
- Multiple people cannot manage it at once.

Why the drum symbol for DB? Data used to be stored in drum shaped hardwares, thus the symbol/logo.

Types of DBMS :

- Relational (e.g. PostgreSQL, MySQL, SQLite)
- NoSQL
  - Key Value (e.g. Redis)
  - Document (MongoDB, couchDB)
  - Wide column (Cassandra)
  - Graph (neo4j)

- Original name of SQL → SEQUEL (Structured English Query Language)
- Edgar Frank "Ted" Codd, wrote paper on relational database model while working for IBM. Later, two programmers also from IBM, Donald Chamberland and Raymond implemented the vision and wrote first version of SQL.
- declarative language → What will happen?
  imperative language → How it will happen?
  SQL is declarative language.

2. Database and SQL Basics

Query: Instruction/SQL statement that asks DB to do something.

SQL is all about quering database. e.g. :- SELECT name FROM "Users";

SQL is very similar to writing normal english, which follows some standards.
We use keywords to do something with DB

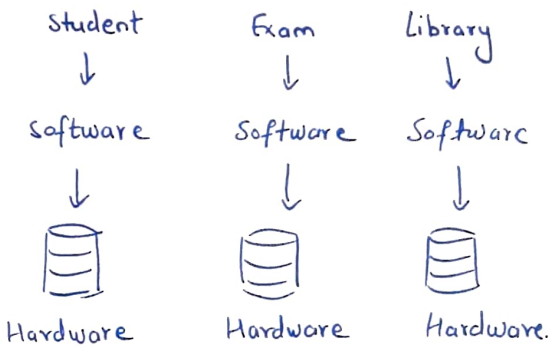
```
                                  identifier.
clause  { SELECT   FNAME
clause  { FROM     USERS
clause  { WHERE    ROLE = 'Manager';
                   \____condition____/
```

keywords :-
SELECT
FROM
WHERE

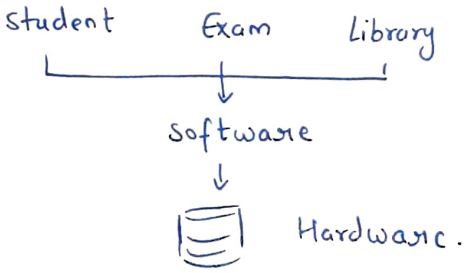
Standards: Rules a language should follow to maintain a version, usable everywhere. Companies put new stuff on top of standards to make their product more appealing.

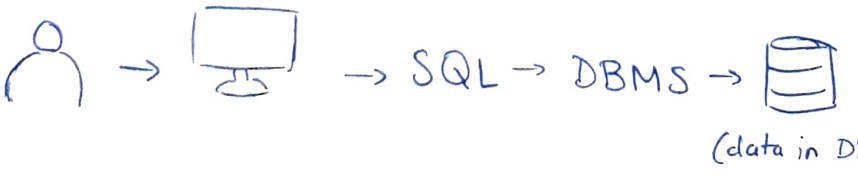
Approaches to store and manage data

1. File Processing Systems
   - No co-relation between files
   - Every system was custom built.
   - User needed to know everything about system.
   - Copy redundant data for reference.



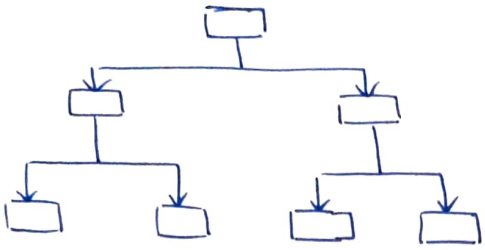
2. Database
   - Data could by related
   - User need not know everything.
   - No redundant data





Database Models : Hierarchial, Networking, Relational, Entity-Relation-ship, Object Oriented, Flat, Semi-Structured, etc.

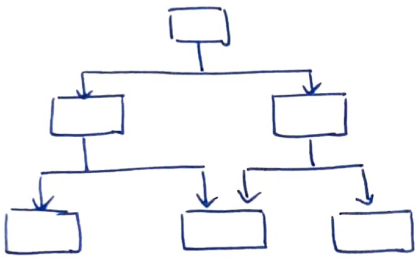
1. Hierarchial Model



- 1 to Many
- deleting parent means deleting child. called tightly coupled data.

Drawback

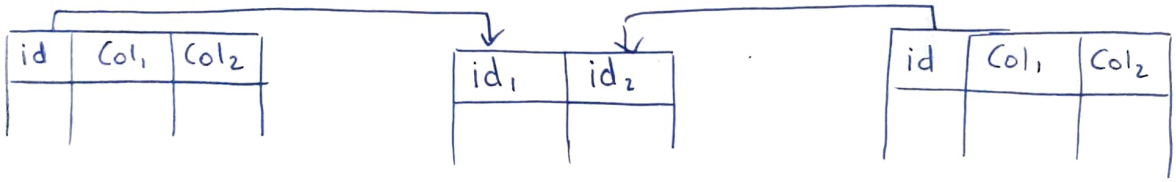
2. Network Model



- Many to Many
- Keep track of relationship to every other data piece.

Drawback

3. Relational Model



DBMS uses a model to store data, i.e., strict set of rules to implement data storage in a particular way.

DBMS can do :

1. CRUD operations (Create, Read, Update, Delete)
2. Manage data (Storage location, Secure authorization, transaction management)

Codd set 13 rules for a RDBMS

RDBMS Terminologies :

1. Column : stores only one type of data.
2. Row : stores single value from each column.
3. Table: collection of rows & columns.
4. Degree : collection of columns.
5. Cardinality: collection of rows.
6. Tuples: Rows (synonym)

~~7. Rows~~

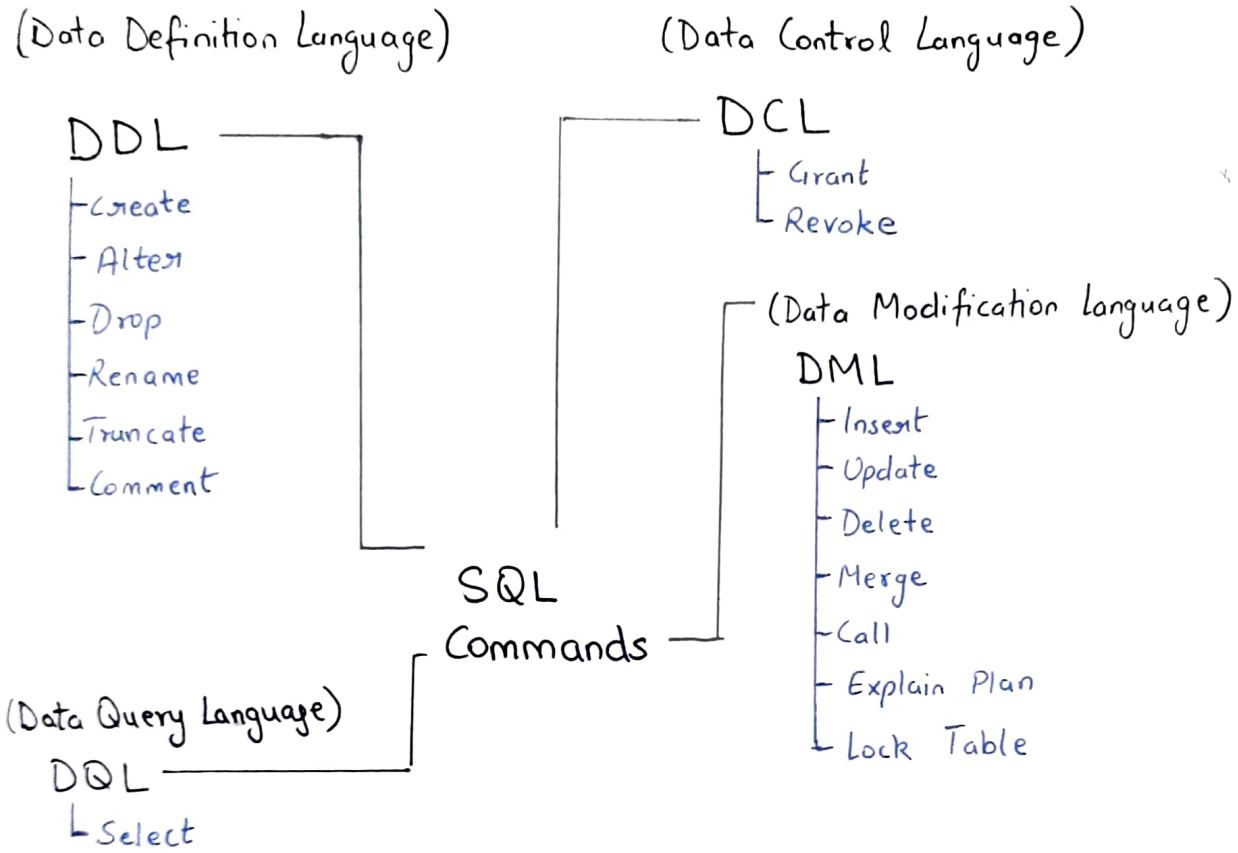
7. Attributes : Columns (synonym)
8. Domain / Constraint : defines: what can be inserted in a column?
9. Primary Key : uniquely identifies a row or piece of data.
10. Foreign Key : reference to primary key of another table.

