PostgreSQL CHEAT SHEET:

- Yash Shirodkar.

I was able to create this cheat sheet with the help of https://www.tutorialspoint.com/postgresql/ , https://www.postgresqltutorial.com/ and other online resources available.

Postgres: Open source, robust, high performance

Create Database:

```
CREATE DATABASE dbname;
```

\l: to list all the databases

\d: to see all tables in the database:

\d tablename: // to describe each table

Connect to a Database :

```
\c testdb
```

```
psql -h localhost -p 5432 -U postgress testdb
```

Drop Database:

```
DROP DATABASE [ IF EXISTS ] name
```

Create Table:

While creating a table make sure to specify the data type of the column and also the primary key:

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

E.g. –

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

Drop Table:

```
DROP TABLE table_name1, table_name2;
```

SELECT:

```
SELECT column1, column2, columnN FROM table_name;
```

INSERT:

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,JOIN_DATE) VALUES
(1, 'Paul', 32, 'California', 20000.00,'2001-07-13');
```

PostgreSQL Logical Operators (Similar to SQL)-
AND, OR, NOT

Conditional Query:

```
testdb=# SELECT COUNT(*) AS "RECORDS" FROM COMPANY;
```

Name start with 'Pa':

```
testdb=# SELECT * FROM COMPANY WHERE NAME LIKE 'Pa%';
```

age either 25 or 27 (is it not a range)

```
testdb=# SELECT * FROM COMPANY WHERE AGE IN (25, 27);
```

for range: use 'between'

```
testdb=# SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
```

AND, OR are always used with WHERE clause:

```
testdb=# SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
```

```
testdb=# SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
```

UPDATE TABLE:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

```
testdb=# UPDATE COMPANY SET SALARY = 15000 WHERE ID = 3;
```

DELETE TABLE:

```
DELETE FROM table_name
WHERE [condition];
```

LIMIT AND OFFSET:

```
testdb=# SELECT * FROM COMPANY LIMIT 3 OFFSET 2;
```

Offset skips the first 2 rows

Common Table Expression (CTE): important, but it is not stored anywhere, so when we run the command which involves CTE_table, we also have to run the CTE. Another thing to notice is that we have to put the select statement directly after the cte.

HAVING Clause:
It is used to apply condition on aggregation, where is used to apply condition on a column.

Correct Order:
SELECT (DISTINCT)
FROM
WHERE
GROUP BY
ORDER BY
HAVING

Declaration of Arrays:

PostgreSQL gives the opportunity to define a column of a table as a variable length multidimensional array. Arrays of any built-in or user-defined base type, enum type, or composite type can be created.

Array type can be declared as

```
CREATE TABLE monthly_savings (
   name text,
   saving_per_quarter integer[],
   scheme text[][]
);
```

or by using the keyword "ARRAY" as

```
CREATE TABLE monthly_savings (
   name text,
   saving_per_quarter integer ARRAY[4],
   scheme text[][]
);
```

Inserting values in array:
Array values can be inserted as a literal constant, enclosing the element values within curly braces and separating them by commas. An example is shown below –

```
INSERT INTO monthly_savings
VALUES ('Manisha',
'{20000, 14600, 23500, 13250}',
'{{"FD", "MF"}, {"FD", "Property"}}');
```

Accessing Arrays:
An example for accessing Arrays is shown below. The command given below will select the persons whose savings are more in second quarter than fourth quarter.

```
SELECT name FROM monhly_savings WHERE saving_per_quarter[2] >
saving_per_quarter[4];
```

Where any saving_per_quarter = 10000

```
SELECT * FROM monthly_savings WHERE 10000 = ANY
(saving_per_quarter);
```

Modifying Arrays:

An example of modifying arrays is as shown below.

