PostgresQL :

- → Create database = Create database "database name;
- → Drop database = drop database database_name;
- → Create table :

Create table actors (actor_id SERIAL PRIMARY KEY, ← Auto generated
(table name) first_name VARCHAR(150),
last_name VARCHAR(150) not null
);

Foreign key Syntax : director_id int referenced directors (director_id)
(directors = foreign table name; director_id = foreign column name)

Declaration of primary key for both col's : primary key (movie_id, actor_id)

- → delete table : drop table table_name;
- → Insert values in table : Insert into customers (first_name, last_name) ← table_name
Values
('Ramlan', 'reddy'),
('sai', 'divya'),
(' ', ' '),
(' ', ' ');

Returning * ;
↓
It will give all the values after query (not like query executed)

- → Insert data which has quotes ⇓ → additional quote
('Bill''O Sullivan');
output
Bill 'O Sullivan

- → Update the table : Update table name
set column name = 'new value', set column2 = ' ',
where column name = 'value';

- → Upsert (Update + Insert) : Insert into tablename (column list)
values (value_list)
on conflict target action;

Order by with null values: nulls first

→ Select * from users
order by salary desc nulls last;

If we put desc option, it sets the nulls last by default.

In postgres, the default behaviour of 'order by' is to treat 'null' values are greater than any non-null values when sorting in asc order.

→ and as smaller than any non-null values in desc order.

In asc order → null (end of sorted result)
In desc order → null (start of sorted result)

Distinct values: Select distinct column from tableName

We cannot uses alias names in where clause.

select firstname as name
from table
where name = 'x' → It is invalid.

Offset: Start number of OFFSET from number.

fetch: It use to fetch portion of rows.

Offset start (ROW|ROWS)
Fetch (First|Next) [row_count] {ROW|ROWS} ONLY.

like & ilike; → Case-sensitive
→ Not case sensitive.

Is null and is not null:
Select * from actors
where date_of_birth is null
or first_name is null
→ list of actors with missing birth date or missing first_name

Concatenation techniques: ('||', Concat(string s1, string s2), Concat_ws('|', col1, col2)

→ select 'Hello' || 'world' as new_string.
→ Select Concat(first_name, ' ', last_name) as "Actor_Name" from actors
→ select concat_ws(' ', first_name, last_name) from actors
↳ print first name + last name

Problem table: 'it', 'if', 'o', 'p'.

Char, varchar and Text:

Char → when data is fixed length (like AP, UK, US, IN)

Text → unlimited

Numeric → not hold null value.
(Numbers / data type)

Date & Time:

① Date → Date only (4 bytes) [YYYY-MM-DD] {Current_date

② Time → Time only (8 bytes) { HH:MM / HH:MM:SS / HHMMSS } { HH:MM:SS.PPPPPP / HH:MI:SS.PPPPPP / HHMMSS.PPPPPP }

③ Timestamp → date+time

④ Timestamptz → date, time and timestamp.

⑤ Interval (difference b/w dates will be stored in interval)

↳ Insert_into

Create table table_dates (
id serial primary key,
hire_date date,
add_date date default current_date
);

Current date+time: select NOW(); → {2021-01-06 18:58:40.55:-05

Time

② Time: column_name TIME(precision)

→ Getting current_time: select CURRENT_TIME; (12:10:26.823628-05:00)

input(4) → precision then it will be 4.

→ local time: select LOCALTIME; (local machine time)

→ Arithmetic operations: 04:00, 10:00

select time '10:00' - time '04:00';

select current_time + interval '2 hours' as result;

③ Timestamp & Timestamptz

Create table table_time_tz (
ts Timestamp,
tstz Timestamptz);

→ show TIMEZONE; → Kolkata.

Let's change Timezone

SET TIMEZONE = 'America/New_York';

→ Convert Timezone to Time based on timezone:

Select ( )

④ → select current_Timestamp;
select timeofday();

Scanned with CamScanner

UUID: (Universal unique Identifier)

↓ 128Bit

40e6215-b5c6-4896-987c-f30f ...

Create extension if not exists "uuid-ossp"; //installing uuid module for uuid-ossp.

Select uuid_generate_v1() //generating uuid

Create table table_uuid (
product_id UUID default uuid_generate_v1(),
product_name varchar(100) not null
);

Change uuid: Alter table table_uuid;
Alter column product_id
set default uuid_generate_v4();

hstore: key-value pairs (text string only)

Create extension if not exists hstore;
Create table table_hstore (
book_id pk;
book_info hstore;
);

→ Insert into (table, book_info)
('Title 1',
'publisher" => "ABC publ",
"pagecount" => "10-00",
);

↓ Fetching

Select book_info → 'publisher'
from table_hstore; → book_info → 'pagecount'

Json:

Create table table_json (
id serial primary key
docs JSONB.
);

Insert into table_json(docs) values
('[1,2,3,4,5,6]'),
('[2,3,4,5,6]'),
('{"key":"value"}');
"key1":"value",

→ select * from table_json
where docs @> '2', → operator.

Create index on table_json using GIN (docs jsonb_path_ops);

Network Address types:

cidr → 7 or 19 bytes (IPv4 & IPv6)

inet → 7/19 bytes (" ")

macaddr → 6 bytes

macaddr8 → 8 bytes

```
Create table table_netaddr (
    id serial pk,
    ip INET )
```

{ Set_masklen (ip, 28) — /24. /28.

inet to cidr: ip,
set_masklen (ip::cidr, 24) as cidr_24
(ip::cidr → normal to cidr.)

Constraints:

→ Alter table web_links
add constraint unique_web_url unique (link_url); (link_url → column.)

Datatype Conversion:

→ Implicit → Automatically.

→ Explicit → Conversion functions { CAST (expression AS target_datatype);
CAST ('10' AS INTEGER)

{ expression :: type.
'10' :: Integer,

Table data Conversion:

```
select rating_id,
    CASE WHEN rating ~ E'^\\d+$' THEN CAST (rating AS INT)
    ELSE 0
    END as rating.
```

(E'^\\d+$' → numeric data.)

Create domain type? (Re-use in multiple columns)

← Create DOMAIN name as data_type constraint

Ex:
- create domain proper_email varchar(150) check (value ~* '[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+[.][A-Za-z]+$');

- drop domain positive_numeric cascade.
  ↳ It also deletes depending objects too.

Composite datatypes: Create type name as (fields d/s - properties)

Create table companies(
  comp_id pk,
  address address,
);

Insert into companies(address)
values (row('New York', 'US')).

select (address).city from companies;

Create type address as(
  city varchar
  country varchar
)

Alter a composite data type: (Alter type myaddress rename to my_address)

Alter owner ; Alter type name owner to postgres
or
admin

add any attribute? ⇒ Alter type address. add attribute
street_address varchar(150)

Enum: Create type status_enum as enum ('queued', 'waiting', 'running', 'done');

Create table jobs (
  job_id
  job_status status_enum default 'pending'
)

Insert into jobs (job_status)
values ('running')

* Null is not same as empty string or Integer.

→ Alter table table_name
add constraint yourname unique (col1, col2)

Al, Apple ✓
Apple, Al ✓
Apple, A ✓

→ Alter table table_name
alter column is_enabled set default 'N' } → update the default value

Update foreign key Constraint:

// first drop constraint
Alter table tablename
drop constraint cname;

// Query of update
Alter table t_products
add constraint ~~t_products~~ foreign key (column name) references
cname tablename(colname);

Check Constraint: (useful more when inserted or updating data)

Create table staff(
staff_id PK,
first_name varchar(50)
Birth_date date check (birth_date > '1900-01-10'),
salary numeric check (salary > 0),
joined_date date check (joined_date > birth_date)
);

→ Alter Table prices .. → name of Constraint
add constraints price_check
check (prices > 0 and discount >= 0 and price > discount)

Sequence: (To identify and make data unique)

→ Create Sequence if not exists test_seq → sequence name
→ select nextval ('test_seq') (when running again to again it will increasing gradually)
→ select currval ('test_seq') (to get current value of our sequence)
→ select setval ('test_seq', 100) do not skip over.
→ Select setval ('test_seq', 200, false)
→ start with 100;
→ Alter sequence test_seq restart with 100.
→ " " rename to my_seq.

