Complete SQL & PostgreSQL Guide - Beginner to Advanced

Table of Contents

Introduction to PostgreSQL

PostgreSQL is a powerful, open-source relational database management system (RDBMS). It's known for its reliability, feature robustness, and performance. Unlike some databases, PostgreSQL is ACID-compliant (Atomicity, Consistency, Isolation, Durability) and supports both SQL and JSON querying.

Key Features:

- ACID compliance
- Support for advanced data types
- Extensible with custom functions
- Strong community support
- Cross-platform compatibility

Installation and Setup

Installing PostgreSQL

On Ubuntu/Debian:

```
sudo apt update
sudo apt install postgresql postgresql-contrib
```

On macOS (using Homebrew):

```
brew install postgresql
brew services start postgresql
```

On Windows: Download from the official PostgreSQL website and run the installer.

First-Time Setup

```
# Switch to postgres user (Linux/macOS)
sudo -u postgres psql

# Create a new database user
CREATE USER myuser WITH PASSWORD 'mypassword';

# Create a new database
CREATE DATABASE mydatabase OWNER myuser;

# Grant privileges
GRANT ALL PRIVILEGES ON DATABASE mydatabase TO myuser;
```

Connecting to PostgreSQL

```
# Connect to default database
psql -U postgres

# Connect to specific database
psql -U myuser -d mydatabase -h localhost

# Connect with password prompt
psql -U myuser -d mydatabase -h localhost -W
```

Basic Database Operations

Creating a Database

```
-- Create a new database
CREATE DATABASE company_db;

-- Create database with specific encoding
CREATE DATABASE company_db
WITH ENCODING 'UTF8'
LC_COLLATE='en_US.UTF-8'
LC_CTYPE='en_US.UTF-8';
```

Listing Databases

```
-- List all databases
\l

-- Or using SQL
SELECT datname FROM pg_database;
```

Switching Databases

```
-- In psql command line
\c database_name

-- Or reconnect
\q
psql -U username -d database_name
```

Dropping a Database

```
-- Delete a database (be careful!)
DROP DATABASE IF EXISTS old_database;
```

Data Types

PostgreSQL supports a rich set of data types. Here are the most commonly used ones:

Numeric Types

```
-- Integer types
SMALLINT        -- 2 bytes, -32,768 to 32,767
INTEGER or INT  -- 4 bytes, -2,147,483,648 to 2,147,483,647
```

```
BIGINT          -- 8 bytes, very large numbers

-- Decimal types
DECIMAL(precision, scale)  -- Exact decimal
NUMERIC(precision, scale)  -- Same as DECIMAL
REAL                       -- 4-byte floating point
DOUBLE PRECISION          -- 8-byte floating point

-- Auto-incrementing
SERIAL      -- Auto-incrementing 4-byte integer
BIGSERIAL   -- Auto-incrementing 8-byte integer
```

Character Types

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

Date and Time Types

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

Boolean Type

```
BOOLEAN            -- TRUE, FALSE, or NULL
```

Other Important Types

```
UUID               -- Universally unique identifier
JSON               -- JSON data
JSONB              -- Binary JSON (faster, indexable)
ARRAY              -- Array of any data type
```

Creating and Managing Tables

Creating Tables

```sql
-- Basic table creation
CREATE TABLE employees (
    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 DECIMAL(10,2),
    is_active BOOLEAN DEFAULT TRUE
);

-- Table with foreign key
CREATE TABLE departments (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    manager_id INTEGER REFERENCES employees(id)
);
```

Viewing Table Structure

```sql
-- Describe table structure
\d table_name

-- List all tables
\dt

-- Show table with details
\d+ table_name
```

Modifying Tables

```sql
-- Add a column
ALTER TABLE employees ADD COLUMN phone VARCHAR(20);

-- Modify column type
ALTER TABLE employees ALTER COLUMN phone TYPE VARCHAR(15);

-- Rename column
ALTER TABLE employees RENAME COLUMN phone TO phone_number;

-- Drop column
ALTER TABLE employees DROP COLUMN phone_number;

-- Add constraint
ALTER TABLE employees ADD CONSTRAINT check_salary CHECK (salary > 0);
```

