# PostgreSQL

## Table of Contents

# Data Types

## Numeric Types

```
-- Integers
SMALLINT             -- 2 bytes, -32768 to 32767
INTEGER              -- 4 bytes, -2147483648 to 2147483647
BIGINT               -- 8 bytes, -9223372036854775808 to
9223372036854775807
SERIAL               -- Auto-incrementing INTEGER
BIGSERIAL            -- Auto-incrementing BIGINT

-- Decimal/Floating Point
DECIMAL(p,s)         -- Exact, p=precision, s=scale
NUMERIC(p,s)         -- Exact, p=precision, s=scale
```

```
REAL                    -- 4 bytes, inexact, 6 decimal digits precision
DOUBLE PRECISION        -- 8 bytes, inexact, 15 decimal digits precision
```

### Character Types

```
CHAR(n)                 -- Fixed-length string, blank padded
VARCHAR(n)              -- Variable-length string with limit
TEXT                    -- Variable unlimited length
```

### Boolean Type

```
BOOLEAN                 -- true/false
```

### Date/Time Types

```
DATE                    -- Date only (YYYY-MM-DD)
TIME                    -- Time only (HH:MM:SS)
TIMESTAMP               -- Date and time (YYYY-MM-DD HH:MM:SS)
TIMESTAMPTZ             -- Date and time with timezone
INTERVAL                -- Time interval
```

### Other Types

```
BYTEA                   -- Binary data
JSON                    -- JSON data
JSONB                   -- Binary JSON data (more efficient)
UUID                    -- Universal unique identifier
ARRAY                   -- Array of values
ENUM                    -- User-defined enumerated type
CIDR, INET, MACADDR     -- Network address types
POINT, LINE, POLYGON    -- Geometric types
TSQUERY, TSVECTOR       -- Text search types
```

## Database Operations

### Create Database

```
CREATE DATABASE database_name;
CREATE DATABASE database_name OWNER username ENCODING 'UTF8';
```

### Drop Database

```
DROP DATABASE [IF EXISTS] database_name;
```

### List Databases

```
\l                    -- psql command
SELECT datname FROM pg_database;
```

## Table Operations

### Create Table

```
CREATE TABLE table_name (
    column1 datatype [constraints],
    column2 datatype [constraints],
    ...
);

-- Example
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    hire_date DATE DEFAULT CURRENT_DATE,
    salary NUMERIC(10,2)
);
```

### Alter Table

```
-- Add column
ALTER TABLE table_name ADD COLUMN column_name datatype [constraints];

-- Drop column
ALTER TABLE table_name DROP COLUMN column_name;

-- Rename column
ALTER TABLE table_name RENAME COLUMN old_column TO new_column;

-- Change column data type
ALTER TABLE table_name ALTER COLUMN column_name TYPE new_datatype;

-- Add constraint
ALTER TABLE table_name ADD CONSTRAINT constraint_name
constraint_definition;
```

```
-- Drop constraint
ALTER TABLE table_name DROP CONSTRAINT constraint_name;

-- Rename table
ALTER TABLE table_name RENAME TO new_table_name;
```

### Drop Table

```
DROP TABLE [IF EXISTS] table_name [CASCADE | RESTRICT];
```

### Truncate Table (Delete all rows)

```
TRUNCATE TABLE table_name [CASCADE | RESTRICT];
```

### Copy Table Structure (without data)

```
CREATE TABLE new_table AS TABLE existing_table WITH NO DATA;
```

### Copy Table with Data

```
CREATE TABLE new_table AS SELECT * FROM existing_table;
```

## Constraints

### Primary Key

```
-- During table creation
CREATE TABLE table_name (
    id SERIAL PRIMARY KEY,
    ...
);

-- Alter existing table
ALTER TABLE table_name ADD PRIMARY KEY (column_name);
```

### Foreign Key

```
-- During table creation
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INTEGER REFERENCES customers(customer_id),
    ...
);
```

```sql
-- Alter existing table
ALTER TABLE orders ADD CONSTRAINT fk_customer
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
    ON DELETE CASCADE;
```

### Unique Constraint

```sql
-- During table creation
CREATE TABLE users (
    username VARCHAR(50) UNIQUE,
    ...
);

-- Alter existing table
ALTER TABLE users ADD CONSTRAINT unique_email UNIQUE (email);
```

### Check Constraint

```sql
-- During table creation
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    price NUMERIC(10,2) CHECK (price > 0),
    ...
);

-- Alter existing table
ALTER TABLE products ADD CONSTRAINT positive_quantity
    CHECK (quantity >= 0);
```