(Creating)
Increment 50
minvalue 400
maxvalue 6000
start with 500;

→ select nextval ('test_seq') ⇒ 550 (500+50)

Create sequence seq_desc
Increment -1
Minvalue 1
Maxvalue 3
start 5
cycle; => we will reverse back

no cycle; -> It give an error, when cycle completed

-> Drop sequence sequence_name;

Attach sequence to table: (column must be serial)

```
Create table users (
    user_id serial PK,
    user_name varchar(50));
```

-> Internal working:
nextval (user_user_id)
table name - column name

List all sequence: select relname sequence_name
from pg_class -> contain all sequence
where relkind = 'S';

Share one sequence b/w multiple tables:

-> Create sequence common_fruits_seq start with 100.

```
create table apple (
    fruit_id int default nextval(' ') not null,
    fruit_name varchar(50)
);
```

```
-> create table mango (
    fruit_id int default nextval(' ') not null,
    fruit_name varchar(50)
);
```

Create Alpha-numeric sequence (ID1, ID2 etc)

Create sequence table_seq;

```
Create table Contacts (
    contact_id Text not null default ('ID' || nextval('table_seq'))
    contact_name varchar(150).
);
```

concat

Alter sequence table_seq owned by Contacts.contact_id
Table name, col name.

=> date_part(field, source)
It is used to retrieve subfields (month, year, day etc)

date_part('year' Timestamp '17-01-01'); // 2017.
return double precision.

=> date_trunc('datepart', field)
date_trunc('hour', Timestamp '2024-10-01 5:15:45'); // 2024-10-01 05:00:00
until hour precision

String functions:

UPPER( ), LOWER( ), INITCAP( ) → proper case / titlecase

LEFT( ) → returns the first n characters in the string.

select left('ABCD', 1) ⇒ A (If n=10, then output = ABCD)

If n is -ve, return expect last 'n' characters.

('ABCD', -2) ⇒ AB

Right() → returns the last n characters in string.

↳ if n is -ve, return except first 'n' characters

Reverse() → Reverse('Amazing postgresql') { Output : lqs.erg... A

Split_part(string, delimiter, position) → Split('1,2,3', ',', 2) ⇒ 2

Split('one,two,three', ',', 3)

↳ three

- Get the release year of all movies

select movie_name, release_date,

split_part(release_date::text, '-', 1) as release_year

from movies;

Trim → Removes the longest string that contains a specific character from a string

LTrim → LTrim(string, [character]); left hand side.

RTrim →

BTrim → LTrim + RTrim

select LTrim('yummy', 'y'); ⇒ ummy

RTrim → yummy

BTrim → umm

RPAD

LPAD(string, length, [, fill]) → Analysis of chart is doubt → Optional (default = space)

select LPAD('Database', 15, '*'); → *****Database

Total length

space

length(string): length('') ⇒ 1 / length(null) → null

position(substring in string): select position('is' in 'This is a computer') ⇒ 3

strpos(<string>, <substring>): select strpos('World Bank', 'Bank') ⇒ 7

→ substring('what a wonderful world' from 1 for 4); ⇒ what

substring(. for 7) ⇒ what a (from starting pos)

pos, length

substring(string, start_pos, length)

repeat('A', 4) → AAAA

replace(string, from_substring, to_substring) ⇒ replace('ABCDCYZ', 'Z', 'X') ⇒ ABCXYI

Date/Time functions:

→ System wants date settings: either 4 factors

Show DateStyle; // ISO, MDY {DEC, Postgres, SQL (or) German}

set DateStyle = 'ISO', 'DMY'; → {MDY, DMY, YMD}.

Time & day formats in Postgres

HH:MM → all balls of time

now() → date, time, timestamp

today, tomorrow, yesterday, epoch → date, timestamp

infinite, -infinite

Strings to date Conversion:

TO_DATE (date, format) → (like YYYY, YYY, Month etc)

select ('2020-01-01', 'YYYY-MM-DD');

('20200101', 'YYYYMMDD'); // 2020-01-01

Using TO_TIMESTAMP function: (String to timestamp)

TO_TIMESTAMP (timestamp, format) → (like 'YYYY, YYY, MON, Mon etc)

Select TO_TIMESTAMP ('2020-01-01 10:30:20', 'YYYY-MM-DD HH:MI:SS');

Formatting Dates:

① TO_CHAR() converts a timestamp, interval, integer, doubleprecision, numeric value to a string.

TO_CHAR (expression, format)

→ TO_CHAR ('2020-01-01 10:00:00'::TIMESTAMP, 'YYYY Month DD');

// 2020 January 01.

→ TO_CHAR ('2020-01-01T10:45:55-6:00'::TIMESTAMPTZ,

'YYYY Month DD HH12:MM

Date Construction Functions:

→ MAKE_DATE (YYYY, MM, DD)

Select MAKE_DATE (2020, 01, 01) // 2020-01-01

→ MAKE_TIME (HH, MM, SS)

Select MAKE_TIME (2, 3, 4.05) // 02:03:04.05

→ MAKE_TIMESTAMP (YYYY, MM, DD, HH, SS)

→ MAKE_INTERAL (years, months, weeks, days, hours, minutes, seconds)

select make_interval(2020,01,01,01,10,30,45) // 2020 years 1 mon 8 days 10:30:45

→ MAKE_TIMESTAMPTZ (2020, 02, 15, 10, 35, 15.38, 'US/Alaska');

// 2020-02-15 17:35:15-05.

Alaska

Date value Extractor:

→ Extract (field from source) { select extract ('DAY' from current_timestamp)

→ DATE_PART (field, source)

math operations with dates:

select date '20200101' + 10 // 2020-01-11.

or

select '20200101'::date + 10 // 2020-01-11.

select time '23:59:59' + Interval '10 second' // 00:00:09.

select current_timestamp + '01:01:01' // 2020-11-27 19:40:29.5071-05.

select '10:10:10' + TIME '10:25:10' // 20:35:20

select DATE '20200101' + Interval '1 hour' // 2019-12-31 23:00:00.

select (DATE '2020-01-01', Date '2020-12-31') overlaps (Date '2020-10-12', Date '2020-12-01') // true

sql:

Select current_date, current_time, current_timestamp, localTime, localTimestamp

postgres:

select now(), transaction_timestamp(), statement_timestamp(), clock_timestamp(), timeofDay()

AGE(date1, date2) → (returns diff b/w date1 & date2) → String format

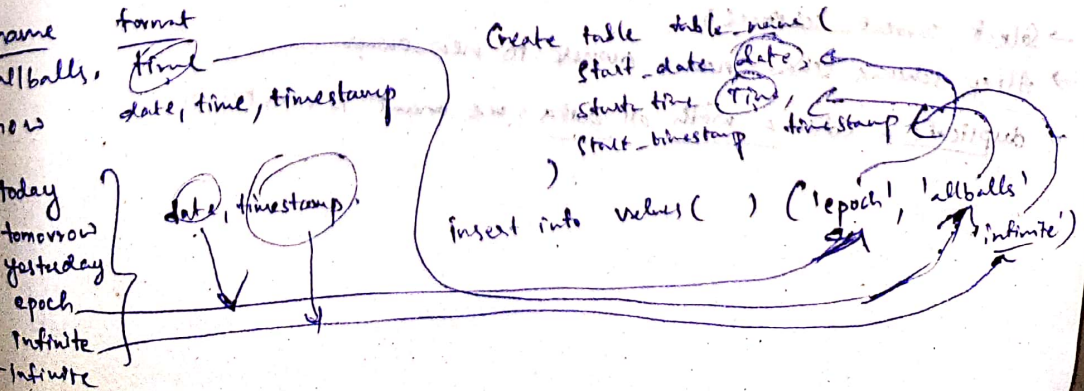
AGE ('2020-01-01', '2019-10-01') // 3 mons

age (current_date, timestamp '2020-01-01');

select (Extract (epoch from Timestamptz '2020-12-20') - Extract (epoch from Timestamptz '2020-10-20')). // 5274000.