Dropping Tables

```sql
-- Drop a table
DROP TABLE IF EXISTS old_table;

-- Drop with cascade (removes dependent objects)
DROP TABLE old_table CASCADE;
```

Basic SQL Queries

SELECT Statement

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

-- Select specific columns
SELECT first_name, last_name, email FROM employees;

-- Select with alias
SELECT first_name AS "First Name",
       last_name AS "Last Name"
FROM employees;
```

INSERT Statement

```sql
-- Insert single row
INSERT INTO employees (first_name, last_name, email, salary)
VALUES ('John', 'Doe', 'john.doe@email.com', 50000);

-- Insert multiple rows
INSERT INTO employees (first_name, last_name, email, salary)
VALUES
    ('Jane', 'Smith', 'jane.smith@email.com', 55000),
    ('Bob', 'Johnson', 'bob.johnson@email.com', 48000);

-- Insert with returning clause
INSERT INTO employees (first_name, last_name, email)
VALUES ('Alice', 'Brown', 'alice.brown@email.com')
RETURNING id, first_name;
```

UPDATE Statement

```sql
-- Update single row
UPDATE employees
SET salary = 52000
WHERE id = 1;
```

```sql
-- Update multiple columns
UPDATE employees
SET salary = salary * 1.1,
    hire_date = CURRENT_DATE
WHERE department = 'Engineering';

-- Update with JOIN
UPDATE employees
SET salary = salary * 1.05
FROM departments
WHERE employees.dept_id = departments.id
AND departments.name = 'Sales';
```

DELETE Statement

```sql
-- Delete specific rows
DELETE FROM employees WHERE is_active = FALSE;

-- Delete with condition
DELETE FROM employees
WHERE hire_date < '2020-01-01'
AND salary < 40000;

-- Delete all rows (keep table structure)
DELETE FROM employees;
```

Filtering and Sorting Data

WHERE Clause

```sql
-- Basic conditions
SELECT * FROM employees WHERE salary > 50000;
SELECT * FROM employees WHERE department = 'Engineering';
SELECT * FROM employees WHERE hire_date >= '2023-01-01';

-- Multiple conditions
SELECT * FROM employees
WHERE salary > 45000 AND department = 'Sales';

SELECT * FROM employees
WHERE salary > 60000 OR department = 'Management';

-- NOT condition
SELECT * FROM employees WHERE NOT department = 'HR';
```

Pattern Matching

```sql
-- LIKE operator (case-sensitive)
SELECT * FROM employees WHERE first_name LIKE 'J%';    -- Starts with J
SELECT * FROM employees WHERE email LIKE '%@gmail.com'; -- Ends with @gmail.com
SELECT * FROM employees WHERE first_name LIKE '_ohn';   -- 4 letters ending in
'ohn'

-- ILIKE operator (case-insensitive)
SELECT * FROM employees WHERE first_name ILIKE 'john%';

-- Regular expressions
SELECT * FROM employees WHERE first_name ~ '^[A-M]';   -- Starts with A-M
SELECT * FROM employees WHERE email ~* '\.com$';       -- Case-insensitive, ends
with .com
```

Range and List Conditions

```sql
-- BETWEEN operator
SELECT * FROM employees WHERE salary BETWEEN 40000 AND 60000;
SELECT * FROM employees WHERE hire_date BETWEEN '2022-01-01' AND '2023-12-31';

-- IN operator
SELECT * FROM employees WHERE department IN ('Sales', 'Marketing', 'HR');
SELECT * FROM employees WHERE id IN (1, 3, 5, 7);

-- NULL checks
SELECT * FROM employees WHERE manager_id IS NULL;
SELECT * FROM employees WHERE phone IS NOT NULL;
```

Sorting Data

```sql
-- ORDER BY single column
SELECT * FROM employees ORDER BY salary DESC;
SELECT * FROM employees ORDER BY last_name ASC;

-- ORDER BY multiple columns
SELECT * FROM employees
ORDER BY department ASC, salary DESC;

-- ORDER BY with expressions
SELECT first_name, last_name, salary
FROM employees
ORDER BY salary * 12 DESC;  -- Annual salary