### Not Null Constraint

```sql
-- During table creation
CREATE TABLE contacts (
    first_name VARCHAR(50) NOT NULL,
    ...
);

-- Alter existing table
ALTER TABLE contacts ALTER COLUMN last_name SET NOT NULL;
```

### Default Value

```sql
-- During table creation
CREATE TABLE orders (
    order_date DATE DEFAULT CURRENT_DATE,
    ...
);

-- Alter existing table
ALTER TABLE orders ALTER COLUMN status SET DEFAULT 'Pending';
```

## Basic Queries

### Insert Data

```
-- Insert single row
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);

-- Insert multiple rows
INSERT INTO table_name (column1, column2, ...)
VALUES
    (value1, value2, ...),
    (value1, value2, ...);

-- Insert with returning
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...)
RETURNING column1, column2;

-- Insert from select
INSERT INTO table_name (column1, column2, ...)
SELECT column1, column2, ... FROM other_table
WHERE condition;
```

### Update Data

```
-- Update all rows
UPDATE table_name
SET column1 = value1, column2 = value2
[WHERE condition];

-- Update with join
UPDATE table_name t1
SET column1 = t2.column1
FROM table_name2 t2
WHERE t1.id = t2.id;

-- Update with returning
UPDATE table_name
SET column1 = value1
```

```sql
WHERE condition
RETURNING column1, column2;
```

## Delete Data

```sql
-- Delete rows
DELETE FROM table_name
[WHERE condition];

-- Delete with returning
DELETE FROM table_name
WHERE condition
RETURNING column1, column2;

-- Delete using joins
DELETE FROM table_name
USING another_table
WHERE table_name.id = another_table.id AND condition;
```

## Select Data

```sql
-- Select all columns
SELECT * FROM table_name;

-- Select specific columns
SELECT column1, column2 FROM table_name;

-- Select with column aliases
SELECT
    column1 AS alias1,
    column2 AS "Alias With Space"
FROM table_name;

-- Select with expression
SELECT
    first_name,
    last_name,
    first_name || ' ' || last_name AS full_name
FROM employees;

-- Select distinct values
SELECT DISTINCT column1 FROM table_name;

-- Select distinct combinations
SELECT DISTINCT ON (column1) column1, column2
FROM table_name
ORDER BY column1, column2;
```

```sql
-- Limit and offset
SELECT * FROM table_name
LIMIT 10 OFFSET 20;

-- Selecting with calculated fields
SELECT
    product_name,
    unit_price * units_in_stock AS inventory_value
FROM products;
```

## Filtering Data

### Basic WHERE Clauses

```sql
-- Comparison operators
SELECT * FROM table_name WHERE column_name = value;
SELECT * FROM table_name WHERE column_name > value;
SELECT * FROM table_name WHERE column_name >= value;
SELECT * FROM table_name WHERE column_name < value;
SELECT * FROM table_name WHERE column_name <= value;
SELECT * FROM table_name WHERE column_name <> value; -- Not equal
SELECT * FROM table_name WHERE column_name != value; -- Not equal

-- BETWEEN
SELECT * FROM table_name
WHERE column_name BETWEEN value1 AND value2;

-- IN operator
SELECT * FROM table_name
WHERE column_name IN (value1, value2, ...);

-- NOT IN
SELECT * FROM table_name
WHERE column_name NOT IN (value1, value2, ...);

-- LIKE (pattern matching)
SELECT * FROM table_name
WHERE column_name LIKE 'pattern%';  -- Starts with 'pattern'
SELECT * FROM table_name
WHERE column_name LIKE '%pattern';  -- Ends with 'pattern'
SELECT * FROM table_name
WHERE column_name LIKE '%pattern%'; -- Contains 'pattern'
SELECT * FROM table_name
WHERE column_name LIKE '_pattern';  -- Second to last chars are 'pattern'

-- ILIKE (case-insensitive pattern matching)
SELECT * FROM table_name
```

```sql
WHERE column_name ILIKE '%pattern%';

-- IS NULL / IS NOT NULL
SELECT * FROM table_name
WHERE column_name IS NULL;
SELECT * FROM table_name
WHERE column_name IS NOT NULL;

-- Logical operators
SELECT * FROM table_name
WHERE condition1 AND condition2;
SELECT * FROM table_name
WHERE condition1 OR condition2;
SELECT * FROM table_name
WHERE NOT condition;
```

