PostgreSQL Cheatsheet

1. DBML Commands (Database Markup Language)

Command	Description	Example
SELECT	Retrieves data from a database.	`SELECT first_name, last_name FROM customers;`
INSERT	Adds new records to a table.	`INSERT INTO customers (first_name, last_name) VALUES ('Mary', 'Doe');`
UPDATE	Modifies existing records in a table.	`UPDATE employees SET employee_name = 'John Doe', department = 'Marketing';`
DELETE	Removes records from a table.	`DELETE FROM employees WHERE employee_name = 'John Doe';`

2. DDL Commands (Data Definition Language)

Command	Description	Example
CREATE DATABASE	Creates a new database.	`CREATE DATABASE mydatabase;`
CREATE TABLE	Creates a new table.	`CREATE TABLE users (id SERIAL PRIMARY KEY, name TEXT);`
ALTER TABLE	Modifies an existing table.	`ALTER TABLE users ADD COLUMN email TEXT;`
DROP TABLE	Deletes a table.	`DROP TABLE users;`
CREATE INDEX	Creates an index (speeds up searches).	`CREATE INDEX users_name_idx ON users (name);`

3. DCL Commands (Data Control Language)

Command	Description	Example
GRANT	Grants privileges to users.	`GRANT SELECT ON users TO public;`
REVOKE	Revokes privileges from users.	`REVOKE SELECT ON users FROM public;`

4. Query Commands (DQL - Data Query Language)

Command	Description	Example
SELECT	Retrieves data from a table.	`SELECT * FROM users;`
FROM	Specifies the table to retrieve data from.	`SELECT name FROM users;`
WHERE	Filters data based on a condition.	`SELECT * FROM users WHERE id = 1;`
ORDER BY	Sorts the result set.	`SELECT * FROM users ORDER BY name ASC;`
LIMIT	Limits the number of rows returned.	`SELECT * FROM users LIMIT 10;`
OFFSET	Skips a specified number of rows before returning rows.	`SELECT * FROM users OFFSET 5;`
DISTINCT	Returns only distinct (different) values.	`SELECT DISTINCT country FROM customers;`

5. Join Commands

Join Type	Description	Example
INNER JOIN	Returns rows only when there is a match in both tables.	`SELECT * FROM orders INNER JOIN users ON orders.user_id = users.id;`
LEFT JOIN	Returns all rows from the left table and matching rows from the right table.	`SELECT * FROM users LEFT JOIN orders ON users.id = orders.user_id;`
RIGHT JOIN	Returns all rows from the right table and matching rows from the left table.	`SELECT * FROM orders RIGHT JOIN users ON orders.user_id = users.id;`
FULL JOIN	Returns all rows from both tables, regardless of a match.	`SELECT * FROM users FULL JOIN orders ON users.id = orders.user_id;`

6. Subquery

Command	Description	Example
IN	Determines whether a value matches any value in a subquery result.	`SELECT * FROM customers WHERE city IN (SELECT city FROM suppliers);`
ANY	Compares a value to any value returned by a subquery.	`SELECT * FROM products WHERE price < ANY (SELECT unit_price FROM supplier_products);`
ALL	Compares a value to all values returned by a subquery.	`SELECT * FROM orders WHERE order_amount > ALL (SELECT total_amount FROM previous_orders);`

7. Aggregate Functions

Function	Description	Example
COUNT()	Counts the number of rows.	`SELECT COUNT(*) FROM users;`
SUM()	Calculates the sum of a numeric column.	`SELECT SUM(price) FROM orders;`
AVG()	Calculates the average of a numeric column.	`SELECT AVG(age) FROM users;`
MIN()	Finds the minimum value in a column.	`SELECT MIN(salary) FROM employees;`
MAX()	Finds the maximum value in a column.	`SELECT MAX(price) FROM products;`