-- LIMIT and OFFSET
SELECT * FROM employees ORDER BY salary DESC LIMIT 10;         -- Top 10
SELECT * FROM employees ORDER BY hire_date LIMIT 5 OFFSET 10; -- Skip 10, take 5
```

Working with Multiple Tables

INNER JOIN

```sql
-- Basic INNER JOIN
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
INNER JOIN departments d ON e.dept_id = d.id;

-- Multiple JOINs
SELECT e.first_name, e.last_name, d.department_name, p.project_name
FROM employees e
INNER JOIN departments d ON e.dept_id = d.id
INNER JOIN projects p ON e.id = p.employee_id;
```

LEFT JOIN (LEFT OUTER JOIN)

```sql
-- LEFT JOIN - includes all records from left table
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.id;

-- Finding employees without departments
SELECT e.first_name, e.last_name
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.id
WHERE d.id IS NULL;
```

RIGHT JOIN (RIGHT OUTER JOIN)

```sql
-- RIGHT JOIN - includes all records from right table
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
RIGHT JOIN departments d ON e.dept_id = d.id;
```

FULL OUTER JOIN

```sql
-- FULL OUTER JOIN - includes all records from both tables
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
FULL OUTER JOIN departments d ON e.dept_id = d.id;
```

CROSS JOIN

```sql
-- CROSS JOIN - Cartesian product of two tables
SELECT e.first_name, d.department_name
FROM employees e
CROSS JOIN departments d;
```

Self JOIN

```sql
-- Self JOIN - joining a table with itself
SELECT
    e1.first_name AS employee,
    e2.first_name AS manager
FROM employees e1
LEFT JOIN employees e2 ON e1.manager_id = e2.id;
```

Aggregate Functions

Aggregate functions perform calculations on multiple rows and return a single result.

Basic Aggregate Functions

```sql
-- COUNT
SELECT COUNT(*) FROM employees;                    -- Count all rows
SELECT COUNT(manager_id) FROM employees;           -- Count non-NULL values
SELECT COUNT(DISTINCT department) FROM employees;  -- Count unique values

-- SUM
SELECT SUM(salary) FROM employees;                 -- Total salary
SELECT SUM(salary) FROM employees WHERE department = 'Sales';

-- AVG
SELECT AVG(salary) FROM employees;                 -- Average salary
SELECT ROUND(AVG(salary), 2) FROM employees;      -- Rounded to 2 decimals

-- MIN and MAX
SELECT MIN(salary), MAX(salary) FROM employees;
SELECT MIN(hire_date), MAX(hire_date) FROM employees;
```

String Aggregate Functions

```sql
-- STRING_AGG - concatenate strings
SELECT STRING_AGG(first_name, ', ') AS all_names
FROM employees;
```

```sql
SELECT STRING_AGG(first_name, ', ' ORDER BY first_name) AS sorted_names
FROM employees;
```

Array Aggregate Functions

```sql
-- ARRAY_AGG - create array from values
SELECT ARRAY_AGG(first_name) AS name_array
FROM employees;

SELECT ARRAY_AGG(salary ORDER BY salary DESC) AS salary_array
FROM employees;
```

Grouping and Having

GROUP BY

```sql
-- Basic grouping
SELECT department, COUNT(*)
FROM employees
GROUP BY department;

-- Multiple columns
SELECT department, hire_year, COUNT(*)
FROM (
    SELECT department, EXTRACT(YEAR FROM hire_date) AS hire_year
    FROM employees
) AS emp_year
GROUP BY department, hire_year;

-- Grouping with aggregates
SELECT
    department,
    COUNT(*) AS employee_count,
    AVG(salary) AS avg_salary,
    SUM(salary) AS total_salary
FROM employees
GROUP BY department;
```

HAVING Clause

```sql
-- HAVING filters groups (used with GROUP BY)
SELECT department, COUNT(*)
FROM employees
GROUP BY department
```

```
HAVING COUNT(*) > 5;

-- HAVING with multiple conditions
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 50000 AND COUNT(*) >= 3;
```

GROUP BY with ROLLUP and CUBE

```
-- ROLLUP - creates subtotals
SELECT department, job_title, COUNT(*)
FROM employees
GROUP BY ROLLUP(department, job_title);