## Advanced Filtering

```sql
-- Regex matching
SELECT * FROM table_name
WHERE column_name ~ 'regex_pattern';

-- Case-insensitive regex
SELECT * FROM table_name
WHERE column_name ~* 'regex_pattern';

-- SIMILAR TO (SQL regex)
SELECT * FROM table_name
WHERE column_name SIMILAR TO 'pattern';

-- Filtering with arrays
SELECT * FROM table_name
WHERE column_name = ANY(ARRAY[value1, value2]);
SELECT * FROM table_name
WHERE column_name @> ARRAY[value1, value2];

-- Filtering with JSON
SELECT * FROM table_name
WHERE json_column->>'property' = 'value';
```

## Sorting Data

```sql
-- Basic sorting
SELECT * FROM table_name
ORDER BY column1 [ASC | DESC];

-- Sorting by multiple columns
```

```sql
SELECT * FROM table_name
ORDER BY column1 ASC, column2 DESC;

-- Sorting with NULLS positioning
SELECT * FROM table_name
ORDER BY column1 NULLS FIRST;
SELECT * FROM table_name
ORDER BY column1 NULLS LAST;

-- Sorting with expressions
SELECT * FROM table_name
ORDER BY LOWER(column1);
SELECT * FROM table_name
ORDER BY column1 + column2;

-- Sorting by column position
SELECT column1, column2, column3 FROM table_name
ORDER BY 2 DESC;  -- Sorts by column2
```

## Joins

### Inner Join

```sql
SELECT a.column1, b.column2
FROM table_a a
INNER JOIN table_b b ON a.key = b.key;

-- Inner join with additional conditions
SELECT a.column1, b.column2
FROM table_a a
INNER JOIN table_b b
    ON a.key = b.key AND b.column3 = 'value';
```

### Left Join (Left Outer Join)

```sql
SELECT a.column1, b.column2
FROM table_a a
LEFT JOIN table_b b ON a.key = b.key;
```

### Right Join (Right Outer Join)

```sql
SELECT a.column1, b.column2
FROM table_a a
RIGHT JOIN table_b b ON a.key = b.key;
```

## Full Join (Full Outer Join)

```sql
SELECT a.column1, b.column2
FROM table_a a
FULL JOIN table_b b ON a.key = b.key;
```

## Cross Join (Cartesian Product)

```sql
SELECT a.column1, b.column2
FROM table_a a
CROSS JOIN table_b b;
```

## Self Join

```sql
SELECT e.name AS employee, m.name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id;
```

## Natural Join

```sql
-- Joins tables using columns with the same name
SELECT a.column1, b.column2
FROM table_a a
NATURAL JOIN table_b b;
```

## Join with USING

```sql
-- When join columns have the same name
SELECT a.column1, b.column2
FROM table_a a
JOIN table_b b USING (common_column);
```

## Anti-Join (Finding rows in one table but not in another)

```sql
-- Using NOT EXISTS
SELECT a.*
FROM table_a a
WHERE NOT EXISTS (
    SELECT 1 FROM table_b b
    WHERE a.key = b.key
);

-- Using LEFT JOIN / IS NULL
```

```sql
SELECT a.*
FROM table_a a
LEFT JOIN table_b b ON a.key = b.key
WHERE b.key IS NULL;
```

## Aggregations

### Basic Aggregate Functions

```sql
SELECT
    COUNT(*) AS total_rows,
    COUNT(column1) AS non_null_values,
    COUNT(DISTINCT column1) AS unique_values,
    SUM(column1) AS total,
    AVG(column1) AS average,
    MIN(column1) AS minimum,
    MAX(column1) AS maximum
FROM table_name;
```

### Other Aggregate Functions

```sql
SELECT
    ARRAY_AGG(column1) AS all_values_array,
    STRING_AGG(column1, ',') AS concatenated_values,
    JSONB_AGG(column1) AS json_array,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY column1) AS median,
    STDDEV(column1) AS standard_deviation,
    VARIANCE(column1) AS variance,
    MODE() WITHIN GROUP (ORDER BY column1) AS most_common_value
FROM table_name;
```

### Conditional Aggregation

```sql
SELECT
    SUM(CASE WHEN condition THEN column1 ELSE 0 END) AS conditional_sum,
    COUNT(CASE WHEN condition THEN 1 END) AS count_if_true,
    AVG(CASE WHEN condition THEN column1 END) AS avg_if_true
FROM table_name;
```