Uses of RDBMS

1. OLTP (Online Transactional Processing) : Supports day to day activity.

2. OLAP (Online Analytical Processing): Supports data analysis.

4. SQL Deep Dive



4.1. DQL (Data Query Language)

DQL is basically read commands. There is a pre-existing DB, which we need to read. **SELECT** command retrieves data from a table, specifying the colum name or names. Main purpose of variety in SELECT is to be able to analyze data.

SELECT * FROM <table_name>; ← read all columns from a table.

SELECT <Column_Name> FROM <table_name>; ← read only the specified column.

SELECT <col1, col2, col3> FROM <table_name>; ← read only the specified columns.

(;) is used to terminate a SQL statement.

FROM command/keyword specifies the table to read from, as a DB can have multiple tables.

WHERE command is used to filter the rows, so that we read only the necessary data.

options separater

```
SELECT {<col1, col2, col3> | *} ← choose either option
FROM <table_name>
WHERE <column_name> = 'value';
```

DQL will be the focus of this section.

4.2. Functions in SQL and their types

While retrieving columns, we can apply some functions on them which can answer multiple analytical questions.

- Aggregate Functions - Run against all values → 1 output
  AVG(), COUNT(), MIN(), MAX(), SUM() ← works on a single column.

- Scalar Functions - Run against each row → multiple outputs
  UCASE(), LCASE(), MID(), LENGTH(), ROUND(), NOW(), FORMAT() ← works on single column.

Aggregate Function output

Colum Name
Value

Scalar function output

Column Name
value1
valu2
value3
value4
⋮ so on...

4.3. Applying Multiple Filters

As the DB grows, the queries we work on become more complex.

Always have this basic understanding:
a. Where is what ?
b. What do you want ?

4.3.1 The 'AND' Keyword and 'OR' Keyword.

WHERE <condition> AND <condition>; Both values need to match.	SELECT * FROM "Employees" WHERE age > 30 AND gender='M';
WHERE <condition> OR <condition>; Either one needs to match.	SELECT * FROM "Employees" WHERE dept = 'Research' 'OR dept = 'Technology';

WHERE {condition {AND | OR} condition} {AND | OR} {condition {AND | OR} condition};

Using parenthesis to apply a filtering condition on multiple other filtering conditions.

4.3.2 The 'NOT' Keyword.

WHERE NOT <condition> Fetch data that does not match this condition	SELECT * FROM "Employees" WHERE **NOT** dept='Research';

4.3.3 Comparision Operators

>, <, >=, <=, =, !=	SELECT "f_name" FROM "Employees" WHERE age <= 40;

CONCAT() : Combines two columns into 1, also combining their values.

AS : Change name of column while quering.

SELECT CONCAT("col1", "<separater>", "col2") AS "Name_New"

e.g., SELECT CONCAT("first", "_", "last") AS "Full Name"

Comments : Piece of code that doesn't runs and simply serves as notes.

-- Single line Comments

/* */ Multi line comments.

Common SELECT Mistakes

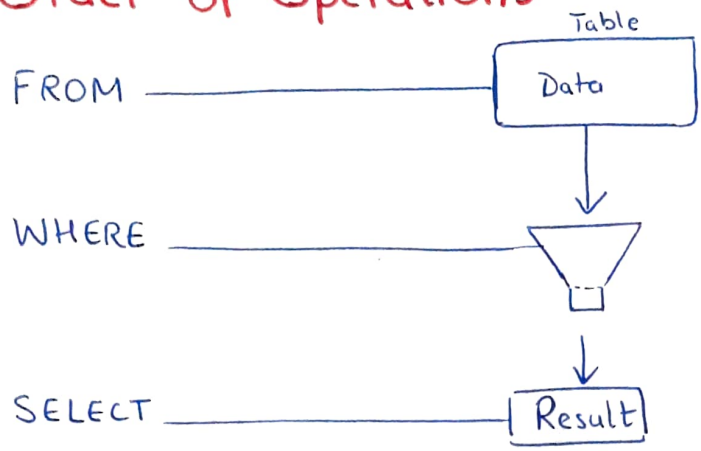
1. Misspelling commands.
2. Using (;) instead of (,) or vice-versa.
3. Using ("") instead of ('), or vice versa.

   Double Quotes for column name.

   Single Quotes for value.
4. Invalid column name.

NOTE: SQL is case insensitive.

4.3.4 Order of Operations



4.3.5 Operator Precedence

A statement having multiple operators is evaluated based on the priority of operators.

::
[ ]
.
-
^
*, /, %
+, -
IS
ISNULL
NOTNULL
User-Defined
IN
BETWEEN
OVERLAPS
LIKE, ILIKE
<, >
=
NOT
AND
OR

Decreasing order of precedence

4.3.6 Checking for Empty Values

- When a record does not have a value, it is called **NULL**.
  It is different from 0 and spaces.
  By default, all fields are nullables.
  All operations on NULL will result in NULL.

- To check if value is NULL or not, we use these operators

  IS NULL
  IS NOT NULL

  Is acts like = (equal to) sign.
  Used with WHERE

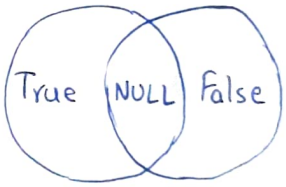
- To replace NULL values, use coalesce() function:

```
SELECT COALESCE(<column>, 'replacing_value')
```

We can also nest functions:

```
SELECT AVG(COALESCE(age, 0))
```

<u>Three valued logic</u>. Besides True or False, the result of a logical expression can also be unknown, i.e., NULL



- SQL NULL could be anything.
- Nothing equals NULL, not even NULL, each NULL can be different.

4.3.7 BETWEEN Command

It is a shorthand to filter data by matching against a range of values.

WHERE <column> BETWEEN 'val1' AND 'val2'; (inclusive range)

↓ equivalent to

WHERE <column> >= 'val1' AND <column> <= 'val2';

4.3.8 IN Command

To check if a data record matches any value in a list of values. Used to filter multiple values without writing endless AND statements.

WHERE <column> IN ('val1', 'val2', 'val3', 'val4', ...)

4.3.9 Partial look ups

When you don't know exactly what you are searching for, but have an idea. Then we use LIKE and ILIKE commands.

WHERE <col.> LIKE 'M%'; [like is case-insensitive]

starts with M

WHERE <col.> ILIKE '%MY%'; [ilike is case-sensitive]

has 'MY' anywhere in between.

Pattern Matching

Wildcard	Meaning
%	Any number of characters.
_ (underscore)	1 character.

placeholder for matching values.

Postgres LIKE does not allow comparisons other than text, so we need to cast (change type) data.

CAST (col AS TEXT) or do col:: TEXT

Use cases

'%2'	ends with 2
'%2%'	has 2 anywhere in value
'_00%'	has zeroes as 2nd & 3rd characters, then n chars.
'%200%'	has 200 anywhere in value.
'2_%_%'	starts with 2 and is atleast 3 chars in length.
'2___3'	5 digit num starts, starts with 2, ends with 3.
'2___3%'	starts with a 5 digit num, that itself starts with 2 and ends with 3
'2%'	starts with 2.

4.4 Date Filters

Dates are hard to deal with due to different time zones.

Sir Sanford Fleming introduced the concept of standard time zone.
It is Greenwich, U.K.

GMT → Greenwich Mean Time (Time Zone)
UTC → Universal Coordinate Time (Time Standard)
] Same time but conceptually different.

To view your timezone, SHOW TIMEZONE;

To change your timezone, SET TIMEZONE 'UTC';

In postgres, you can change the user's time zone by:

CLI : ALTER USER postgres timezone='UTC';

After above command, restart valentina studio.

Manipulating Dates

Formatting Standard → ISO - 8601

ISO → YYYY-MM-DDT HH:MM:SS + offset.

e.g. 2017-08-17T12:47:16 + 02:00

2 hr offset from UTC

Timestamp: date + time with timezone information.

SELECT now()

Ways to store timestamp:

TIMESTAMP WITHOUT TIME ZONE

TIMESTAMP WITH TIME ZONE

Date Operators

Postgres gives us multiple functions to work on dates.

e.g.

```
SELECT NOW() :: DATE ;

SELECT CURRENT_DATE ;

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

format modifiers.

Difference b/w dates

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

This will return difference in number of days.

Casting:

To make a string eligible to be used as a date, ~~mu~~ we DATE keyword and change it to ISO-8601 format.