```
UPDATE monthly_savings SET saving_per_quarter =
'{25000,25000,27000,27000}'
WHERE name = 'Manisha';
```

CONSTRAINTS:
Constraints are the rules enforced on data columns on table.
The following are commonly used constraints available in PostgreSQL.

- **NOT NULL Constraint** – Ensures that a column cannot have NULL value.
- **UNIQUE Constraint** – Ensures that all values in a column are different.
- **PRIMARY Key** – Uniquely identifies each row/record in a database table.
- **FOREIGN Key** – Constrains data based on columns in other tables.
- **CHECK Constraint** – The CHECK constraint ensures that all values in a column satisfy certain conditions.
- **EXCLUSION Constraint** – The EXCLUDE constraint ensures that if any two rows are compared on the specified column(s) or expression(s) using the specified operator(s), not all of these comparisons will return TRUE.

JOINS (Similar to SQL):

Join Types in PostgreSQL are −

- The CROSS JOIN
- The INNER JOIN
- The LEFT OUTER JOIN / LEFT JOIN
- The RIGHT OUTER JOIN / RIGHT JOIN
- The FULL OUTER JOIN

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_filed = table2.common_field;
```

The PostgreSQL **UNION** (similar to sql) clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order but they do not have to be the same length.

TRUNCATE TABLE:

```
TRUNCATE TABLE table_name;
```

INDEXES (Different and IMP in postgres sql):

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table.

PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST and GIN. Each Index type uses a different algorithm that is best suited to different types of queries. By default, the CREATE INDEX command creates B-tree indexes, which fit the most common situations.

Single-Column Indexes:

A single-column index is one that is created based on only one table column. The basic syntax is as follows −

```
CREATE INDEX index_name
ON table_name (column_name);
```

Multicolumn Indexes:

A multicolumn index is defined on more than one column of a table. The basic syntax is as follows –

```
CREATE INDEX index_name
ON table_name (column1_name, column2_name);
```

Whether to create a single-column index or a multicolumn index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Unique Indexes:

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows –

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Partial Indexes

A partial index is an index built over a subset of a table; the subset is defined by a conditional expression (called the predicate of the partial index). The index contains entries only for those table rows that satisfy the predicate. The basic syntax is as follows –

```
CREATE INDEX index_name
on table_name (conditional_expression);
```

Implicit Indexes:

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

An index helps to speed up SELECT queries and WHERE clauses; however, it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Example

The following is an example where we will create an index on COMPANY table for salary column –

```
# CREATE INDEX salary_index ON COMPANY (salary);
```

Let us list down all the indices available on COMPANY table using **\d company** command.

```
# \d company
```

This will produce the following result, where *company_pkey* is an implicit index, which got created when the table was created.

```
       Table "public.company"
 Column  |     Type      | Modifiers
---------+---------------+-----------
 id      | integer       | not null
 name    | text          | not null
 age     | integer       | not null
 address | character(50) |
 salary  | real          |
Indexes:
    "company_pkey" PRIMARY KEY, btree (id)
    "salary_index" btree (salary)
```

When Should Indexes be Avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered −

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

JSON (Different and IMP in postgres sql):

Create Table with JSON type:

```
CREATE TABLE orders (
        id serial NOT NULL PRIMARY KEY,
        info json NOT NULL
);
```

Insert values into json column:

```
INSERT INTO orders (info)
VALUES('{ "customer": "Lily Bush", "items": {"product": "Diaper","qty": 24}}'),
    ('{ "customer": "Josh William", "items": {"product": "Toy Car","qty": 1}}'),
    ('{ "customer": "Mary Clark", "items": {"product": "Toy Train","qty": 2}}');
```

As you can see a dict type structure is used to insert values, furthermore it can also be a nested dict.

IMP: Accessing Json data
-> and ->> are used to query json data.

The operator -> will return json object. That is, you can further use -> or ->> to retrieve data from the json object (nested dicts).

The operator ->> will return text type. You can't use -> or ->> to retrieve any nested data.

Eg:
```
SELECT info -> 'items' ->> 'product' as product
FROM orders
ORDER BY product;
```

In the above example, "items" is a dict (json), thus you will use -> as you might want to iterate through it further. Product is not a dict, so we would want the result as a text, to print its value.

Where Clause:
```
SELECT info ->> 'customer' AS customer
FROM orders
WHERE info -> 'items' ->> 'product' = 'Diaper';
```

COMPLEX TOPICS THAT COULD BE ASKED IN INTERVIEW:

Arrays in postgres.
Querying Json data.
Indexing in POstgres.
Arrays in JSON. (Advance)

Some SQL and Postgres common Interview Questions-
Window Functions.
Case Statements.
Date Manipulation.
CTE