### FILTER Clause (PostgreSQL specific)

```sql
SELECT
    COUNT(*) AS total,
    COUNT(*) FILTER (WHERE condition) AS filtered_count,
    SUM(column1) FILTER (WHERE condition) AS filtered_sum
FROM table_name;
```

## Grouping

### Basic GROUP BY

```
SELECT column1, COUNT(*) AS count
FROM table_name
GROUP BY column1;
```

### GROUP BY Multiple Columns

```
SELECT column1, column2, SUM(column3) AS total
FROM table_name
GROUP BY column1, column2;
```

### HAVING Clause (Filtering after grouping)

```
SELECT column1, COUNT(*) AS count
FROM table_name
GROUP BY column1
HAVING COUNT(*) > 5;
```

### GROUPING SETS

```
SELECT column1, column2, SUM(column3)
FROM table_name
GROUP BY GROUPING SETS (
    (column1, column2),
    (column1),
    (column2),
    ()
);
```

### ROLLUP (Hierarchical grouping)

```
SELECT column1, column2, SUM(column3)
FROM table_name
GROUP BY ROLLUP(column1, column2);
```

### CUBE (All possible grouping combinations)

```sql
SELECT column1, column2, SUM(column3)
FROM table_name
GROUP BY CUBE(column1, column2);
```

## Subqueries

### Scalar Subquery (returns a single value)

```sql
SELECT column1,
       (SELECT AVG(column1) FROM table_name) AS avg_value
FROM table_name
WHERE column1 > (SELECT AVG(column1) FROM table_name);
```

### Row Subquery (returns a single row)

```sql
SELECT *
FROM table_name
WHERE (column1, column2) = (
    SELECT column1, column2
    FROM another_table
    WHERE condition
    LIMIT 1
);
```

### Column Subquery (returns a single column)

```sql
SELECT *
FROM table_name
WHERE column1 IN (
    SELECT column1
    FROM another_table
    WHERE condition
);
```

### Table Subquery (returns multiple rows and columns)

```sql
SELECT a.column1, b.column2
FROM table_name a
JOIN (
    SELECT column1, column2
    FROM another_table
    WHERE condition
) b ON a.column1 = b.column1;
```

### EXISTS Subquery (returns boolean result)

```
SELECT *
FROM table_name a
WHERE EXISTS (
    SELECT 1
    FROM another_table b
    WHERE b.column1 = a.column1
);
```

### ANY/SOME and ALL Subqueries

```
SELECT *
FROM table_name
WHERE column1 > ANY (
    SELECT column1
    FROM another_table
    WHERE condition
);

SELECT *
FROM table_name
WHERE column1 > ALL (
    SELECT column1
    FROM another_table
    WHERE condition
);
```

### Correlated Subquery (references outer query)

```
SELECT a.column1,
       (SELECT COUNT(*)
        FROM another_table b
        WHERE b.parent_id = a.id) AS child_count
FROM table_name a;
```

## Common Table Expressions (CTEs)

### Basic CTE

```
WITH cte_name AS (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
```

```sql
SELECT *
FROM cte_name
WHERE column1 > value;
```

## Multiple CTEs

```sql
WITH cte1 AS (
    SELECT column1, column2
    FROM table1
    WHERE condition1
),
cte2 AS (
    SELECT column1, column3
    FROM table2
    WHERE condition2
)
SELECT cte1.column1, cte1.column2, cte2.column3
FROM cte1
JOIN cte2 ON cte1.column1 = cte2.column1;
```

## Recursive CTE

```sql
WITH RECURSIVE cte_name AS (
    -- Non-recursive term (anchor)
    SELECT column1, column2
    FROM table_name
    WHERE condition

    UNION ALL

    -- Recursive term
    SELECT t.column1, c.column2
    FROM table_name t
    JOIN cte_name c ON t.parent_id = c.id
)
SELECT * FROM cte_name;

-- Example: Generate series of dates
WITH RECURSIVE date_series AS (
    SELECT '2023-01-01'::date AS date

    UNION ALL

    SELECT date + 1
    FROM date_series
    WHERE date < '2023-01-31'
)
SELECT * FROM date_series;
```

### Materialized CTE

```
-- Only in PostgreSQL 12+
WITH cte_name AS MATERIALIZED (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT * FROM cte_name;
```

## Window Functions

### Basic Window Function Syntax

```
SELECT
    column1,
    column2,
    window_function() OVER (
        [PARTITION BY partition_expression]
        [ORDER BY sort_expression [ASC | DESC] [NULLS FIRST | NULLS LAST]]
        [frame_clause]
    ) AS alias
FROM table_name;
```