```
SELECT DATE '1600/01/01';
```

Calculate Age:

```
SELECT AGE ('1543/03/02');
```
← returns error.

```
SELECT AGE (date '1543/03/02');
```
← returns result.

Result format: — years — months — days.

Extracting date elements:

```
SELECT EXTRACT (DAY FROM <date>) AS "Day";

SELECT EXTRACT (MONTH FROM <date>) AS "Month";

SELECT EXTRACT (YEAR FROM <date> AS "Ho-Year";
```

Similarly, we can also extract HOUR, MINUTE and SECOND.

Rounding a date:

```
SELECT DATE_TRUNC ('year', <date>);
```

This will take year and set month & day to lowest value.
Similarly we can round for other date elements, including time.

Interval:

It can store a period of time in years, months, days, hours, minutes, and seconds, and manipulate them.

```
SELECT * FROM orders
WHERE purchasedDate <= now() - INTERVAL '30 days';
```

30 days before current date

More examples -> '1 year 2 months 3 days'
'2 weeks ago'

Extracting from interval:

```
SELECT EXTRACT (YEAR FROM INTERVAL '5 years 20 months');
```

4.5 Distinct Keyword

The distinct keyword removes duplicate values, from the result that we are looking at.

```
SELECT DISTINCT <column-name> FROM <table>;
```

4.6 Sorting Data

e.g.

```
SELECT * FROM customers
ORDER BY "name" ASC;
```

```
SELECT <col> FROM <table>
WHERE <condition>
ORDER BY <col> [ASC | DESC];
```

By default, the order will be ascending, so you can skip ASC keyword.

e.g.

```
SELECT f_name, l_name
FROM employees
ORDER BY f_name, l_name ASC;
```

Here, ASC applies to last name (l_name) only.

It is mandatory that we select the column, which we use in order by.

4.7. Multi-Table Select

In SQL, we can have a combined view of data, which comes from multiple tables simply be giving FROM command, more parameters to operate upon.

e.g.

```
SELECT a.emp_no, b.salary, CONCAT(a.f_name, a.l_name) AS "N"
FROM employees AS a, salaries AS b
WHERE a.emp_no = b.emp_no;
```

matching primary key & foreign key.
This is the link between tables.

A join combines columns from one table with those of another. Take a column from on table that maps to columns of another table.

4.7.1 Inner Join

Finds the intersection b/w datasets and returns a subset.

e.g.1

```
SELECT a.emp_no, CONCAT(a.f_name, a.l_name) AS "Name", b.salary
FROM employees as a
INNER JOIN salaries AS b ON b.emp_no = a.emp_no;
```

e.g.2

```
SELECT a.emp_no, CONCAT(a.f_name, a.l_name) AS "Name", b.salary, c.title,
c.from_date AS "Promoted On" FROM employees AS a
INNER JOIN salaries AS b ON b.emp_no = a.emp_no
INNER JOIN titles AS c ON c.emp_no = a.emp_no
AND (
    b.from_date = c.from_date OR
    c.from_date = (b.from_date + INTERVAL '2 days')
)
ORDER BY a.emp_no ASC;
```

4.7.2 Self Join

This is done when a table has a foreign key referencing its primary key.

e.g.

```
SELECT * FROM table As a, table AS b
WHERE a.supervisor_id = b.id.
```

4.7.3 Outer Join

It returns the values that don't match either from table a or from table b, called LEFT OUTER JOIN and RIGHT OUTER JOIN respectively.

```
SELECT *
FROM table_A AS a
LEFT [OUTER] JOIN table_B AS b ON a.id = b.id;
```

[<something>] ← means it is optional to write it.

Any value that does not match is made to be NULL

e.g. - How many are not managers.

```
SELECT count (emp.emp_no) FROM employees as emp
LEFT OUTER JOIN dept_manager AS dep ON emp.emp_no = dep.emp_no
WHERE dep.emp_no is NULL;
```

4.7.4 USING Keyword.

Using keyword simplifies our joins, by making links easier to write.

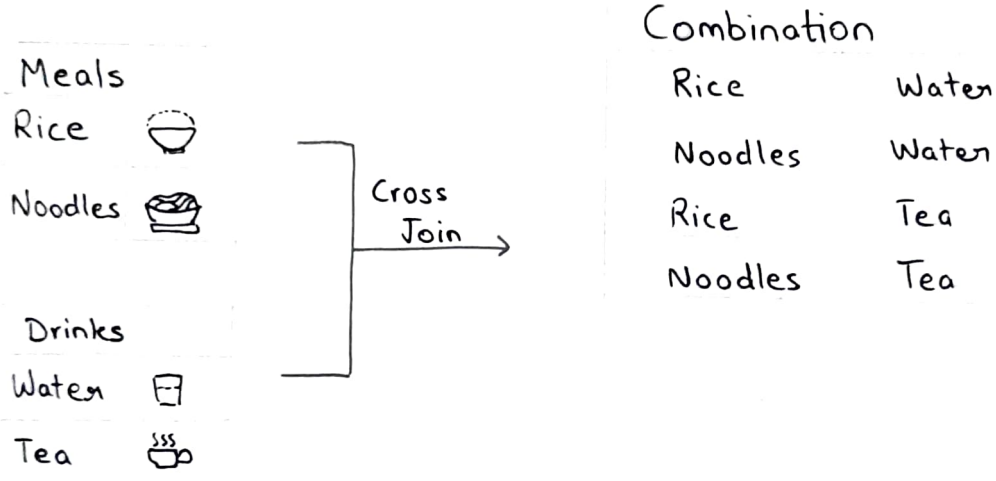
e.g.

```
SELECT e.emp_no, e.f_name, d.dept_no
FROM employees As e
INNER JOIN dept_emp As d USING (emp_no)
```

4.7.5 JOINS Hardly Used

Cross Join : Create a combination of every row with every other row. Cartesian Product is returned.

```
SELECT col_1 FROM table_1 CROSS JOIN table_2;
```



Full Join : Return result from both tables whether they match or not.

```
SELECT * FROM table AS a
FULL JOIN table_2 AS b ON a.pk = b.fk;
```

5. Advanced SQL

5.1 Group By

Split data into groups/chunks so we can apply function against groups rather than entire table.

We use it exclusively with aggregate functions.

Every column not in GROUP BY clause must apply a function.

e.g.

```
SELECT  dept_no , COUNT(emp_no)
FROM    dept_emp
GROUP BY dept_no ;
```

Aggregate functions are applied to reduce all records found for a "group", to a single record. It uses a split-apply-combine strategy.

Order of operations: FROM → WHERE → GROUP BY → SELECT → ORDER

Having Keyword :

HAVING keyword is used to filter groups.

A column in HAVING clause must appear in GROUP BY clause too.

Order of operations: FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER

ORDER BY should come at last of our clauses as it only rearranges the final result.

e.g.

```
SELECT d.dept_name, COUNT(e.emp_no) AS "# of Employees"
FROM employees as e
INNER JOIN dept_emp AS de ON de.emp_no = e.emp_no
INNER JOIN departments AS d ON d.dept_no = de.dept_no
WHERE e.gender = 'F'
GROUP BY d.dept_name
HAVING COUNT(e.emp_no) > 2500
ORDER BY d.dept_name ;
```

Grouping Sets :

We can combine the results of multiple groupings.

Combining results from multiple selects using UNIONs

```
SELECT col1, SUM (col2)
FROM table
GROUP BY col1

[UNION | UNION ALL]

SELECT SUM(col2)
FROM table ;
```

UNION removes duplicates.

UNION ALL does not removes duplicate.

Using aliases is compulsory in case of unions

e.g.

```
SELECT NULL AS "prod_id", sum(ol.quantity)
FROM orderlines AS ol

UNION

SELECT prod_id AS "prod_id", sum (ol.quantity)
FROM orderlines AS ol
GROUP BY prod_id
ORDER BY prod_id ASC;
```

Combining results using GROUPING SETS

The alternative for last code is :↓

```
SELECT prod_id AS "prod_id", sum(ol.quantity)
FROM orderlines AS ol
GROUP BY
    GROUPING SETS(
        (),  <------ Means group by nothing
        (prod_id)
    )
ORDER BY prod_id ASC;
```

GROUPING SETS is a subclause of group by that allows us to define multiple grouping.

Combining results using ROLLUP

Rollup takes a group and then it does grouping based on the Power Set of the provided grouping set, including 'nothing'.