1970-01-01

60/60/24 // 610466



Handling Null values in group By:

- Coalesce (source, '')

Coalesce (department, 'No department')
(Col name) (If null, then no department should display.)

```
Select
    Coalesce (department, 'No department')
    Count (salary) as
from employee_test
group by department.
```

Union, Union all.

Intersects: order, no of cols and datatypes for both tables should be same.
↓
It gives common data from both tables.

Except: select col1, col2
from table1
except / intersect
select col1, col2
from table2

Schema: It is a namespace that contains named database objects such as table(s), views, indexes, datatypes, functions, stored procedures and operators

→ Create schema sales (we can't create duplicate schemas in one DB)
→ Alter schema sales rename to programming.
→ Drop schema programming

Schema Hierarchy:

physical: host (cloud/localhost) > cluster (PG) > database > schema > objectname.

object access: database.schema.object_name.

Eg: hr.public.employees → table_name.

→ Move table to new schema ⇒ Alter table table_name set schema schema_name

→ Select current_schema()
→ Alter schema schema_name owner to new_owner
→ duplicate a schema with all data: we cannot to execute in

Array functions:

→ Array [value1, value2, value3, ...]

- select array[1,2,3]
  array['2-12-23'::date]
  array[current_date, current_date-5]

Comparision: =, <>, <, >, <=, >=
(Not equal to)

Inclusion: @> , <@ , && → overlap;
(contains) (Contained by)

Array constructors: All Array[1,2,3] => {1,2,3}
Array_prepend(4, Array[1,2,3]) => {4,1,2,3}
Array_append(Array[1,2,3], 4) => {1,2,3,4}
Array_prepend(element, Array)

Array_NDIMS(Array[[1,2,3],[4,5,6]]) = 2

Select Array_length(Array[1,2,3,4],1) = 4
Array_lower(Array[1,2,3,4],1) = 1
Array_upper(Array[1,2,3,4],1) = 4
Cardinality(Array[[1,2],[3,4]]) = 4 (dimensions inside)

search:
Select Array_position(Array['Jan','Feb','March'], 'Feb') = 2

Json: (name-value pairs)

[{objects}]

"firstname" : "Adnan" , "Country" : "India"

{"firstname": "Adnan", "Country": "USA"}

JSONB ✓ (preferable) → as this faster and provide full text indexing and fast indexing.

JSON

JSON document in Binary format

↓

select '{"title": "The Lord of Rings"}'::json

::jsonb

select book_info ->> 'title'

from books

where book_info ->> 'author' = 'author1'

Update & delete json data:

- Update books (add books)

set book_info = book_info || '{"author": "Adnan"}'

where book_info ->> 'author' = 'author10'

delete:

Update books

set book_info = book_info - 'bestseller'

Tables into Json format: (Column name / entire table)

select row_to_json(directors) from directors

json_agg():

json_build_array(values) → json_build_array(1,2,3,4,5) // [1,2,3,4,5]

json_build_object(1,2,3,4,5,6); // {"1":2,"3":4,"5":6}

json_object({keys},{values})

Null values in json docs:

Index? (It help to improve the access of data in our databases) but add a cost

Tuple: index, data

We can create indexes for 32 cols at a time

By default access method is btree

Create index index_name
on table_name (col1, col2, ...)

Create unique index index_name
on table (col1, col2, ...)

Create index index_name on table_name [using method]
(
column_name [ASC|DESC] [NULLS {FIRST|LAST}],
...
);

-- lets create an index on order_date on orders table.
col_name.

create index idx_orders_order_date on orders (order_date, ship_city);

Order of fields are also imp.

→ If we create index, the db won't be able to create duplicate records on existing records.

List all indices:

select * from pg_indexes where schemaname = 'public'

The table contains all the indexes by default in postgres

→ obj(schemaname, tablename, indexname, tablespace, indexdef)

size: select pg_size_pretty (pg_indexes_size ('orders')); //200KB.

→ If there is no indexes also, there will be some size (which are internal files of table).

whenever we add new index, it occupies disk space

list counts of all indexes:

select * from pg_stat_all_indexes

It contains all stats about table indexes

(relid, indexrelid, schemaname, relname, indexrelname, idx_scan, idx_tup_read, idx_tup_fetch)

Drop index: drop index [Concurrently] [If Exists] index_name [cascade | Restrict]

By default (rewrite in machine level)

One node output will be another node input.

Nodes are stackable ::

Various types of nodes:

Select * from pg_am → gives all type of nodes

→ Sequential scan: (default one).
- Read from Beginning of dataset
- Highly cost,

→ Index Nodes:
- Index scan: (when we have indexes on the columns and fetching that column).
- Index only scan: (When we selecting column and table which have indexes)
- Bitmap index scan:

Join Nodes:
- Hash Join
- Merge Join
- Nested Join

(4) Stages of execution for sql:
(1) Parser (handles the textual form)
(2) rewrite (applying any syntactic rules)
(3) optimizer (finding the fastest path to get data that statement needs)
(4) Executor (responsible for going to storage and retrieving data).

parameters: (Cost, Threads, Nodes, Nodestype)

Node1, Node2

(It divides query actions into individual nodes)

Types of Index:

- B Tree Index (By default):
  - → Self-balancing tree
  - → All operators (=, >=, <=, <, >)

Hash Index:

- → ('=')
- → larger than btree indexes.

(create index index_name on table_name using hash (col_name);

BRIN Index:

- → Block range Index.
- → used linear sorted order

GIN Index:

- → point to multiple tuples

Explain statement: (Analyze and execution plan of query)

(It only gives info, not run the query)

Seq Scan on suppliers (cost=0.00..1.36 rows=1 width=144)

- Scan type node (Sequential scan)
- startup cost
- final cost
- rows=0 (tuples)
- Bits occupy by tuple

{If we see we have cost and row, that is node...}

Explain (format json) select * from orders where order_id=1

Explain analyze statement: (It runs and give information about query)

→ Execution depends on input values:

Explain Analyze select * from t_big ⇒ Index scan

" " t_big " n ⇒ Index only scan

Partial Index: to improve the performance of the query while reducing index size

Create index index_name
where conditions

Expression index: Create index index_name
on table_name (expression)

Adding data while ~~Creating~~ indexing: create index concurrently

Rebuild Index:

reindex [(verbose)] {Index | Table | Schema | Database | System} [Concurrently] name

Views: (You can save your query into a view, so instead of writing long queries, you can just refer to a view.)

(Virtual table) ⇒ Regular views do not store any data except materialized views

→ Create or replace view view_name as query

Create or replace view v_movie_quick as (select movie_name, movie_length from movies mv);

→ We can't create views on duplicate columns (when join condition)

Use view: Select * from v_movie_quick);

Rename: Alter view view_name Rename to new_view_name;

Drop: Drop view view_name;

Re-arrange existing view cols: delete the existing view and then create new view for re-arranging cols.

Remove a col from an existing view:

→ Postgres doesn't support removing an existing col in a view.

Add a col in a existing view: we can't change order of a view.

Regular view as dynamic: It does not store data physically

It always give updated data

→ If we delete one data in query, then it auto-matically delete the record in view also

Updatable view:

→ No query conditions are there

→ Cannot contain distinct, group by, with, limit, offset, union, intersect, except, having at top level of query. (or window functions, set-returning functions, aggregate functions)

→ If we update in view, then it update in table and vice-versa.