-- CUBE - creates all possible combinations
SELECT department, job_title, COUNT(*)
FROM employees
GROUP BY CUBE(department, job_title);
```

Subqueries

Subqueries are queries nested inside other queries.

Scalar Subqueries

```
-- Subquery returning single value
SELECT first_name, last_name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

-- Subquery in SELECT clause
SELECT
    first_name,
    last_name,
    salary,
    (SELECT AVG(salary) FROM employees) AS avg_company_salary
FROM employees;
```

Row Subqueries

```
-- Subquery returning multiple values
SELECT first_name, last_name
FROM employees
WHERE dept_id IN (
```

```sql
    SELECT id FROM departments
    WHERE location = 'New York'
);

-- EXISTS subquery
SELECT first_name, last_name
FROM employees e
WHERE EXISTS (
    SELECT 1 FROM projects p
    WHERE p.employee_id = e.id
);

-- NOT EXISTS subquery
SELECT first_name, last_name
FROM employees e
WHERE NOT EXISTS (
    SELECT 1 FROM projects p
    WHERE p.employee_id = e.id
);
```

Correlated Subqueries

```sql
-- Correlated subquery (references outer query)
SELECT first_name, last_name, salary
FROM employees e1
WHERE salary > (
    SELECT AVG(salary)
    FROM employees e2
    WHERE e2.department = e1.department
);
```

Table Subqueries

```sql
-- Subquery in FROM clause
SELECT dept_stats.department, dept_stats.avg_salary
FROM (
    SELECT department, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department
) AS dept_stats
WHERE dept_stats.avg_salary > 50000;
```

Advanced Joins

LATERAL JOIN

```sql
-- LATERAL JOIN - allows subquery to reference columns from preceding table
SELECT e.first_name, e.last_name, recent_projects.project_name
FROM employees e
LEFT JOIN LATERAL (
    SELECT project_name
    FROM projects p
    WHERE p.employee_id = e.id
    ORDER BY p.start_date DESC
    LIMIT 3
) AS recent_projects ON true;
```

JOIN with USING

```sql
-- USING clause when column names are the same
SELECT e.first_name, d.department_name
FROM employees e
JOIN departments d USING (dept_id);
```

Natural JOIN

```sql
-- NATURAL JOIN automatically joins on columns with same names
SELECT e.first_name, d.department_name
FROM employees e
NATURAL JOIN departments d;
```

Views

Views are virtual tables based on queries.

Creating Views

```sql
-- Simple view
CREATE VIEW employee_summary AS
SELECT
    first_name,
    last_name,
    department,
    salary
FROM employees
WHERE is_active = TRUE;

-- Complex view with joins
CREATE VIEW department_stats AS
SELECT
```

```
    d.department_name,
    COUNT(e.id) AS employee_count,
    AVG(e.salary) AS avg_salary,
    SUM(e.salary) AS total_salary
FROM departments d
LEFT JOIN employees e ON d.id = e.dept_id
GROUP BY d.id, d.department_name;
```

Using Views

```
-- Query a view like a table
SELECT * FROM employee_summary;
SELECT * FROM employee_summary WHERE salary > 50000;
```

Modifying Views

```
-- Replace view definition
CREATE OR REPLACE VIEW employee_summary AS
SELECT
    first_name,
    last_name,
    department,
    salary,
    hire_date
FROM employees
WHERE is_active = TRUE;
```

Dropping Views

```
-- Drop a view
DROP VIEW IF EXISTS employee_summary;
```

Materialized Views

```
-- Materialized view (stores actual data)
CREATE MATERIALIZED VIEW monthly_sales AS
SELECT
    DATE_TRUNC('month', sale_date) AS month,
    SUM(amount) AS total_sales
FROM sales
GROUP BY DATE_TRUNC('month', sale_date);

-- Refresh materialized view
REFRESH MATERIALIZED VIEW monthly_sales;
```

```
-- Refresh concurrently (without blocking reads)
REFRESH MATERIALIZED VIEW CONCURRENTLY monthly_sales;
```

Indexes

Indexes improve query performance by creating fast access paths to data.

Creating Indexes

```
-- Simple index
CREATE INDEX idx_employee_last_name ON employees(last_name);

-- Composite index
CREATE INDEX idx_employee_dept_salary ON employees(dept_id, salary);