```
SELECT ________
FROM ________
GROUP BY
    ROLLUP(
        <superset group>
        <- - - - - - ->,
        <- - - - - - - ->
    )
ORDER BY
    - - - -
    - - - -
```

Grouping will be done for every possible combination of provided groups. Thus, a power set.

Here, supersset means that we write all the unique grouping conditions, then rollup will do the combinations itself.

5.2 Window Functions

Window functions are used to apply a function against a set of rows related to the current row.

Window functions create a new column based on functions performed on a subset or "window" of data.

Window functions are used with SELECT command.

```
SELECT
    <col>,
    window_function(<col1>, <col2>, ...) OVER(
        [PARTITION BY <partition-column>]
        [ORDER BY <sort-column> [ASC | DESC]]
        [frame clause]
    )
FROM <table>
```

- Window function: aggregate or ranking function.
- They run against each row again, so they take some time.
- PARTITION BY divides rows into groups to apply the function.
- ORDER BY operates differently with window functions. It has a special property - framing, it changes frame of window functions.
- Frame clause → The window frame is a set of rows related to current row where the window function is used for calculation.
- If ORDER BY is specified, then the frame is

  RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

  or use

  RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

Key	Meaning
ROW or RANGE	whether you want to use n rows or a range (devided logically) as a frame.
PRECEDING	n rows before current row
FOLLOWING	n rows after current row.
UNBOUNDED PRECEDING	return all rows before current row
CURRENT ROW	current row
UNBOUNDED FOLLOWING	return all rows after current row

List of Window functions :

- Ranking Functions
  - row_number() , rank() , dense_rank()
- Distribution Functions
  - percent_rank() , cume_dist()
- Analytic Functions
  - lead() lag() ntile() first_value() last_value()
  - nth_value()
- Aggregate Functions
  - avg() count() max() min() sum()

NULLIF function :

SELECT NULLIF(val1, val2)

- If val1 is equal to val2, then NULL is returned.
- Can be used to fill empty spots with a NULL, or avoid divide by zero.

e.g. Get last pay date of an employee and tell what was the salary.

```
SELECT
    DISTINCT s.emp_no,
    LAST_VALUE (s.from_date) OVER (
        PARTITION BY s.emp_no
        ORDER BY s.from_date
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
        FOLLOWING
    ) AS "Last Pay Date",

    LAST_VALUE (s.salary) OVER (
        PARTITION BY s.emp_no
        ORDER BY s.from_date
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
        FOLLOWING
    ) AS "Current Salary"
FROM salaries AS s
JOIN dept_emp AS de USING(emp_no)
ORDER BY s.emp_no;
```

<u>Limiting the output we see</u> :

```
SELECT *, MAX(salary) OVER()
FROM salaries
LIMIT 100;
```

for all the rows in query result.

To view only 100 rows of result.

5.3 Conditional Statements

Statements used when we want to select something based on a condition. It can be used in ~~pt~~ multiple places in a query.

```
CASE
    WHEN    <condition>
    THEN    <expression>
    ELSE    <expression>
END
```

e.g. 1

```
SELECT  SUM(
            CASE
                WHEN o.netamount < 100 THEN -100
                ELSE o.netamount
            END
    ) AS "Returns", SUM(o.netamount) AS "Normal total"
    FROM orders as o;
```

e.g. 2

```
SELECT
    o.orderid,
    o.customerid,
    CASE
        WHEN o.customerid = 1 THEN 'First Customer'
        ELSE 'Not first Customer'
    END,
    o.netamount
FROM orders AS O
ORDER BY o.customerid;
```

5.4. Views

When we want to query the results of a query, we store it in a variable, called **view**.

Types of Views :

- Materialized : stores data physically and periodically updates it when tables change.
- Non-Materialized : query get re-run each time the view is called on. Stores definition, not data.

Creating a View

```
CREATE VIEW <name_of_view> AS
<write-your-query>
```

To use this view, e.g.,

```
SELECT <col>
JOIN <name-of-view> USING(fk)
WHERE <condition> ;
```

Updating a View

```
CREATE OR REPLACE <view_name> AS
<write-your-query>
```

If the view does not exist, then create one, else update it with the specified query.

Renaming a View

```
ALTER VIEW <view_name> RENAME TO <new_name> ;
```

Deleting a View

```
DROP VIEW [IF EXISTS] <view_name>
```

Advantages of View

- Join syntaxes become easier to read.
- Easier to reason about.
- Separates concerns.

Disadvantages of View

- Take extra space
- Can be slower.
- Becomes inactive if table is deleted.

5.5. Indexes

A table of contents that help you find where a piece of data is. It is basically table of pointers. It is used to improve quering performance, however, it slows down intersection and updates.

Types of Indexes

- Single-Column → used against most frequently used column in a query.
- Multi-Column → used against most frequently used columns in a query.
- Partial Index → index over a subset of data.
- Unique Index → used for speed & integrity.
- Implicit Index → automatically created by DB for primary key and unique key.

When to use indexes ?

- Index foreign keys.
- Index primary keys and unique columns.
- Index on columns that end up in the order by/where clause often.

When not to use indexes ?

- Don't use on small tables.
- Don't use on tables updated frequently.
- Columns that contain null.
- Columns that have large values.
- Just to use index.

Creating Index

multi-column

```
CREATE INDEX <index_name> ON <table> (col1, col2, col3, ...)
```

single-column

```
CREATE INDEX <name> ON <table> (column)
```

unique

```
CREATE UNIQUE INDEX <name> ON <table> (column)
```

partial

```
CREATE INDEX <name> ON <table> (column) WHERE <condition>
```

e.g.

```
CREATE INDEX idx_countrycode ON city (countrycode)
<query...>
SELECT "name", district, countrycode FROM city
WHERE countrycode IN ('TUN', 'BE', 'NL');
```

Delete Index

```
DROP INDEX <name_of_index>
```

Analyzing Queries

EXPLAIN ANALYZE
<query>

Do this on a query before and after creating index and notice time change in execution time.

gives the details of query execution instead of the result.

QUERY PLAN
Seq Scan on <table> ...
Filter: ....
Rows Removed by Filter: ...
Planning Time: ...
Execution Time: ...

Index Algorithms

Each index type uses a different algorithm. Postgres provides different algos: (Most used are listed below)

- **B-Tree (Default)** Best used for comparisons with <, <=, >, >=, =, BETWEEN, IN, IS NULL, IS NOT NULL.
- **Hash** handle equality checks only.
- **GIN (Generalized Inverted Index)** best used when multiple values are stored in a single field.
- **GIST (Generalized Search Tree)** useful in indexing geomatric data and full text search.

Using specific Algorithm for Index

(hash, gin, gist)

CREATE INDEX <name> ON <table> USING <method> (column)

5.6 Subquery

- A query inside a query, i.e., a nested query.
- Also called <u>Inner Query</u> or <u>Inner Select</u>.
- Used in SELECT, FROM, HAVING and WHERE clause, but most often used in WHERE clause.
- Used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
- Enclosed within parenthesis.
- Outer query is called <u>main query</u>, inner query is called <u>sub query</u>
- Subqueries must be on right side of comparison.
- Cannot manipulate their result, thus, cannot use ORDER BY in subquery.
- Try to use JOIN instead of subqueries whenever possible.
- Use single row operators with single row subqueries.

<u>Types of Subqueries</u>

- Single Row — Return 1 record (row)
- Multiple Rows — Return multiple records (rows)
- Multiple Columns — Return multiple columns
- Correlated — Reference to column in outer query.
- Nested — Subquery within a subquery.

e.g. Get latest salary.

```
SELECT
    emp.no,
    salary AS "Most Recent Salary",
    from_date
FROM salaries AS s
WHERE from_date = (
        SELECT MAX(from_date)
        FROM salaries AS sp
        WHERE sp.emp_no = s.emp_no
)
ORDER BY emp_no ASC;
```

e.g., Get ~~ste~~ salary on most recent pay date.