With check option:

Local check option: It add in local, but not in view? (But it execute the code)

Cascade check option: we can't add this view itself

Materialized view: It stores results of a query and update data periodically.

↳ Create materialized view if not exists as query with [no] data; ↳ with data (query output will go into materialized view)

↳ with no data (refresh ...)

→ Drop materialized view view_name;

Insert: → First add in table → Refresh materialized view → Then see (it will update)

Check materialized view populated (yes) not:

→ Refresh materialized view (mv_name);

Select relispopulated from pg_class where where='mv_name';

List all mv's: select oid::regclass::text from pg_class where relkind = 'm';

Text to structured data:

→ Case formatting : Select upper ('world'); // WORLD | select lower('World')

Select initcap ('world') // World | Select char_length('world') // 5

↓ Initialize the first capital (not)

[A-Za-z0-9_] (use in: csv file)

Regular expressions: ([a-z]3[a-z] [A-Z], \w, \d, \t, \s, \n, \r,

\d → any digit, \t → tab character, \s → space, \n → new line character

? → zero (or) one time.

* → zero (or) more time.

+ → one (or) more time.

{m,n} → match between m and n times.

{n} → exactly 'n' times / {m,n} → match between m and n times.

a|b → either a or b / (? :) ⇒

| Similar to | ≃ | LIKE |

POSIX regular Expression:

→ ~ (Match regular Expression, case sensitive)

~* (" , case insensitive)

!~ (Does not match regular expression, case sensitive)

!~* (" , case insensitive).

Select 'same' ~ 'same' // true.

'Same' ~* 'Same' // true (case insensitive)

→ select substring ('The name' from '.'); // T (first character)

'.*'); // the name (all characters)

→ REGEXP_MATCHES (source-string, pattern, [, flags]). // returns text

→ select REGEXP_MATCHES ('Amazing #Postgres', '#([A-Za-z0-9_]+)'); // postgres

↓ #postgres, #sql , 'g');

↓ multiple matches

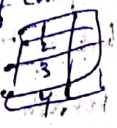
→ REGEXP_REPLACE (source, pattern, replacement-string, [, flags]).

→ select ('Kundan Reddy', '(.*) (.*)', '\2 \1'); // reddy kundan

space, space

→ REGEXP_SPLIT_TO_TABLE ('1,2,3,4', ','); // split by means of comma into rows

1
2
3
4

→ REGEXP_SPLIT_TO_ARRAY ('kundan, Reddy', ','). // {kundan, reddy}

Full-text search: 'query' → queries (we will not able find 'queries' word by writing 'query' single word. we can find by full-text search).

↓

Full text data types:

1) tsvector (text to searched and stored in optimized format)
    └→ lexemes (Reduced texts to lexemes Eg: washes, washed etc to wash).

2) tsquery.

1) to_tsvector() ⇒ Select to_tsvector('washes') // 'wash':1

('The quick brown fox jumped over the lazy dog.');

// 'brown':3 'dog':9 'fox':4 'jump':5 'lazi':8 'quick':2

(Since articles and unused words are removed like the, a, over etc and words reduced to lexemes (jumped → jump)) → (high or create) → (low to high) → {Lick}:2, {quick}:3

Operators: @@ , &, |, !, <-> , <2> : <3> etc { we can mention numbers i.e the words in between }

@@ ↓ match operators

<-> ↓ search for adjacent words or words a certain distance apart.

→ Select to_tsvector('The quick brown fox jumped over the lazy dog')
@@ to_tsquery('foxes') { searching foxes in above text. } searching foxes in text.

('fox and dog') // true

query:

select doc_id, doc_text
from docs
where doc_text_search @@ to_tsquery('jump');

↓ tsvector()

ts_rank(speech_text_search, to_tsquery('military & defense')
)

↑ to_tsvector()

Table partitions: (If we have huge table, then to improve performance, we partitions tables).

Table inheritance:

Create table master ( ) => Create table master_child ( ) inherits (master);

);

→ Master table takes the data from master_child table. → Here it is child of.

→ If we want to drop table, first child table and then master table.

Types of partitions: Range, List & Hash.

Create table table_name (col_name

) partition by Range (Birth_date);

Create table employees_range_y2000 partition of master_table for values from value1 to value2.

('2000-01-01') ('2001-01-01');

→ When we do partition for one (main table), then when we insert data into main table, then it goes to partition table instead of main table.

ONLY syntax

List: Create table employee_list (col_name

) partition by LIST (Country_code)

Create table partion_table_name partition of master_table for values in (field); ('UK', 'DE', 'IN', 'FR');

Hash partition: (When we can't logically divide our data).

Create table employee_hash (=> Create table partion_table_name partition of master_table

) partition by Hash (id); for values with (modulus 4, Remainder 0, 1, 2, 3);

(It evenly distributes the data, there is no logic)

(It will not divide/partition by id, it will partition by remainder) No. of tables.

~~It modulus~~;

Default partition: (What happen if there is no partition table for particular inserting record. Eg: partition_table_US. But we are inserting 'UK'. Then it will go to default partition)

→ Create table partition_table_name partition of parent_table_name default;

Multilevelpartitioning:

US → list
EU → list
→ EU_1 → Hash
→ EU_2 → Hash

Create table employees_master_us partition of employees_master for values in ('US');

Create table employees_master_eu pov values in ('UK', 'DE', 'IT', 'FR', 'BE') → partition by hash(id);

Create table employees_master_eu1 partition of employees_master_eu for values with (modulus 3, Remainder 0);
(mod 2, Rem 1)
(mod 3, Rem 2)

Detaching: (master)
→ Alter table table_name detach partition partition_table_name

Altering the Bounds of partition:

0-100, 200 to 300 (Here we miss 100-200);

Create table t1 (id int,) partition by range(a);

Create table t1p1 partition of t1 for values from (0) to (200);
" " t2p1 " " t1 " (200) to (300);

① Detach
② Alter
③ Attach
→ Begin
Alter table t1 detach partition t1p1;
" " " attach " " for values from (0) to (200)
Commit transactions.

Partition Indexing:

→ Create index on master table, it will automatically create same indexes to every attached partition

→ Create unique index idx_uniq_name on employees_list (id, country_code) → mandatory;

Partition pruning: (when it is on, then partition key is used to identify which partitions the query should scan), (it scan only partition table). (Because we partitioning the table by country code).

→ If it is off, it scan all tables (master and partitions) when where condition

Procedural languages: (Stored programming)
(It doesn't return values like functions, but operates like functions)

Postgresql Security:

- System administrator can grant access/revoke access
- Roles are everything → Groups → Individual users

Levels of Security
→ Instance level
→ DB level
→ Schema level
→ Table level
→ Column level
→ Row level.

① Instance level security: (Highest level)

CreateDB - can create DB
CreateRole - can make roles
login - can login into DB
Replication - can be used for replication
SuperUser - Super access, all access.

Eg: Create role programmer noSuperuser noCreateDB noCreateRole nologin

② DB level security:

Create - Make a new schema
Connect - Connect to DB
TEMPTemporary - Create a temp table

Eg: Grant connect on Database db_name to role

③ Schema level:

→ Grant permission ON SCHEMA schema_name to role_name.

④ Table level:

Select, Insert, update, delete, truncate, Trigger.

Grant select on table table_name to role

⑤ Column level:

Functions:

Create or replace function function_name() returns void as
' -- SQL command '
LANGUAGE SQL.

1 → Create or replace function fn_mysum(int, int)
Returns int as
$$ select $1 + $2; $$ → $body
language sql.

Select fn_mysum(1, 2); // 3
(20, 30); // 50

Dollar quoting: ($body)

Function using parameters:

Create or replace function function_name(p1 type, p2 type ...)
return_type as $$ we can access $1 & $2 as p1 ; p2.
$$
language SQL

→ Create or replace function fn_mid(p_string varchar, p_starting_point integer)
returns varchar as
$$ select substring(p_string, p_starting_point); $$
language sql.