### Ranking Functions

```
SELECT
    column1,
    ROW_NUMBER() OVER (PARTITION BY column2 ORDER BY column3) AS row_num,
    RANK() OVER (PARTITION BY column2 ORDER BY column3) AS rank,
    DENSE_RANK() OVER (PARTITION BY column2 ORDER BY column3) AS
dense_rank,
    PERCENT_RANK() OVER (PARTITION BY column2 ORDER BY column3) AS
percent_rank,
    CUME_DIST() OVER (PARTITION BY column2 ORDER BY column3) AS cume_dist,
    NTILE(4) OVER (PARTITION BY column2 ORDER BY column3) AS quartile
FROM table_name;
```

### Aggregate Window Functions

```
SELECT
    column1,
    column2,
    SUM(column3) OVER (PARTITION BY column1) AS total_by_column1,
    AVG(column3) OVER (PARTITION BY column1) AS avg_by_column1,
    COUNT(*) OVER (PARTITION BY column1) AS count_by_column1,
    MAX(column3) OVER (PARTITION BY column1) AS max_by_column1,
    MIN(column3) OVER (PARTITION BY column1) AS min_by_column1
FROM table_name;
```

## Value Window Functions

```
SELECT
    column1,
    column2,
    FIRST_VALUE(column2) OVER (PARTITION BY column1 ORDER BY column3) AS
first_value,
    LAST_VALUE(column2) OVER (
        PARTITION BY column1
        ORDER BY column3
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS last_value,
    LEAD(column2) OVER (PARTITION BY column1 ORDER BY column3) AS
next_value,
    LEAD(column2, 2, 'N/A') OVER (PARTITION BY column1 ORDER BY column3)
AS second_next_value,
    LAG(column2) OVER (PARTITION BY column1 ORDER BY column3) AS
previous_value,
    LAG(column2, 2, 'N/A') OVER (PARTITION BY column1 ORDER BY column3) AS
second_previous_value
FROM table_name;
```

## Window Frames

```
SELECT
    column1,
    column2,
    SUM(column3) OVER (
        PARTITION BY column1
        ORDER BY column2
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS moving_sum,
    AVG(column3) OVER (
        PARTITION BY column1
        ORDER BY column2
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_avg
FROM table_name;
```

## Indexes

### Create Index

```
-- Basic index
CREATE INDEX index_name ON table_name (column_name);

-- Multi-column index
CREATE INDEX index_name ON table_name (column1, column2);

-- Unique index
CREATE UNIQUE INDEX index_name ON table_name (column_name);

-- Partial index
CREATE INDEX index_name ON table_name (column_name)
WHERE condition;

-- Expression index
CREATE INDEX index_name ON table_name (LOWER(column_name));

-- GIN index (for arrays, jsonb, etc.)
CREATE INDEX index_name ON table_name USING GIN (column_name);

-- GIST index (for geometric data types, etc.)
CREATE INDEX index_name ON table_name USING GIST (column_name);

-- BRIN index (for large tables with sorted values)
CREATE INDEX index_name ON table_name USING BRIN (column_name);

-- Hash index (equality operations only)
CREATE INDEX index_name ON table_name USING HASH (column_name);
```

### Alter Index

```
-- Rename index
ALTER INDEX index_name RENAME TO new_index_name;

-- Set tablespace
ALTER INDEX index_name SET TABLESPACE tablespace_name;
```

### Drop Index

```
DROP INDEX [IF EXISTS] index_name [CASCADE | RESTRICT];
```

### Reindex

```sql
-- Rebuild single index
REINDEX INDEX index_name;

-- Rebuild all indexes on a table
REINDEX TABLE table_name;

-- Rebuild all indexes in a database
REINDEX DATABASE database_name;
```

## Views

### Create View

```sql
-- Basic view
CREATE VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE condition;

-- Updatable view
CREATE VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE condition
WITH [CASCADED | LOCAL] CHECK OPTION;

-- Materialized view (saved result set)
CREATE MATERIALIZED VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE condition;
```

### Alter View

```sql
-- Rename view
ALTER VIEW view_name RENAME TO new_view_name;

-- Change owner
ALTER VIEW view_name OWNER TO new_owner;
```

### Drop View

```sql
-- Regular view
DROP VIEW [IF EXISTS] view_name [CASCADE | RESTRICT];
```

```
-- Materialized view
DROP MATERIALIZED VIEW [IF EXISTS] view_name [CASCADE | RESTRICT];
```