```
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;
```

Subquery Operators

There are various operators that you can apply on subqueries in the WHERE clause.

- EXISTS — check if subquery returns any row
- IN — check if the value is equal to any of the row values, remember, null yields null.
- NOT IN — reverse of IN mentioned above, i.e., checks for non-existence.
- ANY/SOME — check if any comparison matches among the returned rows, then returns TRUE.
- ALL — check each row against operation and all should match.
- Single value comparison — subquery returns 1 row, match it against where condition, using (=).

Examples

1. Exists :
```
SELECT prod_id FROM products
WHERE EXISTS ( SELECT category from categories
               WHERE categoryname IN ('Comedy', 'Family')
)
```

2. IN :
```
SELECT prod_id FROM products
WHERE category IN (
        SELECT category FROM categories
        WHERE categoryname IN ('Classics', 'Family')
)
```

3. Not In :
```
SELECT prod_id FROM products WHERE category NOT IN (
        SELECT category FROM categories
        WHERE categoryname IN ('Classics', 'Family')
)
```

4. Any / Some :

```
SELECT prod_id FROM products
WHERE category = ANY(
    SELECT category FROM categories
    WHERE categoryname IN ('Comedy', 'Family', 'Classics')
)
```

5. All :

```
SELECT prod_id, title, sales FROM sales
JOIN inventory AS i USING(prod_id)
WHERE i.sales > ALL(
    SELECT AVG(sales) FROM inventory
    JOIN products AS pl USING(prod_id)
    GROUP BY pl.category
)
```

6. Single Row Comparison :

```
SELECT prod_id FROM products
WHERE category = (
    SELECT category FROM categories
    WHERE categoryname IN ('Comedy')
)
```

6. Database Management

This module is all about learning DDL and DML. It is very important to learn how databases are created and what happens behind the scenes, because databases are hard to manage. Small mistakes lead to huge data losses. Will also see DCL.

DDL: Commands to create, modify, delete and manipulate different structures.

DML: Commands to manipulate the data.

Options available while creating DBs:

- Regular - built off of a blueprint.
- Template - create a blueprint.

When we create database in Postgres, 3 DBs are created:

Postgres	Template 0	Template 1
(default DB created when you setup postgres)	(used to create template 1. Backup for template 1)	(templates used to create new DBs)
	Never Change This	

The first time you initialize Postgres, you can connect to Postgres DB to setup you first DB.

CLI: psql -U <user> <database>

tells that we are going to give a user

psql -U postgres } connect to DB called postgres. Assumes this is DB.

\conninfo -> shows our current connection.

exit -> to close the connection.

Creating a new DB :

CLI :- psql -U postgres postgres ↵] important to connect to postgres.

CREATE DATABASE <db_name>; ↵

Syntax → CREATE DATABASE <name>
[[WITH] [OWNER [=] <user_name>]
[TEMPLATE [=] template]
[ENCODING [=] encoding]
[LC_COLLATE [=] lc_collate]
[LC_CTYPE [=] lc_ctype]
[TABLESPACE [=] tablespace]
[CONNECTION LIMIT [=] connlimit]]

Settings	Default
TEMPLATE	template 1
ENCODING	UTF8
CONNECTION LIMIT	100
OWNER	Current User

e.g CLI - CREATE DATABASE ztm;

To create DB with a specific name and from a template:

CLI -

CREATE DATABASE <db_name> WITH TEMPLATE <template>;

To create our own template:

```
CREATE DATABASE <temp_name> ;
CREATE TABLE <table_name> ( ... );
CREATE DATABASE <db_name> WITH TEMPLATE <temp_name>;
```

Deleting a DB :

CLI :- DROP DATABASE <db_name>;

6.1 Data Organization

A DB has multiple structures in it, like views, tables, indexes, etc. It is important to separate concerns. For that, Postgres offers the concept of Schemas. These are like boxes in which we organize tables, views, indexes, etc.

By default, each DB has a "public" schema which is available to all other schemas.

Reasons to use Schema :

- Allow many users to use DB without interfering with each other.
- To organize DB objects into logical groups to make them easy to manage.
- Third party applications can be put in different schemas so they don't collide with names of other objects.

NOTE : To rename a DB,

CLI :- ALTER DATABASE <old_name> MODIFY NAME = <new_name>;

Creating a Schema :

CLI:-

CREATE SCHEMA <name> [AUTHORIZATION <user>] [schema_element[...]]

can use IF NOT EXISTS also.] Postgres feature.

schema_element → A SQL statement defining an object to be created within the schema.

Altering Schema :

ALTER SCHEMA <name> RENAME TO <new_name>

ALTER SCHEMA <name> OWNER TO { new_owner }

] Postgres feature. Not in SQL standard.

Deleting a Schema :

Postgres feature.

DROP SCHEMA [IF EXISTS] <name> [CASCADE | RESTRICT]

CASCADE → Automatically drop objects (tables, functions, etc) that are contained in schema.

RESTRICT → Refuse to drop if schema contains any objects. This is the default.

View Schemas :

CLI: \dn ← This command shows all databases.

6.2 Roles, Users and Privileges

Roles in sql determine what is allowed in DB and by whom. A role can be a user or a group. They also have ability to grant memberships to other roles.

Attributes are the properties of a user. They define privileges.

Privilege is the right to do something in DB. They can be given independent of attributes.

Common Attributes :↓

Attributes	Purpose.
login privilege	"database user"
Superuser status	bypass all permission checks.
Database Creation	a role must be explicitly given permission to create dbs.
Role Creation	ability to create / drop other roles (explicity)
Password	significant only if you give login privilege.
Initiating replication	Explicit permission to initiate streaming replication.

CLI: \du ← Command to view all roles.

Creating roles :↓

CREATE ROLE <name> LOGIN

CREATE ROLE <name> SUPERUSER

CREATE ROLE <name> CREATEROLE

CREATE ROLE <name> PASSWORD '<string>'

CREATE ROLE <name> REPLICATION LOGIN

e.g.
CREATE ROLE readonly WITH LOGIN ENCRYPTED PASSWORD 'reader'

Always encrypt when storing a role that can login. By default only the creator of DB or superuser has access to its objects.

Attributes Examples :↓

- Create DB
- No createDB
- Superuser
- No superuser
- Createrole
- No createrole
- Login
- Nologin

Creating Users & Configuring Login :↓

CREATE USER <name> is an alias for CREATE ROLE. Only difference is when we use USER, LOGIN is assumed by default, whereas in ROLE, NOLOGIN is assumed/taken.

CREATE USER is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

CREATE ROLE <r_name> WITH LOGIN ENCRYPTED PASSWORD 'pass';

↓ same as.

CREATE USER <u-name> WITH ENCRYPTED PASSWORD 'pass';

When we are not connected to Postgres, we can create user by:

```
createuser -- interactive ] will ask some questions
                           about attributes.
```

Edit a role using: ALTER command.

e.g. ALTER ROLE <name> WITH ENCRYPTED PASSWORD 'string';

Connect to DB using a user/role you created:

```
psql -U <u.name> <db>

        OR

psql -U <u-name> <db> -W ] forced authentication.
```

The former does not ask for password, the latter continues even with wrong password. This is due to configuration of postgres, which trusts all connections be default.

pg-hba.conf : file that specifies how you are allowed to connect

postgresql.conf : file that contains general configuration of sql.