→ select fn_mid('Amazing postgresql', 2) // mazing postgresql

$$ select count(*) from customers
where city = p_city
$$ (p_city varchar) ...

Composite return type: returned row with, separated values.

→ (function_name()).* → Convert table format.
→ (function_name()).field_name → If we want only get particular field
→ fieldname(function_name()) →

Function returning multiple rows: "Returns setof"

Create or replace function
(p_year integer) returns setof Employees as
$$
$$
language sql

Function as table; Select column-list
from function_name();

If order of parameters matters!

Function returns a table (once i must return all cols.)

Create or replace function fn_top_orders (p_customer_id bpchar, p_limit integer)

Returns Table (
order_id small int,
customer_id bpchar,
product_name varchar,
)

AS

$$

$$
language sql.

we start default, then following parameters are default.

Function Default parameters:

Create function function_name (p1 type Default v1, p2 type default v2)

Eg. Create or replace function fn_sum (x int, y int default 10, z integer default 10) returns integer as

$$ select x+y+z;
$$
language sql

-> select fun_sum(1,2,3) 4 6
" (1) 1 + 10 + 10 = 21.

Function Based on a view:

Create or replace function fn_active_queries (p_limit int) returns setof v_active_queries as
(view)

$$

$$
language sql.

Drop function: Drop function [IF EXISTS] function_name (arg list)
[cascade|restrict];

Pl/pgsql language: (SQL scripting language)

Pl/pgsql (vs) SQL
↓ executes individually

↓ Executing multiple statements combinely

(It stores multiple queries as object and whole object executing combinely in server)

Create function function_name (p1 type, p2 type...) returns return_type as
$$
Begin
... statements
end
$$
language plpgsql

select → return.

Block structure syntax:

```
DECLARE
  variable_name data_type[:= expression];
BEGIN
  RAISE NOTICE 'my variables %,%.%.%'
END;
```

It is like print statement.

Alias: newname Alias for oldname;

Declaring variables in functions:

→ using position numbers($1, $2) — first argument, 2nd argument

→ using Alias (x Alias for $1, y Alias for $2)

we declare $2 as y.

Copying datatypes:

→ %type → refers to datatype of a table column or another variable.

```
Declare
  variable_name table_name.column_name %type;
```

Assign variables from query:

→ select expression into variable_name(//must return only a single result)

*, col_name

```
Select * From products into product_row limit 1;
Select product_row.product_name into product_name;
$$
 Declare
  product_only_name products.product_name%type;
 Begin
  select product_name from products into product_only_name
  where product_id = 1 limit 1;
  Raise notice product_only_name;
```

// ~~declare~~ access full row ('record' keyword)

declare
row_product record;

Begin
select ~~product_name~~ * from products into row_product ~~product_title~~.
where product_id = 1 limit 1;
RAISE NOTICE row_product.product_name; // obj

IN, OUT, INOUT, without returns:

create or replace function fn_sum (IN x integer, IN y integer, OUT z integer) as

input variables — output variable

We can 'out' more than 'one' variables.

```
$$
BEGIN
  z := x + y;
END;
$$ language PLPGSQL;
```

Nested Blocks:

```
Block_1
  Declare
    v1 int;
  BEGIN
    Block_2
      Declare
        v1 int;
```

→ referencing variables:
Block_name.variable_name;

How to return query results:

create or replace function function_name returns setof table_name as

setof means all cols → So, we want to use only * not col names

```
$$ Begin
  return query select ..
end;
$$ language plpgsql.
```

IF Condition:

If Boolean-expression then
//statement 1 //actions
elsif Boolean-expression then
//statements
else
//statement 3
end if;

Case: → simple (list of values)
→ searched (range of values)

List:
CASE expression
WHEN expression THEN
//statement
WHEN expression THEN
//statement
ELSE //statement
END CASE;

Range:
Case expression → we don't need

Loop:
LOOP
//statement
exit condition (or) EXIT; (or) EXIT WHEN
END loop;

Continue [loop label] [when condition].

FOR LOOP:
FOR [counter name] IN
[REVERSE] [START VALUE]..
[END VAVE]
[BY STEP]]
LOOP
[statements];
END loop;

BEGIN
LOOP
EXIT WHEN CONTINUE condition()

Forcach loop:
forcach var in Array array-name
LOOP
//statements
END LOOP.

While loop:
while your-condition
LOOP //statements
END loop;

Using return table Return table (Col list)

→ Create or replace function fn-products (p-pattern varchar) returns Table(
product_name varchar, unit-price real) AS
↓ diff name as col name to avoid confusion)

$$ BEGIN
Return query
Select product_name, unit-price from products
where product_name like p-pattern;
$$
END.

Return next: appending function result.

Error and Exception handling: Exception
when NO_DATA_FOUND THEN
RAISE EXCEPTION 'No order found';

Too many exceptions: SQLstate '9002'
Exception
when too_many_rows then
Raise Exception 'Your query returns too many rows';

Data Exception - error: division-by-zero.

Functions vs stored procedures:
→ User defined functions cannot execute transactions
→ If return values
→ support transactions (doesn't return values) 'transactions'.
→ may or may not use parameters.
→ May or may not have declaration functions.

Create a transactions: $$
Begin
statements
commit;
End;
$$

use of stored procedures:
→ To ensure data consistency
→ Security
→ Modularity

→ To return a value from stored procedures we use 'INOUT' parameter mode.

Create or replace procedure pr_orders (INOUT total_cost integer default 0)
$$
Begin
End;
$$ language PLPGSQL;

Drop a procedure/stored procedure: Drop procedure [if exists] procedure_name (arg list) [cascade | restrict].

Trigger: (It is an event) {Use to basic auditing}.

→ It is a function called automatically when an event associated with table occurs. (Insert, update, delete, truncate)

→ special user-defined function

→ Diff between user-defined func and trigger is, trigger is automatically called when a triggering event occurs.

→ Before, after, Instead
- Before: Before Insert, Before update
- After: After Insert

→ first All Before, row operations and after operations

Types of triggers:
- Row level Triggers (run every time row)
- Statement level Triggers (run only once)

Trigger Table:

When	Event	Row-level	Statement-level
Before	Insert/Update/Delete	Tables	Tables and views
	Truncate	—	Tables
After	Insert/Update/Delete	Tables	Tables and views
	Truncate	—	Tables
Instead of	Insert/Update/Delete	views	—
	Truncate	—	—

Key points:

~~Key points~~: No trigger on select statement, because select doesn't modify any row.

→ Multiple triggers can be used/runs in alphabetical order.

→ User defined functions are allowed in triggers

→ Single trigger can support multiple actions.

Creation of triggers

→ Create triggers using function statement

Insert/update/delete/truncate

```
Create trigger trigger_name {Before | After} {event}
on table_name
[FOR [EACH] {ROW | statement}]
```

Create procedure trigger_function.

```
Create function trigger_name()
Returns trigger AS $$ Begin
--- trigger logic
end
$$ language plpgsql;
```

de auditing with triggers!

Create or replace function fn_players_name_change_log ()
returns trigger
language plpgsql
as
$$
Begin
//compare new value vs old value
keyword
If New.name <> old.name then
Insert into players_audits (player_id, name, edit_date)
values (OLD.player_id, old.name, NOW())
row before update

//we want to update old values into players_audits.

end if
Return new

Insert, update → New.
Update, delete → old.

Bind this function to trigger table

Create trigger trg_players_name_changes
before update on players
for each row
execute procedure fn_players_n ()

modify data at insert event!

View Trigger variables: TG_NAME, TG_RELNAME, TG_TA etc.