8. String Commands

Function	Description	Example
LENGTH(str)	Returns the length of a string.	`SELECT LENGTH('Hello');` --> 5
LOWER(str)	Converts a string to lowercase.	`SELECT LOWER('PostgreSQL');` --> postgresql
UPPER(str)	Converts a string to uppercase.	`SELECT UPPER('PostgreSQL');` --> POSTGRESQL
SUBSTRING(str, start, length)	Extracts a substring from a string.	`SELECT SUBSTRING('PostgreSQL', 1, 4);` --> Post

Function	Description	Example
REPLACE(str, from, to)	Replaces all occurrences of a substring.	`SELECT REPLACE('Hello World', 'World', 'PostgreSQL');`
CONCAT(str1, str2, ...)	Concatenates two or more strings.	`SELECT CONCAT('Hello', ' ', 'World');` --> Hello World

9. Date/Time Commands

Function	Description	Example
NOW()	Returns the current date and time.	`SELECT NOW();`
CURRENT_DATE	Returns the current date.	`SELECT CURRENT_DATE;`
CURRENT_TIME	Returns the current time.	`SELECT CURRENT_TIME;`
EXTRACT(field FROM source)	Extracts a specific part (year, month, day) from a date/time value.	`SELECT EXTRACT(YEAR FROM NOW());`
DATE_PART(field, source)	Extracts a specific part (year, month, day) from a date/time value.	`SELECT DATE_PART('year', NOW());`
AGE(timestamp)	Calculates the age (in years) from a timestamp.	`SELECT AGE(birthdate) FROM employees;`

10. Conditions

Operator	Description
=	Equal to
<>	Not equal to
!=	Not equal to (alternative)
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
BETWEEN	Between a range of values
LIKE	Matches a pattern

Operator	Description
IN	In a list of values
IS NULL	Checks for null values
IS NOT NULL	Checks for non-null values

Conditional Expressions

Command	Description	Example
CASE	The CASE statement allows you to perform conditional logic within a query.	`SELECT order_id, total_amount, CASE WHEN total_amount > 1000 THEN 'High Value Order' WHEN total_amount > 500 THEN 'Medium Value Order' ELSE 'Low Value Order' END AS order_status FROM orders;`
COALESCE	The COALESCE() function returns the first non-null value from a list of values.	`SELECT COALESCE(first_name, middle_name) AS preferred_name FROM employees;`
NULLIF	The NULLIF() function returns null if two specified expressions are equal	`SELECT NULLIF(total_amount, discounted_amount) AS diff_amount FROM orders;`

11. Set Operations

Operator	Description	Example
UNION	Combines the results of two or more queries.	`SELECT first_name, last_name FROM customers UNION SELECT first_name, last_name FROM employees;`
INTERSECT	Returns rows common to both queries.	`SELECT first_name, last_name FROM customers INTERSECT SELECT first_name, last_name FROM employees;`
EXCEPT	Returns rows in the first query but not the second.	`SELECT first_name, last_name FROM customers EXCEPT SELECT first_name, last_name FROM employees;`

12. Indexes

Purpose: Improve the speed of data retrieval operations (SELECT queries) on tables.

How they work: Indexes create a separate data structure that stores a subset of table data in a sorted order or a hash structure. This allows the database to locate matching rows much faster than scanning the entire table.

Syntax:

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column_name [ASC | DESC], ...);
```

Types of Indexes:

Index Type	Description
B-tree (Default)	Suitable for equality and range queries, as well as sorting. Most common index type.
Hash	Very fast for equality comparisons, but not suitable for range queries or sorting.
GIN (Generalized Inverted Index)	Efficient for searching arrays and other composite data types. Useful for full-text search.
GiST (Generalized Search Tree)	Used for geometric and spatial data types, as well as full-text search. Provides flexible indexing.
SP-GiST (Space-Partitioned GiST)	Variant of GiST optimized for multidimensional data.
BRIN (Block Range Index)	Designed for very large tables, storing index data at a block level to save space.

Choosing the Right Index:

- **Equality Queries:** Use B-tree or Hash indexes on frequently queried columns.
- **Range Queries:** B-tree is the best choice.
- **Sorting:** B-tree indexes can speed up sorting on indexed columns.
- **Full-Text Search:** Use GIN or GiST indexes depending on your specific requirements.
- **Multi-Column Indexes (Composite Indexes):** Create indexes on multiple columns to optimize complex queries. The order of columns in the index definition matters!

13. Transaction

Command	Description	Example
COMMIT	Commits the current transaction.	BEGIN TRANSACTION; -- SQL statements and changes within the transaction INSERT INTO employees (name, age) VALUES ('Alice', 30); UPDATE products SET price = 25.00 WHERE category = 'Electronics'; COMMIT;
ROLLBACK	Reverts the current transaction.	BEGIN TRANSACTION; -- SQL statements and changes within the transaction INSERT INTO employees (name, age) VALUES ('Bob', 35); UPDATE products SET price = 30.00 WHERE category = 'Electronics'; ROLLBACK;

14. Window Function

- Performs a calculation across a set of table rows.
- Example: `SELECT name, salary, AVG(salary) OVER () AS avg_salary FROM employees;`