```
CLI : show hba_file;      ] show paths of above mentioned files.
      show config_file;   ] You need to be superuser for this.
```

pg-hba.conf :

TYPE	DATABASE	USER	ADDRESS	METHOD
local	all	all		trust] default.

Change default to:

local all all <address>. scram-sha-256

strongest encryption method generally used when external machines are connected to DB.

Change method to scram-sha-256 for all rows.

Restart DB.

CLI: psql -U <u_name> <db_name> ; ↵

Now it will ask for password. Although it will fail.

In postgresql.conf :

set 'password_encryption' = scram-sha-256

field previously set to md5. Don't add quotations, used here to put emphasis.

After this you need to alter user and provide encrypted password again after DB restart; put method back to trust (hba file)

ALTER USER <u_name> WITH PASSWORD 'string';

Turn method (hba-file) back to scram-sha-256.

Now type:

psql -U <u_name> <db_name> ; ↵

After setting up encryption method, no need to use keyword ENCRYPTED.

Usually, in local machine, in hba-file, first row is set to

local all postgres <nothing: blank-spaces> trust

and method for others is set to scram-sha-256.

Privileges :↓

Attributes give you limited privileges, specially when you are not a superuser.

By default objects are only available to the one who creates them.

GRANT ALL PRIVILEGES ON <table> TO <user>;

GRANT ALL ON ALL TABLES [IN SCHEMA<schema>] TO <user>;

GRANT [SELECT, UPDATE, ...] ON <table> [IN SCHEMA <schema>] TO <user>;

Be very wise about the privileges you grant.

Granting Privileges & Role Management :↓

Createuser --interactive ← give it attrs & name

psql -U <name> <db> ←

\dt ← will show all tables in this DB.

SELECT * FROM titles; ← This will fail.

ERROR: permission denied for table titles;

In another command line, connect as superuser. Then type this:

GRANT SELECT ON <table> TO <u.name>; [must be connected to DB which has this table]

↳ titles

Go back to previous command line, run:

SELECT * FROM titles; ← This will now work.

Can remove privileges using REVOKE command:

REVOKE SELECT ON <table> FROM <u_name>; ← Doing as user who granted it.

To grant & revoke roles:

GRANT <ROLE> TO <user>;

REVOKE <ROLE> FROM <user>;

→ For this to work, first create a role and give this role some privileges.

Be very careful about granting privileges. Start with least privileges.

Best Practices for Role Management: ↓

- Always go with the "Privilege of Least Privilege".
- Don't use admin/superuser by default. Give users roles and privileges to maintain controle.

6.3 Data Types

Data stored in DBs can be of different types. Data type tells us what kind of data is being stored and how to handle it.

We put constraints on a field of table to only only specific data type to be stored in it.

Types

- Numeric Types
- Arrays
- Character Types
- Date / Time
- Booleans
- UUID Types

} Core types

- Many More (Bits, XML, etc.)

Character Data Type :

Postgres provides three character data types.

CHAR(N)	VARCHAR(N)	TEXT
Fixed length with space padding. Adds spaces if less characters given.	Similar to CHAR, except, it does not add spaces if length provided is less than limit.	Unlimited length string.

Space Padding → count spaces as characters.

Numeric Data Type :↓

- Integers
  - Smallint (Range: ± 32,768)
  - int (Range: ± 2,147,483,648)
  - Big int (Range: ±9223372036854775808)

- Floating
  - Float 4
  - Float 8
  - Decimal

- Rounds up if greater points are given.
- Only decimal / fraction numbers.

Arrays Data Type :↓

Group of data elements of same type.

Syntax: <type> [ ],

e.g. char(2) [] ← each data element can be of char type only with length limit of 2.

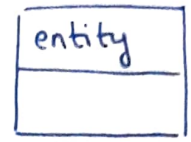
6.4. Data Models with Naming Conventions

Before creating a DB, we should first create a DB model.

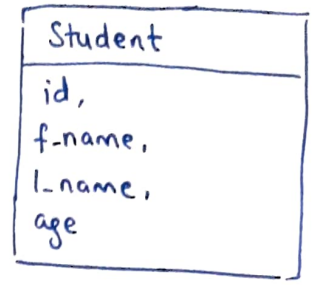
Model :- Design used to visualize what we are going to build.

Crow's Feet Notation :

- **Entity:** It is a representation of a class of objects. Represented by a rectangle with its name on top. The name is singular.

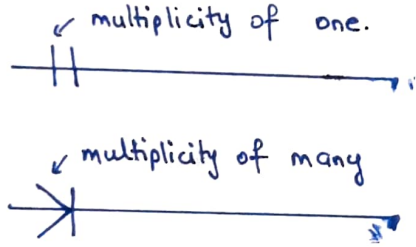


- **Attributes:** Property that describes a particular entity



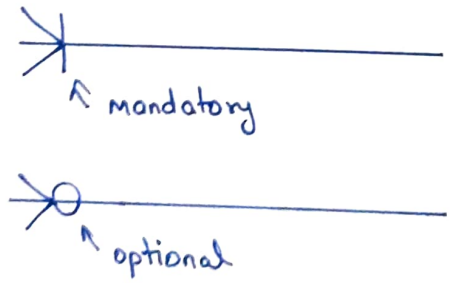
- **Relationships:** Associations between two entities. Presented as a straight line. They have two indicators.

1.



multiplicity refers to max number of times that an instance of entity can be associated with that of other entities.

2.



min number of times one instance can be related to another. It can be zero or one.

The possible edges are

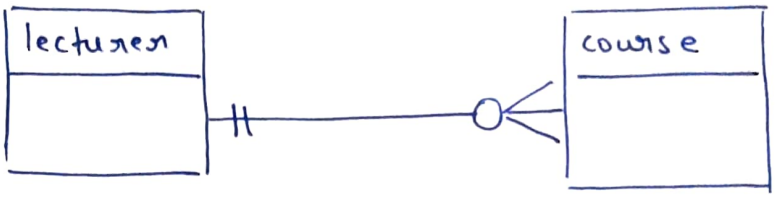
- zero or many
- one and only one
- zero or one
- one or many

Examples

- One to One

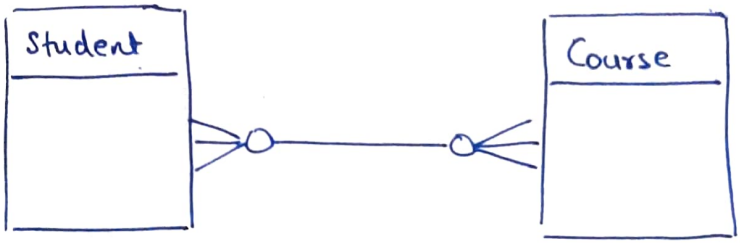
1 student can have only 1 seat. 1 seat can have only 1 student on it.

- One to Many



1 lecturer can teach multiple courses (optional). Multiple courses can be taught by 1 lecturer.

- Many to Many



1 student can be enrolled in many courses, or none.

1 course can have multiple students or none.

The diagrams you see above are called ERD (Entity Relationship Diagram)

Naming Conventions :↓

- Table names must be singular.
- Column names are lowercase and underscore separated words.
- Columns with mixed case are acceptable.
- Columns with upper case are unacceptable.

NOTE: Be consist about you Database model and write down your own rules.

6.5. Creating Tables

```
CREATE TABLE <name> (
        <col_name> <TYPE> [constraint],
        table_constraint [constraints]
) [INHERITS <existing_table>] ;
```

For a column that will contain unique ids, we can have the following constraint

```
DEFAULT uuid_generate_v4().
```

CLI : create extension if not exists "uuid-ossp";

Postgres allows you to install extensions.

CLI : \dc <table_name> ← shows structure of table.

Temporary Tables :

CREATE TEMPORARY TABLE <name> (<cols>);

They are type of table that exist in a special schema, so you cannot define a schema name when declaring temporary table.

These will be dropped at the end of your session. Only visible to creator.

Column Constraints :

Constraints are a tool to apply validation methods against data that will be inserted.

Column constraint is defined as part of column definition.

- NOT NULL
- CHECK
- PRIMARY KEY
- REFERENCES
- UNIQUE

Table Constraints :

Definition is not tied to one particular column. It can encompass more than columns. All column constraints can be used.

- UNIQUE (column_list)
- PRIMARY KEY (column_list)
- CHECK (condition)
- REFERENCES (column_list)

e.g.,

```
CREATE TABLE teacher (
    teacher_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    date_of_birth DATE NOT NULL,
    email TEXT
)
```

Domain :

A data type with optional constraints. The domain name must be unique among the types and domains existing in its schema.

name of domain.

```
CREATE DOMAIN <RATING> smallint CHECK (VALUE >0 AND VALUE <=5);
```

```
CREATE TYPE feedback AS (
    student_id UUID,
    rating <RATING>,
    feedback TEXT
)
```

← creating our own data type that we can store in columns of tables.

we can use it here.

```
CREATE TABLE course (
    course_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    "name" TEXT NOT NULL,
    description TEXT,
    teacher_id UUID REFERENCES teacher (teacher_id),
    "feedback" feedback[]
)
```

We can also use two columns as primary key. e.g.,

```
CREATE TABLE enrollment (
    course_id UUID REFERENCES course (course_id),
    student_id UUID REFERENCES student (student_id),
    date DATE NOT NULL,
    CONSTRAINT pk_enrollment PRIMARY KEY (course_id, student_id)
```

Using UUID :↓

Universally Unique Identifier. It allows us to generate unique identifiers for primary keys.

Extensions :- Pieces of softwares that allow you to expand what postgres can do or expand how certain programmes run.

Pros	Cons
• Unique everywhere.	• Larger values to store.
• It's easier to shard.	• Can have performance impact.
• Easier to merge/replicate	• More difficult to debug
• Expose less info about your system.	

Altering Tables :↓

```
ALTER TABLE [IF EXISTS] [ONLY] <name> [*]
    ADD COLUMN <col> <type> <constraint> ;

ALTER TABLE [IF EXISTS] [ONLY] <name> [*]
    ALTER COLUMN <name> TYPE <new_type> [USING <expression>];

ALTER TABLE [IF EXISTS] [ONLY] <name> [*]
    RENAME COLUMN <old_name> TO <new_name> ;

ALTER TABLE [IF EXISTS] [ONLY] <name> [*]
    DROP COLUMN <col> [RESTRICT | CASCADE] ;
```

Deleting Table :↓

```
DROP TABLE <table_name> ;
```

6.6. Inserting into Tables

```
INSERT INTO <table> (col1, col2, col3)
        VALUES (val1, val2, val3)
```

e.g.