Disallowing delete: (we cannot delete)

Creating audit trigger!
→ who changed data
→ when the data was changed
→ which operation changed the data (Insert, update etc)

Create or replace function fn_audit_trigger()
returns Trigger
language plpgsql
AS $$ declare old_row = null json
new_row = null json
Begin
IF TG_OP IN ('UPDATE', 'DELETE') THEN
old_row = row_to_json (OLD).
end if
('Insert', 'update') then
new_row = row_to_json (New);
Adds to table

Trigger
After insert or update or delete

Conditional triggers: (by using generic when clause)

→ we can't use subquery.

Create or replace function fn_cancel()
returns trigger
language plpgsql
As $$
Begin
RAISE exception 'TG_ARGV[0]';
Return null;
end;
$$

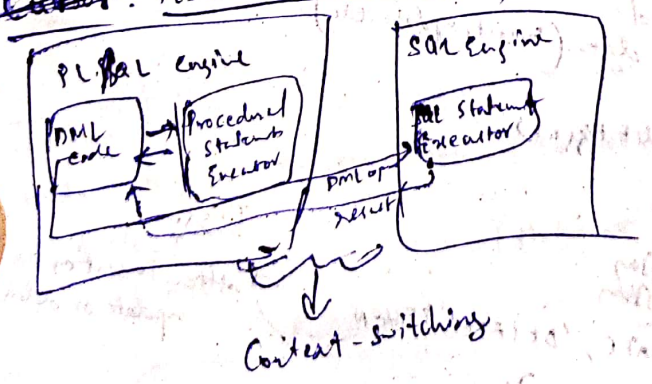
Create trigger tr_update
Before insert or update or
delete or truncate
on any table
for each statement level
when ( )

Use triggers very cautiously:
→ do not change data in: Primary key, Foreign key or unique key

Event triggers: (Before or after) → No instead of
→ we can't write in sql lang.
→ Use in audit trail
→ use to prevent changes to table.

Create event trigger: Create event trigger trg_name.
→ did not return any values
Create or replace function fn_event_audit()
returns event_trigger
language plpgsql → acting as a super user.
security definer —
as $$ Begin
end;
$$

~~Cursor~~ PLSQL Architecture: → Physical Architecture
→ Logical Architecture



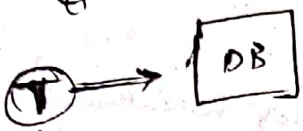
→ Cooperates with SQL Engine.
→ Enables subprograms.
→ Dynamic queries
→ Case insensitive.

Degree of table : = no. of Columns in table

Cardinality of table : = no. of Rows in table

Acid properties : (Why? : We want to maintain a consistent DB. Instead of maintaining a DB consistently we maintain a "process" that interact with DB. so that it remains consistent.)

~~Atomicity~~ :



(If transaction satisifies the ACID properties and interact with DB. It should be consistent)

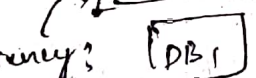
(1) Atomicity : ~~The~~ transaction should be ~~transaction~~ Atomic in nature

→ Either all the instructions in transaction must execute.

(or) none of them is execute.

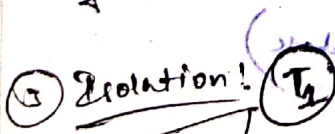
→ Transaction management Component manages the Atomicity in DB.

(2) Consistency : (It is responsible for system programmer) (If Atomicity, Isolation, Durable is consistent, then automatically consistent)



If the DB was consistent before the execution of transaction, then it should be consistent after the execution of transactions.

(3) Isolation : (multiple transactions)



If T_1 and T_2 executing simultaneously, it should not be effected by one another.

(4) Durability : If we have made changes in the DB and it we have updated the DB in new state and that changes must precise in the system irrespective of software (or) hardware failure

(Recovery management component) (Committed changes are permanent)

SQL: (Analytics)

→ What is SQL? (Structured Query language) → Interact with RDBMS.
→ Overview on SQL Commands
→ DDL & DML
→ Constraints & Data-types
→ select queries
→ Diff functionalities associated with select.

DDL: (Create):

To create table we have to understand datatype & constraints

Varchar, int, date, float
↓ ↓
Variables of Numbers & Splchrs
Any date format

Boolean → (True or False)

Constraints: Limitations/Restrictions
↓
(Maintain data integrity)

(Check) Age is positive
So we have to use Check constraint.

NOT NULL:

UNIQUE: (Help to identify any duplicate entry) (NULL values are allowed)

PRIMARY KEY: (Unique + Not null).

FOREIGN KEY: (Parent-child relationship).

Drop: Removes the database objects (such as table, view, function etc

Alter: Can be used rename a table/ columns.
modify column datatypes, add new columns, remove columns

DML : (Data Manipulation Language)

→ Insert : load data into table

→ Update : Modify data in table.

Update
set
where

→ Delete : Delete data from table

Commit : Execute COMMIT ; after every DML operation (Insert, Update, delete) to save the changes.

DQL : (Data query language) :

→ Select

→ Union :

It is used to combine two queries (Each query has same no of columns & datatypes)

It removes the duplicate values b/w two queries and gives the result.

→ Union All : It doesn't removes the duplicate values.

Difference b/w delete & truncate (DML) → DDL

faster (for millions records)

Difference between Distinct and group by :

Window Functions :

(we will assign our own values)

Syntax

① Row_Number :

```
select e.*
row_number() over () as rn
from employee e)
```

partition by dept_name → syntax

```
select * from (
select e.*
row_number() over (partition by dept_name) as rn
) x
where x.rn < 3;
```

→ it will fetch the 1&2 values of all department

(whenever coming to rank, it gives same rank to duplicate values and skip the next number.)

② Rank () : (Eg: fetch top 3 employees in each depart earning max salary)

```
select * from (
select e.*
rank () over (partition by dept_id order by salary desc) as rn
from employee e; ) x
where x.rn < 4;
```

Eg:

Salary	Rank
5000	1
4000	2
4000	2
2000	4

③ Dense_rank () : (In dense_rank, it gives same rank to duplicate values rank () but it doesn't skip any values)

Eg:

Salary	dense_rank
5000	1
4000	2
4000	2
2000	3

Record from previous row to current row

Default value if row has null in that column

Lag (salary, 1, 0) → Records from preceding to current record

Lead() / Lag():

(Eg: fetch a query to display if salary of employee is higher, lower or equal to the previous employee).

```
select e.*
lag(salary) over (partition by dept_name order by Employee_id)
as prev_
from employee e;
```

Lead():

(It is based on record from next rows to current record)

First_value():

accepts one argument (product_name that should be displayed)

(Eg: write a query to display the most Expensive product under each category corresponding to each record).

```
select *
first_value(product_name) over (partition by product_category order by price desc)
as most_exp
from product;
```

Last_value():

accepts one argument (column/record that needs to display).

(Eg: write a query to display the least expensive product under each category corresponding to each record)

```
select *
last_value(product_name) over (partition by product_category order by price desc)
as least
from products;
```

It doesn't display the proper output because of default frameclause using by sql.

Frame clause() : (Basically frame is a subset of partition. Window creates a partition.)

default frame clause() : range between unbounded preceding and current row.

using in sql.

functions like avg, sum, min, max, rows

Aggregate, last_value(), nth value will effects by frame_clause(). So to resolve this problem we want to change the default frame_clause() in the following way:

default frame_clause() : range between unbounded preceding and unbounded following

(will consider last value when it has duplicate values)

rows ↑

So,

```
select *
last_value(product_name) over (partition by pro_category
order by price desc range between
unbounded preceding and unbounded following)
as least_exp
from product;
```

→ Alternate query (writing a window function):

over w.

window w as (_ _ _)

nth value(()) : accepts 2 arguments (display product name of from each position)

(Eg: write query to display the second most expensive product under each category).

value of 2.

```
select *
nth_value(product_name, 2) over w as second_most_exp_product
from product
window w as (partition by product_category order
             by price desc range between
             unbounded preceding and unbounded
             following);
```

If we mentioned beyond n value it returns "null".