```
function_name(arguments) OVER ( [PARTITION BY column] [ORDER BY column] [frame_clause] )
```

Components:

- **`function_name`:** Aggregate function (SUM, AVG, COUNT, etc.), ranking function (RANK, DENSE_RANK, ROW_NUMBER), or others.
- **`arguments`:** Columns or expressions used by the function.
- **`OVER clause`:** Defines the window:
  - **`PARTITION BY column`:** (Optional) Divides rows into groups (partitions).
  - **`ORDER BY column`:** (Optional) Specifies the order within each partition.
  - **`frame_clause`:** (Optional) Further defines the window using keywords like `ROWS`, `RANGE`, `PRECEDING`, `FOLLOWING`.

Common Window Functions:

Function	Description
`ROW_NUMBER()`	Assigns a unique sequential number to each row within its partition.
`RANK()`	Assigns a rank to each row, with ties receiving the same rank.
`DENSE_RANK()`	Similar to `RANK()`, but ties don't create gaps in rank values.
`SUM(column)`	Calculates the sum of `column` over the window.
`AVG(column)`	Calculates the average of `column` over the window.
`COUNT(column)`	Counts the number of non-null values in `column` over the window.
`FIRST_VALUE(column)`	Returns the first value of `column` in the window.
`LAST_VALUE(column)`	Returns the last value of `column` in the window.
`LAG(column, offset)`	Accesses data from a previous row within the partition.
`LEAD(column, offset)`	Accesses data from a following row within the partition.

Example:

```
SELECT
    department,
    employee,
    salary,
    AVG(salary) OVER (PARTITION BY department) AS avg_salary_dept
FROM employees;
```

This query calculates the average salary for each department using a window function and displays it alongside each employee's information.

15. WITH Keyword (Common Table Expressions - CTE)

- Defines a temporary named result set.
- Example:

```
WITH top_products AS (
  SELECT product_id, SUM(quantity) AS total_sold
```

```
  FROM orders
  GROUP BY product_id
  ORDER BY total_sold DESC
  LIMIT 10
)
SELECT * FROM top_products;
```

16. View

- A virtual table based on a stored query.
- Created with `CREATE VIEW view_name AS SELECT ...`.

```
CREATE VIEW order_summary AS
SELECT
    c.name AS customer_name,
    COUNT(o.order_id) AS order_count,
    SUM(o.total_amount) AS total_spent
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name;
```

17. Materialized View

- A pre-computed view that stores data physically.
- Created with `CREATE MATERIALIZED VIEW view_name AS SELECT ...`.

```
CREATE MATERIALIZED VIEW order_summary AS
SELECT
    c.name AS customer_name,
    COUNT(o.order_id) AS order_count,
    SUM(o.total_amount) AS total_spent
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name;
```

18. Query Analysis (Explain Plan)

- Use `EXPLAIN` or `EXPLAIN ANALYZE` to understand the query plan.