```
INSERT INTO subject (subject, description)
        VALUES ('SQL', 'A DB management language')
```

6.7. Update & Delete Commands

Update :

```
UPDATE <table_name>
SET col1= val1 , col2 = val2 , ...
WHERE <condition>;
```

change existing values.

Delete :

```
DELETE FROM <table_name>
WHERE <condition>
```

delete existing row(s).

without 'where' condition, all rows will be deleted.

6.8. Backups

Backup is the copy of your data that you keep incase something undesired happens to original data.

What can go wrong ?

1. Hardware failures.
2. Viruses.
3. Power outages.
4. Hackers.
5. Human Error.

How do I make a plan ?

- What needs to be backed up
  - Full Backup (back up all data)
  - Incremental (everything after last back up)
  - Differential (everything since last full back up)
  - Transaction log (Backup transactions track (changes in db))

- What's the appropriate way to back up ?
- Decide how frequently are you going to backup ?
- Decide where to store ?
- Have a retention policy for the backup.

Backup in PostgreSQL :↓

SQL Dump : text file that either contains schemas or data.

pgBackRest : tool that works with PostgreSQL for handling backups.

Create :↓

Right Click → Create Dump → <set options> → Next → Next .
(on DB name)

Restore Backup :↓

Right Click → Load Dump → Connection → options
(in DBs section)

Backup restorations are meant to be run in command line:

CLI: \i <path>

6.8. Transactions

A transaction is a unit of instructions.

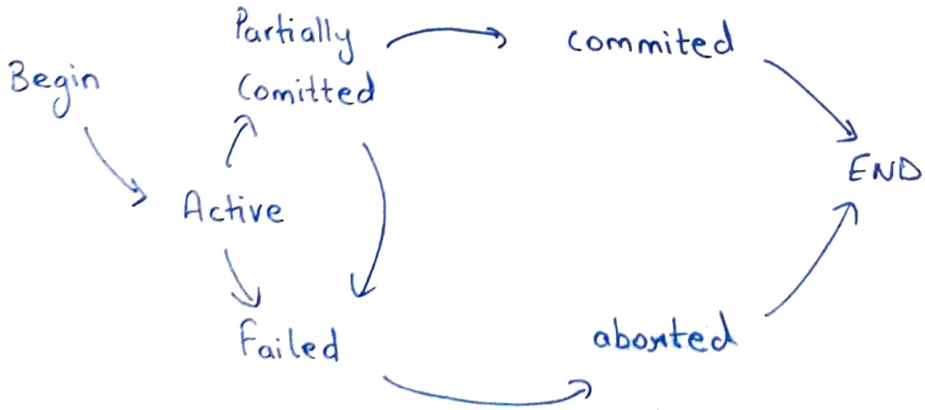
The DBMS has a mechanism in place to manage transactions.

For updates, deletes, etc., certain resources are locked to avoid collisions.

CLI: BEGIN; ← to start transaction.

CLI: END; ← to commit current transactions.
A postgres equivalent to COMMIT

Lifecycle:



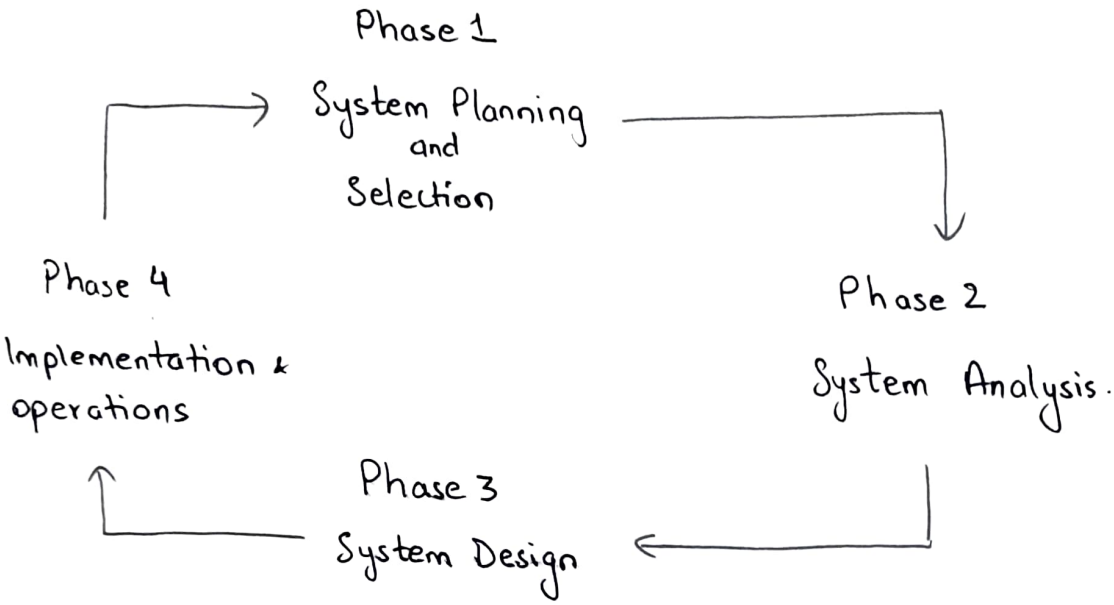
ACID :

To maintain the integrity of a DB, all transactions must obey ACID properties.

- **Atomicity**: Either execute entirely or not at all.
- **Consistency**: Each transaction should leave the DB in a consistent state (Commit or Rollback).
- **Isolation**: Transactions should be executed in isolation from other transactions.
- **Durability**: After completion of a transaction, the changes in DB should persist.

8. System Design and SDLC

Software Design Life Cycle :



Goal: To design robust system.

Phase 1: Getting info on what needs to be done

Phase 2: Analyzing if it can be done on time in budget

Phase 3: Design system architecture for all related components - Databases, APIs, etc.

Phase 4: Building software.

There can be more phases, such as testing.

Focus of this Module → Phase 3, for databases.

System design is all about creating a structure that can be understood and communicated.

Techniques to design databases :

- **Top-down** :
  - optimal choice when creating new databases.
  - Starts from zero to full DB.
- **Bottom-up** :
  - optimal choice when migrating an existing DB.
  - There is a DB or data already, on which we build a new DB.

Think abstract & communicate clearly ← skill to be good DB engineer.

Phase 3 → How to design
- Top-down → ERD
- Bottom-Up → Normalization.

ERD (Entity Relationship Diagram/Modelling) : a diagram that functions as a way to structure high level requirements.

Relational Model Extended :

- **Relation Schema**: design/structure of a table.
- **Instance** : set of data/rows.
- **Relation key**: key that uniquely identifies a row.
- **Super key**: combination of attributes which is unique.
- **Candidate key**: min amount of attributes required to uniquely identify a row.

- <u>Primary key</u>: the relation key column we decide to use.
- <u>Foreign key</u>: referencing primary key in another table.
- <u>Compound key</u>: multiple columns and any of foreign key.
- <u>Composite key</u>: multiple columns but not foreign key.
- <u>Surrogate key</u>: having nothing to do with individual data, but can be used to identify a row.
- <u>Alter key</u>: set of compound, composite & surrogate key.

<u>A note on Many to Many relations</u>:

Always avoid a many to many relation for following reasons:

- Insertion, update & delete overhead.
- Duplicate data.

Always try to solve Many to Many to 1 to Many.

Tables created to solve Many to Many are called intermediate entities.

Anomalies: problems that arise when your DB is not structured correctly. 3 types of anomalies:

- Insert anomaly.
- Update anomaly.
- Delete anomaly.

Bo

Normalization: Process to isolate entities in bottom-up approach.

Edgar Codd proposed the idea/theory of normalization.

To do normalization, we must understand 2 things:

- Functional Dependencies
- Normal Forms.

Functional Dependency:

A function showing the relationship b/w attributes, of entities. It exists when a relationship b/w two attributes allow you to uniquely determine the corresponding attribute's value.

A is dependent on B, is shown by

$$B \longrightarrow A$$

Notation: Determinant $\longrightarrow$ Dependent.

Can also have

$$col1, col2 \longrightarrow col3$$

Normalization & Normal Forms :

Normalization happens when we run attributes through normal forms.

0 NF → 1 NF → 2 NF → 3 NF → BCNF → 4NF → 5 NF → 6 NF

Each normal form aims to further separate relationships into smaller instances so as to create less redundancy and anomalies.

NF 0 to NF BC (3.5 NF) are the most common NFs to run through.

NF 4 to 5 to further reduce anomalies. Hardly needed.

NF 6 is not yet standardized.

0 NF :

- Data is unnormalized.
- Repeating groups of fields.
- Positional dependence of data.
- Non - atomic data.

0NF to 1NF :

- Eliminate repeating columns.
- Atomic data
- Determine primary key.

NOTE : A repeating group means that a table contains two or more columns that are closely related.

1 NF to 2NF :

- It is in 1 NF.
- All non-key attributes are fully functionally dependent on primary key. Determinant should not be combination of keys.

3NF (2NF to 3NF):

- It is in 2NF.
- No transitive dependencies:

$$\left.\begin{array}{l} B \rightarrow A \\ C \rightarrow B \\ A \rightarrow C \end{array}\right] \text{transitive meaning.}$$

Boyce Codd NF :

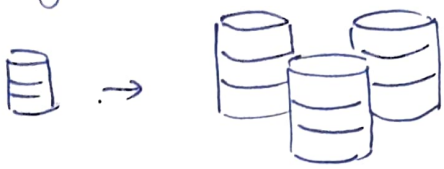
- It is in 3NF
- For any dependency $A \rightarrow B$, 'A' should be super key.
- BCNF does not allow attributes to be part of candidate key, which are not primary key. But 3NF allows this.
- The relationship is not in BCNF if:
  - The PK is a composite key.
  - More than 1 candidate key.
  - Some attributes have keys in common.

9. Database Landscape

Scalability: → Capability of DB to handle growing data.

- **Vertical :** Add more physical hardware - disk/space, etc.

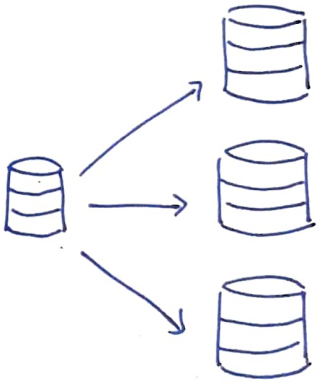
- **Horizontal :** Adding more machines with more DBs.



Sharding: To hande growing amount of data, it is divided into categories. Each category is a DB.

Replication:

- Synchronous Update to DB, then all DB, then reply.
- Asynchronous Update to DB, reply, update all DBs in background.



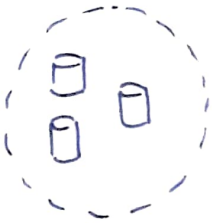