NTILE(): (equally segregated), accepts one agrument (no. of buckets) (group together data and place in certain buckets)

(Eg: write a query to segregate all the expensive phones, mid range phones and cheaper phones).

(It will give 1st bucket some priority when it can't divide by 3).

```
select *
ntile(3) over (order by price desc) as buckets.
from product)
where product_category = 'phones';
```

CUME_DIST(): (cumulative distribution) no args

0 to 1.

(Eg: Query to fetch all products which are constituting the first 30% of the data in products table based on price).

```
select *
cume_dist() over (order by price desc) as cume_dist
from product;
```

N/

percent_rank ( ) : → no args

Formula = $\frac{\text{Current Row No} - 1}{\text{Total no of rows} - 1}$

↓ 0 to 1.

Eg: Query to identify how much percentage more expensive is "Galaxy Z Fold 3" when compared to all products.

select *, round(percent_rank ( ) over (order by price) :: numeric * 100, 2) as percentage_rank
from product

(Sub query factory).

WITH CLAUSE : → (Common Table Expression (CTE)).

→ It is used to reduce the usage of same query multiple times.

with average_salary (avg_sal) as (select avg(salary) from Employee)   ↑ should be displayed

select *
from employee e, average_salary av
where e.salary > av.avg_sal;

} → Fetching employees who earn more than avg salary of all employees

Advantages of WITH clause

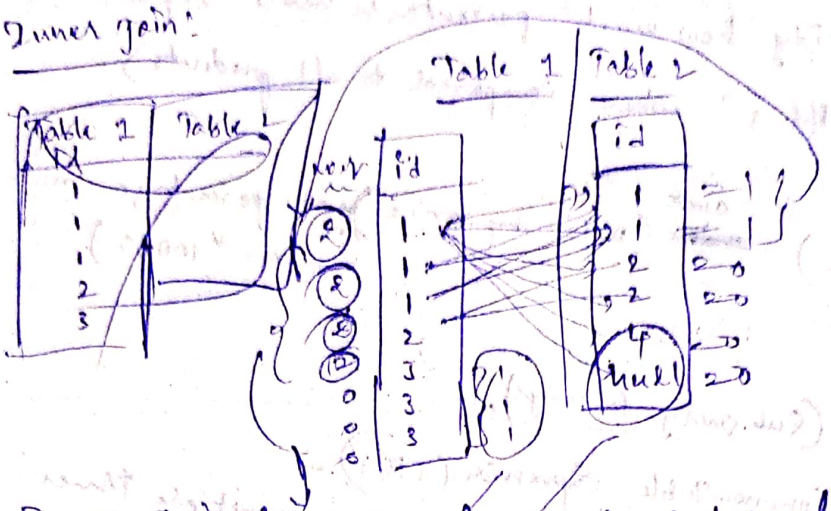
→ Improvement in performance
→ Something like a temporary table until it gets executed

(13)

Two null values are not the same -

Joins:

Inner join:



Inner join: 8 records can be returned.

Left join: Left join = inner join + fetch any additional records from left table which is not present in right table

8 + 3 = 11

(null values will be considered while we seeing matching)

Right join: inner join + fetch any records from right table which is not present in left table

8 + 2 = 10

full join = inner join + fetch additional records from left table which is not present in right table + fetch additional records from right table which is not present in left table

8 + 3 + 2 = 13

Natural join: If we do not mention the ON clause also, by default sql joins the two tables in means of common column (If same column name) (If not same column name it will try to do inner join)

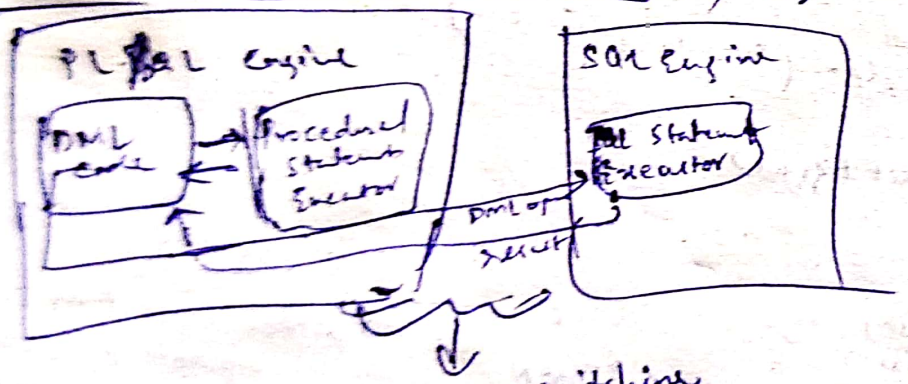
(Microsoft sql is not supported)

Cross join: no. of records in Table 1 * no. of records in Table 2

7 * 6 = 42 records

PL/SQL Architecture:

- Physical Architecture
- Logical Architecture



Context - switching

- Cooperates with SQL Engine
- Enables subprograms
- Dynamic queries
- Case insensitive.

Oracle PL/SQL Bootcamp

Anonymous Blocks: Declare → Begin → Exception → end.

PL/SQL Blocks:
- Declare section (optional) (Declaration of variables)
- Begin → mandatory (execution section) (sql statements)
- Exception Handling section → Optional → logical errors

Declare / Begin / Null / end; → Three types of Blocks:
- Anonymous Blocks (If we write code in a worksheet and run it directly, it is anonymous Block)
- Procedures (Must have return type)
- Functions

Plsql output:
- set serveroutput on
- DBMS_OUTPUT (It has some procedures & functions which help to get output)

```
set serveroutput on
begin
dbms_output.put_line('Hello world');
end;
```

declaring variable: Name [CONSTANT] datatype [NOT NULL] [:= DEFAULT value/expression];

```
v_text varchar(50) NOT NULL DEFAULT 'HELLO';
Begin
v_text := 'PLSQL' || 'Course';
```

→ var_name table_col_name %type;

declare outside Blocks.

Bind variables: (Host variable)

```
variable var_text varchar(30);   (We will use this variable any where)
begin
:var_text := 'Hello PL/SQL';
end;
```

We can print Bind variables using print var_text

set autoprint on (This command will print Bind variables) — another method

Sequence: Select seq_name.nextval/currval into variable/col_name from table_name/dual;

var_name/col_name := seq_name.nextval/currval

records ↖ Collections → multiple rows ← rows

Composite data: (return records)

→ r_name Table_name %rowtype; (only one row) → If we want all columns (to store)

→ type type_name is record (variable_name variable_type, [,] [. . .]);
r_name type_name
↓
(If we want only specific columns)

Eg:
```
declare
  r_emp employees %rowtype;
begin
  select * into r_emp from employees where employee_id = '101';
  dbms_output.put_line(r_emp.first_name);
end;
```
(r_emp ↓ employees)

Collections:
- → Nested Tables → (keys are starting from 1) → unbounded.
- → VArrays → (They are bounded, we specify exact size)
- → Associative arrays → (We can specify the value of keys & keys may be strings too).