### Refresh Materialized View

```
-- Complete refresh
REFRESH MATERIALIZED VIEW view_name;

-- Concurrent refresh (doesn't block reads)
REFRESH MATERIALIZED VIEW CONCURRENTLY view_name;
```

## Functions and Stored Procedures

### Create Function

```
CREATE [OR REPLACE] FUNCTION function_name(param1 type, param2 type)
RETURNS return_type
LANGUAGE language_name
[IMMUTABLE | STABLE | VOLATILE]
[PARALLEL SAFE | PARALLEL RESTRICTED | PARALLEL UNSAFE]
AS $$
    -- Function body
$$;

-- Example PL/pgSQL function
CREATE OR REPLACE FUNCTION get_employee_salary(emp_id integer)
RETURNS numeric
LANGUAGE plpgsql
AS $$
DECLARE
    emp_salary numeric;
BEGIN
    SELECT salary INTO emp_salary
    FROM employees
    WHERE employee_id = emp_id;

    RETURN emp_salary;
END;
$$;
```

### Create Procedure (PostgreSQL 11+)

```
CREATE [OR REPLACE] PROCEDURE procedure_name(param1 type, param2 type)
LANGUAGE language_name
AS $$
    -- Procedure body
```

```sql
$$;

-- Example PL/pgSQL procedure
CREATE OR REPLACE PROCEDURE update_employee_salary(
    emp_id integer,
    new_salary numeric
)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE employees
    SET salary = new_salary
    WHERE employee_id = emp_id;

    COMMIT;
END;
$$;
```

### Alter Function/Procedure

```sql
-- Rename function
ALTER FUNCTION function_name(param_types) RENAME TO new_name;

-- Change owner
ALTER FUNCTION function_name(param_types) OWNER TO new_owner;

-- Rename procedure (PostgreSQL 11+)
ALTER PROCEDURE procedure_name(param_types) RENAME TO new_name;
```

### Drop Function/Procedure

```sql
-- Drop function
DROP FUNCTION [IF EXISTS] function_name(param_types) [CASCADE | RESTRICT];

-- Drop procedure (PostgreSQL 11+)
DROP PROCEDURE [IF EXISTS] procedure_name(param_types) [CASCADE |
RESTRICT];
```

### Call Procedure (PostgreSQL 11+)

```sql
CALL procedure_name(arg1, arg2);
```

## Transactions

### Basic Transaction Control

```sql
-- Begin a transaction
BEGIN;
-- or
BEGIN TRANSACTION;
-- or
BEGIN WORK;

-- Execute statements
INSERT INTO table_name VALUES (1, 'value');
UPDATE another_table SET column = 'new value' WHERE id = 1;

-- Commit changes
COMMIT;
-- or
COMMIT TRANSACTION;
-- or
COMMIT WORK;

-- Rollback (undo) changes
ROLLBACK;
-- or
ROLLBACK TRANSACTION;
-- or
ROLLBACK WORK;
```

## Savepoints

```sql
-- Begin transaction
BEGIN;

-- Execute statements
INSERT INTO table_name VALUES (1, 'value');

-- Create a savepoint
SAVEPOINT my_savepoint;

-- Execute more statements
UPDATE table_name SET column = 'new value' WHERE id = 1;

-- Rollback to savepoint (undo updates but keep inserts)
ROLLBACK TO SAVEPOINT my_savepoint;

-- Release savepoint (remove it)
RELEASE SAVEPOINT my_savepoint;

-- Commit remaining changes
COMMIT;
```

### Transaction Isolation Levels

```
-- Set transaction isolation level
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- or
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- or
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- or
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Execute statements
SELECT * FROM table_name;

-- Commit
COMMIT;
```

### Transaction Modes

```
-- Read-only transaction
BEGIN;
SET TRANSACTION READ ONLY;

-- Read-write transaction (default)
BEGIN;
SET TRANSACTION READ WRITE;

-- Deferrable transaction
BEGIN;
SET TRANSACTION DEFERRABLE;
```

## Users and Privileges

### Create User/Role

```
-- Create user
CREATE USER username WITH PASSWORD 'password';

-- Create role (can be assigned to multiple users)
CREATE ROLE role_name;

-- Create role with login capability
CREATE ROLE role_name WITH LOGIN PASSWORD 'password';
```

```
-- Create superuser
CREATE ROLE role_name WITH SUPERUSER LOGIN PASSWORD 'password';
```