```
EXPLAIN ANALYZE
SELECT c.name, SUM(o.total_amount)
```

```
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE c.city = 'New York'
GROUP BY c.customer_id, c.name;
```

19. Partition

Purpose: Improve performance for large tables by dividing them into logical parts.

- Divides a table into logical parts for performance.
- Created with `CREATE TABLE ... PARTITION BY ...`.

Types:

- **Range Partitioning:** Partitions are based on a range of values in a specified column (e.g., dates, numbers).
- **List Partitioning:** Partitions contain a specific list of values for a column.
- **Hash Partitioning:** Partitions are determined by a hash function applied to a column.

Creating Partitioned Tables:

1. **Create the Master Table (Partitioned Table):**

```
CREATE TABLE table_name (
    -- Column definitions
) PARTITION BY RANGE (partition_column);
```

2. **Create Partitions:**

```
CREATE TABLE partition_name PARTITION OF table_name
FOR VALUES FROM (start_value) TO (end_value);
```

Example: Range Partitioning (Orders by Year)

```
-- Create the master table partitioned by order date
CREATE TABLE orders_partitioned (
    order_id SERIAL PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10, 2)
) PARTITION BY RANGE (order_date);
```

```
-- Create partitions for specific years
CREATE TABLE orders_2023 PARTITION OF orders_partitioned
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE orders_2024 PARTITION OF orders_partitioned
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
```

Benefits:

- **Improved Query Performance:** Queries targeting specific partitions can be processed much faster.
- **Faster Data Loads and Deletes:** Operations on individual partitions are generally more efficient.
- **Easier Maintenance:** Archiving or deleting old data is simplified by dropping or truncating partitions.

Important Considerations:

- Choose appropriate partition keys for your queries.
- Ensure that data is distributed evenly across partitions to avoid "partition skew."
- Test queries with `EXPLAIN` to verify that partitions are being used effectively.

20. Trigger

Purpose: Define actions that automatically execute before or after specific database events (e.g., INSERT, UPDATE, DELETE).

- A stored procedure that automatically executes when an event occurs.
- Created with `CREATE TRIGGER trigger_name ... ON table_name ...`.

Types:

- **Statement-Level Triggers:** Execute once per SQL statement, regardless of the number of rows affected.
- **Row-Level Triggers:** Execute once for each row affected by the statement.

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE } [OR ...]
ON table_name
[ FOR EACH { ROW | STATEMENT } ]
```

```
[ WHEN (condition) ]
EXECUTE FUNCTION function_name();
```

Example: Updating Order Total using a Trigger

```
-- 1. Table for order items
CREATE TABLE order_items (
    order_id INT REFERENCES orders(order_id),
    product_id INT REFERENCES products(product_id),
    quantity INT,
    price DECIMAL(10, 2)
);

-- 2. Function to calculate and update the order total
CREATE OR REPLACE FUNCTION update_order_total()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE orders
    SET total_amount = (
        SELECT SUM(quantity * price)
        FROM order_items
        WHERE order_id = NEW.order_id
    )
    WHERE order_id = NEW.order_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- 3. Trigger to call the function after changes to order_items
CREATE OR REPLACE TRIGGER update_order_total_trigger
AFTER INSERT OR UPDATE OR DELETE ON order_items
FOR EACH ROW
EXECUTE FUNCTION update_order_total();
```

Explanation:

- `update_order_total()` **Function:**
  - Calculates the total amount for the given `order_id` from `order_items`.
  - Updates the `total_amount` in the `orders` table.
- `update_order_total_trigger` **Trigger:**
  - Executes **after** any `INSERT`, `UPDATE`, or `DELETE` operation on the `order_items` table.
  - It's a **row-level trigger** (`FOR EACH ROW`), so it runs for every row changed.

- It calls the `update_order_total()` function to recalculate and update the order's total amount.