↓ (key-value pairs)

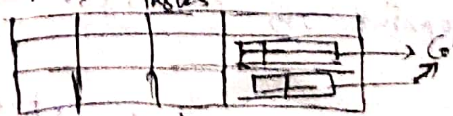
Varray:
```
declare
  type e_list is varray(5) of varchar2(50);
  employees e_list;
begin
  employees := e_list('Alex', 'Bruce', 'John', 'Bob', 'Okay');
  for i in 1..5 loop
    dbms_output.put_line(employees(i));
  end loop;
end;
```

Inbuilt methods: v_array_name.count() → length of varray
varray.first(), varray.last() {index value}, varray.limit() max limit of array
varray.exists(i) → Boolean value,
varray.extend → extend the size of varray

Nested tables:
```
declare
  type e_list is table of varchar2(50);
  emp e_list;
begin
  emp := e_list('Alex', 'Bruce', 'John');
```

get values from DB: declare
emp e_list := e_list()

Associative arrays:



Storing collections in table:

```
create or replace type t_phone_number as object (p_type varchar2(10), p_number varchar2(50));
create or replace type v_phone_numbers as varray(3) of t_phone_number;
create table emps_with_phones (employee_id number, first_name varchar2, last_name varchar2, phone_number v_phone_numbers);
```

Scanned with CamScanner

Exceptions: predefined
→ Non-predefined (Pragma Exception_init (exception_name, error code))
→ RAISE_APPLICATION_ERROR)

Cursors: (When we have small data it is stored in PGA. When we have large data, it is stored in context area. To retrieve the result set faster. Now, at that point, we face some of the ... of the result with the 'cursors'.)

(Cursors are point to the data)

→ We can't do anything on this Because DB server manages this

Fetching & handling data from memory.
→ Implicit cursor. (Created by DB)
→ Explicit cursor. (created by programmers)

→ If we have store large data, at this time we cursors instead of collections because all data doesn't store in memory

Explicit cursors:
1. Declare
2. Open
3. Fetch
4. check.
5. close

```
declare
  cursor cursor_name is select_statement;    -- not into clause
begin
  open cursor_name;
  fetch cursor_name into variables, records etc.
  close cursor_name;
end;
```

→ When cursor fetches a row, the pointer goes to next row of the active set.

Eg:
```
declare
  cursor c_emps is select first_name, last_name from employees;
  v_first_name employees.first_name %type;
  v_last_name employees.last_name %type;
Begin
  open c_emps;
  fetch c_emps into v_first_name, v_last_name;
  dbms_output.put_line ( v_first_name||' '||v_last_name);
  close c_emps;
end;
```

It gives output in 2 rows of first_name lastname

records with cursors:

```
cursor c_emps is Select first_name, last_name from employees
v_emp c_emps %rowtype;

Begin
  open c_emps;
  fetch c_emps into v_emp.first_name, v_emp.last_name;
  dbms_output.put_line (
  close
end;.
```

where current of clause:

Looping with cursors:

```
declare cursor c_emps is select * from employees where dept_id = 30;
v_emps c_emps %rowtype;
begin
   open c_emps;
   loop
      exit when c_emps%notfound;
      fetch c_emps into v_emps;
      output (          )
   end loop;
   close c_emps;
end;
```

for loops:

```
declare
begin
   for i in (c_emps) loop
      output (i.employee_id || i.first_name);
   end loop;
end;
```

we can replace → (Select * from employees where dept_id=30)

Cursors with parameters:

```
declare
   cursor cursor_name (parameter_name datatype, ---)
   is select_statement;
begin
   open cursor_name (parameter_values);
   fetch cursor_name into variables, records etc;
   close cursor_name;
end;
```

Cursor attributes: (like Functions)

- → %_found - returns true if the fetch returned a row
- → %NOTfound - opposite of %found
- → %_ISOPEN - return true if the cursor is open
- → %_ROWCOUNT - Returns the no. of fetched rows up to now.

(fetch)

For update clause of cursors:

```
cursor cursor_name (parameter_name datatype--)
is select_statement
for update [of cols] [nowait | wait n].
```

Functions & procedures:

(returns a value) (doesn't return a value)

→ Create [or replace] procedure procedure_name
((parameter_name [IN|OUT|INOUT] type(,...)]) {IS|AS}
--- → declarative section (No declare keyword)
begin
--
exception
--
end;

Calling a procedure:
Execute procedure_name;
In begin Block:
procedure_name;

Methods for passing parameters: → Positional Notation (fun(a,b,c))
→ Named Notation (fun(a=>a, b=>b))
→ Mixed Notation (fun(a,c=>c))

Creating a functions: Create [OR REPLACE] Function func_name
[(parameter_name [IN|OUT|IN OUT] type[,...])]
return return_datatype {IS|AS}
Begin
<function_Body>
end [function_name];

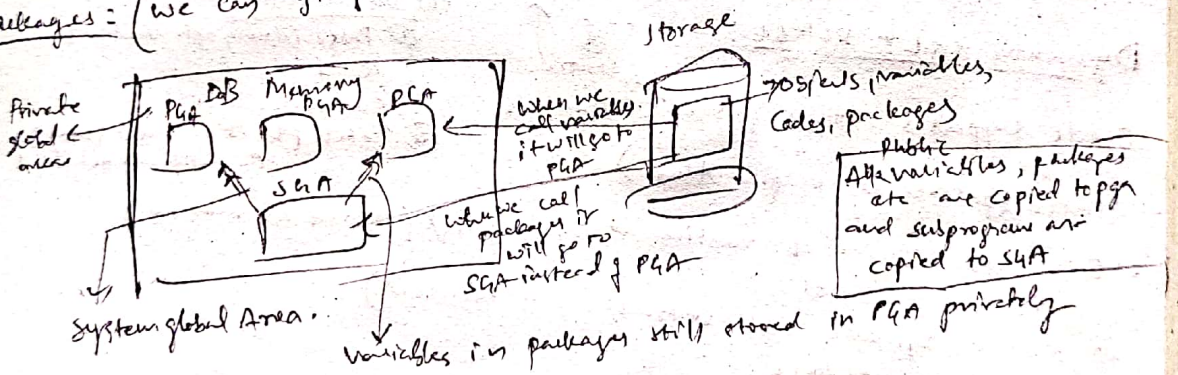
declaration area

Local subprograms: (It doesn't stores in schema, it is in program and deleted automatically after completion)

→ If exception occurs in functions, then in exception block we want to return null, because function returns a value.

Table functions: (Return collection of rows) → Varray → nested tables
↓ → Pipelined (large returns amount of data)
Regular (return small amount of data)

Packages: (we can group variables, functions, procedures, etc types in one container)



→ Packages increases performance & group logically
→ Easy maintance & Security (Private & public) → All are considered as public

Create package: → package specification → (declare)
→ package Body

Create or replace package package_name as
(declarations) →

Create or replace package Body package_name as
(subprograms & the declarations)

How to use packages: package_declarations.
exec package_name.package_functions;

* We can't change the name of package, if we do, it will create another package and old package is also exists.

Visibility of package objects:

-> All variables declared inside spec are public (accessed by anyone)

-> In Body, local variable in subprograms is private (after in/as keyword)

Body -> Accepts forward declarations (we can declare the function after as keyword)

Package initialization: (package initialize block)

Inside Body: begin
V_sal_inc := 500; -> It is variable already declared inside spec.

Persistance state of packages:

If we use PRAGMA SERIALLY_REUSABLE keyword the variables stored in ~~PGA~~ SGA instead of PGA's

grant execute on constant_pkg to my_user; // to give permissions to my_user for accessing the hr package
(package name) (user_name)

Dynamic SQL in PL/SQL:

Oracle supports 2 types of SQL -> Dynamic (preferred) (*) Static

Static (parsed at compile time)

SQL execute steps:
1. Parse (Query optimization)
2. Bind
3. Execute
4. Fetch (If it is select query the fetch will be added)

-> Generate queries at runtime -> Dynamic sql (parsed at runtime)

Generate Dynamic SQL:

-> Native dynamic SQL statements (more efficient)

-> DBMS_SQL package

-> SQL injection is a technique for placing malicious code into your Dynamic SQL statements

Native dynamic SQL: (faster, more powerful and easy to code)

Three ways:
-> Execute Immediate Command
-> open-for, fetch & close statements.

Syntax: execute immediate Dynamic_SQL_string;
[BULK COLLECT] into {variable [, {variable}...] | record}]
[USING [IN|OUT|IN OUT] bind_argument
[, [IN|OUT|IN OUT] bind_argument]...];