## Alter User/Role

```
-- Change password
ALTER USER username WITH PASSWORD 'new_password';

-- Add attributes
ALTER ROLE role_name WITH CREATEDB CREATEROLE;

-- Remove attributes
ALTER ROLE role_name WITH NOCREATEDB NOCREATEROLE;
```

## Drop User/Role

```
DROP USER [IF EXISTS] username;
DROP ROLE [IF EXISTS] role_name;
```

## Grant Privileges

```
-- Grant table privileges
GRANT SELECT, INSERT, UPDATE, DELETE ON table_name TO role_name;

-- Grant all privileges on table
GRANT ALL PRIVILEGES ON table_name TO role_name;

-- Grant column-level privileges
GRANT SELECT (column1, column2), UPDATE (column1) ON table_name TO
role_name;

-- Grant database privileges
GRANT CREATE ON DATABASE database_name TO role_name;

-- Grant schema privileges
GRANT USAGE ON SCHEMA schema_name TO role_name;

-- Grant with grant option (recipient can grant to others)
GRANT SELECT ON table_name TO role_name WITH GRANT OPTION;
```

## Revoke Privileges

```
-- Revoke table privileges
REVOKE SELECT, INSERT ON table_name FROM role_name;
```

```sql
-- Revoke all privileges
REVOKE ALL PRIVILEGES ON table_name FROM role_name;

-- Revoke grant option only
REVOKE GRANT OPTION FOR SELECT ON table_name FROM role_name;
```

### Role Membership

```sql
-- Grant role to another role
GRANT role1 TO role2;

-- Revoke role from another role
REVOKE role1 FROM role2;
```

## PostgreSQL-Specific Features

### Arrays

```sql
-- Create array column
CREATE TABLE table_name (
    id serial PRIMARY KEY,
    tags TEXT[]
);

-- Insert arrays
INSERT INTO table_name (tags) VALUES (ARRAY['tag1', 'tag2', 'tag3']);
INSERT INTO table_name (tags) VALUES ('{"tag1", "tag2", "tag3"}');

-- Query arrays
SELECT * FROM table_name WHERE 'tag1' = ANY(tags);
SELECT * FROM table_name WHERE tags @> ARRAY['tag1', 'tag2'];
SELECT * FROM table_name WHERE tags && ARRAY['tag1', 'tag4']; -- Overlap

-- Array functions
SELECT array_length(tags, 1) FROM table_name; -- Length of dimension 1
SELECT array_to_string(tags, ', ') FROM table_name; -- Join elements
SELECT unnest(tags) FROM table_name; -- Expand array to rows
```

### JSON/JSONB

```sql
-- Create JSON/JSONB column
CREATE TABLE table_name (
    id serial PRIMARY KEY,
    data JSON,
    data_b JSONB  -- Binary JSON, more efficient for operations
);
```

```
-- Insert JSON data
INSERT INTO table_name (data, data_b)
VALUES ('{"name": "John", "age": 30}', '{"name": "John", "age": 30}');

-- Query JSON
SELECT data->>'name' AS name FROM table_name; -- Get as text
SELECT data->'age' AS age FROM table_name;    -- Get as JSON
SELECT * FROM table_name WHERE data->>'name' = 'John';

-- JSONB operations (more efficient than JSON)
SELECT * FROM table_name WHERE data_b @> '{"name": "John"}';
SELECT * FROM table_name WHERE data_b ? 'name'; -- Has key
SELECT * FROM table_name WHERE data_b ?| ARRAY['name', 'email']; -- Has
any key
SELECT * FROM table_name WHERE data_b ?& ARRAY['name', 'age']; -- Has all
keys

-- JSONB modification
UPDATE table_name SET data_b = data_b || '{"email":
"john@example.com"}'::jsonb;
UPDATE table_name SET data_b = jsonb_set(data_b, '{age}', '31');
UPDATE table_name SET data_b = data_b - 'email'; -- Remove key
```

## Full-Text Search

```
-- Create tsvector column for full-text search
CREATE TABLE articles (
    id serial PRIMARY KEY,
    title TEXT,
    body TEXT,
    search_vector TSVECTOR
);

CREATE INDEX articles_search_idx ON articles USING GIN (search_vector);

-- Update search vector
UPDATE articles
SET search_vector =
    setweight(to_tsvector('english', COALESCE(title, '')), 'A') ||
    setweight(to_tsvector('english', COALESCE(body, '')), 'B');

-- Simple search
SELECT id, title, body
FROM articles
WHERE search_vector @@ to_tsquery('english', 'search & terms');

-- Ranking search results
```

```sql
SELECT id, title, body,
    ts_rank(search_vector, query) AS rank
FROM articles, to_tsquery('english', 'search & terms') query
WHERE search_vector @@ query
ORDER BY rank DESC;

-- Highlight search results
SELECT id, title,
    ts_headline('english', body, to_tsquery('english', 'search & terms'))
FROM articles
WHERE search_vector @@ to_tsquery('english', 'search & terms');
```