Generate Data in Series

```
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    city VARCHAR(50),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
INSERT INTO customers (first_name, last_name, email, city)
SELECT
    'John' || i, 'Doe' || i,
    'john.doe' || i || '@example.com',
    'City' || (i % 100),
FROM generate_series(1, 10000) AS s(i);
```

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    employee_name VARCHAR(100) NOT NULL,
    department VARCHAR(50),
    hire_date DATE,
    salary NUMERIC(10, 2),
    is_active BOOLEAN DEFAULT TRUE
);
```

```
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    category VARCHAR(50),
    unit_price NUMERIC(10, 2) NOT NULL,
    stock INT CHECK (stock >= 0)
);
```

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
```

```
    customer_id INT REFERENCES customers(customer_id),
    order_date DATE DEFAULT CURRENT_DATE,
    total_amount NUMERIC(10, 2) NOT NULL,
    status VARCHAR(20) CHECK (status IN ('Pending', 'Shipped',
'Delivered', 'Cancelled'))
);
```

```
INSERT INTO orders (customer_id, order_date, total_amount, status)
SELECT
    (random() * 9999 + 1)::int AS customer_id,
    CURRENT_DATE - (random() * 365)::int AS order_date,
    round((random() * 980 + 20), 2) AS total_amount,
    CASE
        WHEN random() < 0.25 THEN 'Pending'
        WHEN random() < 0.5 THEN 'Shipped'
        WHEN random() < 0.75 THEN 'Delivered'
        ELSE 'Cancelled'
    END AS status
FROM generate_series(1, 10000);
```

Latest Partition Code

```
-- master table

CREATE TABLE orders_partitioned (

order_id SERIAL,

customer_id INT REFERENCES customers(customer_id),

order_date DATE,

total_amount NUMERIC(10, 2),

status VARCHAR(20) CHECK (status IN ('Pending', 'Shipped',
'Delivered', 'Cancelled')),

PRIMARY KEY (order_id, order_date)

) PARTITION BY RANGE (order_date);
```

```
-- Create partitions for specific years


CREATE TABLE orders_2023 PARTITION OF orders_partitioned

FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');


CREATE TABLE orders_2024 PARTITION OF orders_partitioned

FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');



-- Insert Data

INSERT INTO orders_partitioned (order_id, customer_id, order_date,
total_amount, status)

SELECT order_id, customer_id, order_date, total_amount, status

FROM orders;



-- query data

select * from orders_partitioned;


explain select * from orders_partitioned;


select * from orders_partitioned where order_date BETWEEN '2024-01-
01' AND '2024-12-31';
```

```
explain select * from orders_partitioned where order_date BETWEEN
'2024-01-01' AND '2024-12-31';
```

```
-- 1. Table for order items

CREATE TABLE products (

product_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

category VARCHAR(50),

unit_price NUMERIC(10, 2) NOT NULL,

stock INT CHECK (stock >= 0)

);



CREATE TABLE order_items (

order_id INT REFERENCES orders(order_id),

product_id INT REFERENCES products(product_id),

quantity INT,

price DECIMAL(10, 2)

);



-- 2. Function to calculate and update the order total

CREATE OR REPLACE FUNCTION update_order_total()

RETURNS TRIGGER AS $$

BEGIN

UPDATE orders
```

```
SET total_amount = (
SELECT SUM(quantity * price)
FROM order_items
WHERE order_id = NEW.order_id
)
WHERE order_id = NEW.order_id;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;



-- 3. Trigger to call the function after changes to order_items
CREATE OR REPLACE TRIGGER update_order_total_trigger
AFTER INSERT OR UPDATE OR DELETE ON order_items
FOR EACH ROW
EXECUTE FUNCTION update_order_total();



-- Add data
INSERT INTO products (name, category, unit_price, stock)
VALUES
('Laptop', 'Electronics', 1200.00, 50),
('Phone', 'Electronics', 800.00, 100),
('Book', 'Stationery', 20.00, 200);
```

```sql
select * from products;


INSERT INTO orders (customer_id, order_date, total_amount, status)
VALUES (1, '2024-11-13', 0.00, 'Pending') RETURNING order_id;


INSERT INTO order_items (order_id, product_id, quantity, price)
VALUES
(1, 1, 2, 1200.00), -- 2 Laptops
(1, 2, 1, 800.00); -- 1 Phone


-- verify
SELECT * FROM orders WHERE order_id = 1;


UPDATE order_items
SET quantity = 3
WHERE order_id = 1 AND product_id = 1;



DELETE FROM order_items
WHERE order_id = 1 AND product_id = 2;
```