DBs in different locations.

Backups: Copy of data in a secured location, in case something happens to original data. Not done as often.

Distributed & Centralized DBs: Where is the DB kept and who controls it.

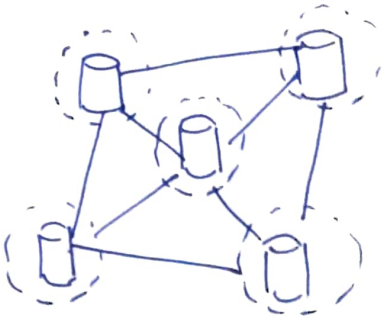
Centralized:

- Kept in 1 locations.
- Controlled by 1 organization.
- No outdated data



Decentralized:

- Kept in different locations
- Can be controlled by different organizations or just 1.
- More complex.
- Good for backups



Database Security: Common security practices are:

- Users see what they are authorized to see.
- Keep unauthorized users to access DB (only certain DB)
- Prevent things like data corruption (safe & cool location)
- DB is available when needed. No system crashes.

Injections: Injecting code into another piece of code.

' or 1 = 1 --

To close the quotes of value

Return True.

Comment out rest code.

Can be put in password field. Returns all user info, after logging you in.

Encryptions : pgcrypto → tool to encrypt data in Postgres DB.

Relational DB vs NoSQL DB :

Relational	NoSQL
↑ · Stored in Tables	↓ · Stored in documents
↑ · Data Integrity - no duplicates	↓ · Duplicate data.
↑ · Has a standard - SQL	↓ · No standard.
↓ · Schema - everything needs to be designed from starting.	· flexible structure. Easy to add new data.
↓ · Harder to scale horizontally	↑ · Easy to scale horizontally
↓ · In different tables, need to know relations to query data	↑ · All related data is in one location.
e.g. PostgreSQL, MySQL	e.g. MongoDB, CouchDB,

Future of Relational DBs :

NewSQL, is something new ~~immerging~~ emerging in industry. It is built on top of Relational DB to make it scale horizontally.

e.g. Google Spanner

Coackroach DB

Elastic Search: A database specifically build for fast full text searches. It is less reliable than Relational DB.

As data and company grow larger, various DBMS are used for various purposes.

Amazon S3: Object storage built to store and retrieve any amount of data from anywhere.

Top DBMS to use:

- PostgreSQL
- MySQL
- SQLite
- MongoDB
- Redis
- Amazon Document DB
- Firebase
- Elastic Search
- Amazon S3.

10. Big Data + Analytics

Spreadsheets → Relational DB → "Big Data"

Evolution in collection and analyzes of data.

NoSQL MongoDB

Machine Learning

Data Engineering Steps : Steps in a full Machine Learning Project.

- **Data Collection**: Collect data from various resources. (Hardest Part)
- **Data Modelling**: what problems are we trying to solve? → what data do we have → what defines success. ↓ what features should we model.
- **Deployment**: Implement what you learned.

Introduction to Data Engineering :

A data engineer takes data coming from various resources and form a well organized DB out of it that makes sense to reader.

Why businesses care about data ?

Part of product is - Are we doing ok ?

Can we do better ?

Types of Data

1. Structured Data : Usually comes from Relational DB
2. Semi-structured data : XML, CSV, json.
3. Unstructured Data : PDF, emails or any other document.
4. Binary Data.

What does Data Engineer do ?

Data Minning	Big Data	Data Pipeline
extract knowledge from data	data so big that it runs on multiple machines	Helps us to extract knowledge.

Builds data that Data Scientists / Data Analyzers study. They don't care where data comes from or if it is valid. That is job of Data Engineer.

Use tools to create Data Warehouse from Data Lake → Query the warehouse → Derive value/ predictions (Job of Data Analyst)

e.g., Hadoop

(Job of Data Engineer)

e.g., Google Big Query

(Job of Data Scientist)

Overview:

1. Build ETL (Extract Transform Load) pipeline.
2. Building analysis tools.
3. Maintain Data Warehouse & Data Lakes.

Types of Databases

SQL / Relational (PostgreSQL, MySQL)

NoSQL (Mongo DB, Redis)

New SQL (Cockroach DB)

Hadoop, HDFS, MapReduce

1. Hadoop: Open source distributed processing system. Allowed to manage data lake. Popular due to these two reasons:

   1. HDFS (Hadoop Distributed file System) → Allows to store files on multiple computers.
   2. MapReduce → Allows to perform jobs on data in data lake. Run batch jobs.

2. Hive: Allowed to use SQL on data in HDFS. Data warehouse software built on top of Apache Hadoop for providing data query and analyst.

Apache Spark & Apache Flink :

Before spark, people used MapReduce for processing data. Spark improved it through in-memory processing. It is a go to for batch processing. You can store in Hadoop and process using Spark.

Later, Real-time processing came. Apache Flink allows real-time stream processing.

Main tools for stream processing → Kafka. Allows us to read and write streams of data. It allows us to recieve message and pass it to different location. We call this a Message Broker.

We often use Kafka to collect data from various resources, then pass it to Hadoop cluster which can be processed by spark.

11. Redis

Cache: Temporary data storage for quick reference is called cache.

Types of data storages in Computers:

CPU	RAM	Hard-Disk
CPU has on chip memory called registers.	Not as close a registers, but closer to CPU.	Physical disks used for permanent storage.
Closest & smallest, but also fastest.	Large enough to hold significant data.	Data exists even without power.
	Data gone when power gone.	

cheaper and slower.

Redis is used for caching DB data.

Redis is in-memory DB used for short lived data in DBs. Thus, used in sessions and cache. In-memory data means small pieces of data, we don't care if it is lost, although, redis takes snapshots regularly and store it in disk time to time.

Redis Commands: ↓ (All in Command Line)

SET <key> value	(store some value)
GET <key>	(read some value)
EXISTS <key>	(to see if it exists)
DEL <key>	(to delete a key)

e.g. SET session "Jenny"
EXPIRE session 10] to delete a key after 10 seconds.

SET val 1000

INCRBY <key> (to increment a value)

DECRBY <key> (to decrement a value)

Redis Data Types :

Multiple SET : MSET <key1> <val1> <key2> <val2> ...

Multiple GET : MGET <key1> <key2> ...

5 Major Redis-data types are :

- String
- Hashes
- Lists
- Sets
- Sorted Sets.

- Hashes are maps b/w string fields & string values.

e.g. HMSET user id 45 name "Andrei"

The key user, has two parameters/attributes, id and name.

HGET user id

HGET user name

- **Lists (Linked lists)**

LPUSH <key> <value 1> (left push)

RPUSH <key> <value 2> (right push)

GET <key> (will not work)

LRANGE ourlist 0 1

0 → start index
1 → stop index

RDROP <key> (To delete key)

- **Sets & Sorted Sets** (cannot have duplicate values)

SADD <key> <val1> <val2> <val3> <val4> ... (create set with these values)

SMEMBERS <key> (get command for sets)

SISMEMBERS <key> <value> (check if this value exists)
(will return 1 for yes and 0 for no)

ZADD <key> <val1> <val2> <val3> ... (Add for values in a key of type sorted set)

ZRANGE 0 1 (return specified range of elements)

ZRANK "value" (return rank of member)

Sorted according to int values or string weightage.

THE END