## Inheritance

```sql
-- Create parent table
CREATE TABLE cities (
    id serial PRIMARY KEY,
    name text,
    population integer
);

-- Create child table that inherits
CREATE TABLE capitals (
    country text,
    elevation integer
) INHERITS (cities);

-- Query from parent (includes child rows)
SELECT * FROM cities;

-- Query only from parent
SELECT * FROM ONLY cities;
```

## Table Partitioning

```sql
-- Declarative partitioning (PostgreSQL 10+)

-- Create partitioned table
CREATE TABLE measurements (
    id serial,
    logdate date,
    peaktemp integer,
    unitsales integer
) PARTITION BY RANGE (logdate);

-- Create partitions
CREATE TABLE measurements_y2022 PARTITION OF measurements
```

```
    FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');

CREATE TABLE measurements_y2023 PARTITION OF measurements
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

-- Insert data (automatically routes to correct partition)
INSERT INTO measurements (logdate, peaktemp, unitsales)
VALUES ('2022-06-01', 98, 1234);
```

## Extensions

```
-- List available extensions
SELECT name, default_version, installed_version
FROM pg_available_extensions;

-- Install extension
CREATE EXTENSION extension_name;

-- Common extensions
CREATE EXTENSION postgis;       -- Spatial database
CREATE EXTENSION pg_stat_statements; -- Query statistics
CREATE EXTENSION pgcrypto;      -- Cryptographic functions
CREATE EXTENSION uuid-ossp;     -- UUID generation
CREATE EXTENSION hstore;        -- Key-value store
CREATE EXTENSION pg_trgm;       -- Trigram matching for fuzzy search
```

## Performance Tips

### Explain Analyze

```
-- View query plan
EXPLAIN SELECT * FROM table_name WHERE condition;

-- Execute query and show actual times and row counts
EXPLAIN ANALYZE SELECT * FROM table_name WHERE condition;

-- Format output as JSON
EXPLAIN (FORMAT JSON) SELECT * FROM table_name WHERE condition;

-- Verbose output with more details
EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM table_name WHERE condition;
```

### Table Statistics

```
-- Update statistics
ANALYZE [table_name];
```

```
-- Set statistics target for a column
ALTER TABLE table_name ALTER COLUMN column_name SET STATISTICS 1000;
```

## Optimize Queries

```
-- Use parameterized queries in your application code
-- Bad:  SELECT * FROM users WHERE name = 'John';
-- Good: SELECT * FROM users WHERE name = $1; (with parameter 'John')

-- Avoid SELECT * - specify only needed columns
-- Bad:  SELECT * FROM large_table;
-- Good: SELECT id, name FROM large_table;

-- Use EXISTS for existence check
-- Bad:  SELECT 1 FROM table WHERE EXISTS (SELECT * FROM another_table);
-- Good: SELECT 1 FROM table WHERE EXISTS (SELECT 1 FROM another_table);

-- Use COUNT(*) for counting all rows
-- Bad:  SELECT COUNT(id) FROM table;
-- Good: SELECT COUNT(*) FROM table;

-- Use appropriate indexes for JOIN, WHERE, ORDER BY
```

## Vacuuming

```
-- Reclaim space and update statistics
VACUUM [table_name];

-- More thorough vacuum (locks table)
VACUUM FULL [table_name];

-- Run vacuum and analyze in one command
VACUUM ANALYZE [table_name];
```

## Monitoring

```
-- Check activity
SELECT * FROM pg_stat_activity;

-- Identify slow queries
SELECT query, calls, total_time, rows,
       (total_time/calls) as avg_time
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 20;
```

```sql
-- Check index usage
SELECT * FROM pg_stat_user_indexes;

-- Check table I/O statistics
SELECT * FROM pg_statio_user_tables;
```

## Connection Pooling

```
-- Not SQL, but configuration tips:
-- 1. Use a connection pooler like PgBouncer or Odyssey
-- 2. Configure max_connections appropriately
-- 3. Reuse connections in application code
```