-- Unique index
CREATE UNIQUE INDEX idx_employee_email ON employees(email);

-- Partial index (with WHERE clause)
CREATE INDEX idx_active_employees ON employees(dept_id)
WHERE is_active = TRUE;

-- Expression index
CREATE INDEX idx_employee_full_name ON employees(lower(first_name || ' ' ||
last_name));
```

Index Types

```
-- B-tree index (default)
CREATE INDEX idx_salary_btree ON employees USING btree(salary);

-- Hash index (for equality comparisons)
CREATE INDEX idx_dept_hash ON employees USING hash(dept_id);

-- GIN index (for arrays, JSON, full-text search)
CREATE INDEX idx_skills_gin ON employees USING gin(skills);

-- GiST index (for geometric data, full-text search)
CREATE INDEX idx_location_gist ON stores USING gist(location);
```

Managing Indexes

```
-- List indexes
\di
```

```
-- Show indexes for a table
\d table_name

-- Drop index
DROP INDEX IF EXISTS idx_employee_last_name;

-- Reindex
REINDEX INDEX idx_employee_last_name;
REINDEX TABLE employees;
```

Stored Procedures and Functions

Creating Functions

```
-- Simple function
CREATE OR REPLACE FUNCTION get_employee_count()
RETURNS INTEGER AS $$
BEGIN
    RETURN (SELECT COUNT(*) FROM employees);
END;
$$ LANGUAGE plpgsql;

-- Function with parameters
CREATE OR REPLACE FUNCTION get_salary_by_id(emp_id INTEGER)
RETURNS DECIMAL AS $$
BEGIN
    RETURN (SELECT salary FROM employees WHERE id = emp_id);
END;
$$ LANGUAGE plpgsql;

-- Function returning a table
CREATE OR REPLACE FUNCTION get_high_earners(min_salary DECIMAL)
RETURNS TABLE(
    employee_id INTEGER,
    full_name TEXT,
    employee_salary DECIMAL
) AS $
BEGIN
    RETURN QUERY
    SELECT
        id,
        first_name || ' ' || last_name,
        salary
    FROM employees
    WHERE salary >= min_salary;
END;
$ LANGUAGE plpgsql;
```

Using Functions

```
-- Call simple function
SELECT get_employee_count();

-- Call function with parameters
SELECT get_salary_by_id(1);

-- Use table-returning function
SELECT * FROM get_high_earners(60000);
```

Creating Procedures

```
-- Stored procedure (PostgreSQL 11+)
CREATE OR REPLACE PROCEDURE update_employee_salary(
    emp_id INTEGER,
    new_salary DECIMAL
)
LANGUAGE plpgsql AS $
BEGIN
    UPDATE employees
    SET salary = new_salary
    WHERE id = emp_id;

    IF NOT FOUND THEN
        RAISE NOTICE 'Employee with ID % not found', emp_id;
    END IF;

    COMMIT;
END;
$;

-- Call procedure
CALL update_employee_salary(1, 55000);
```

Function Control Structures

```
-- Function with IF-ELSE
CREATE OR REPLACE FUNCTION get_bonus_percentage(emp_salary DECIMAL)
RETURNS DECIMAL AS $
BEGIN
    IF emp_salary > 80000 THEN
        RETURN 0.15;
    ELSIF emp_salary > 60000 THEN
        RETURN 0.10;
    ELSIF emp_salary > 40000 THEN
        RETURN 0.05;
    ELSE
```

```
        RETURN 0.02;
    END IF;
END;
$ LANGUAGE plpgsql;

-- Function with LOOP
CREATE OR REPLACE FUNCTION calculate_factorial(n INTEGER)
RETURNS BIGINT AS $
DECLARE
    result BIGINT := 1;
    i INTEGER;
BEGIN
    FOR i IN 1..n LOOP
        result := result * i;
    END LOOP;
    RETURN result;
END;
$ LANGUAGE plpgsql;

-- Function with exception handling
CREATE OR REPLACE FUNCTION safe_divide(a DECIMAL, b DECIMAL)
RETURNS DECIMAL AS $
BEGIN
    IF b = 0 THEN
        RAISE EXCEPTION 'Division by zero is not allowed';
    END IF;
    RETURN a / b;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Cannot divide by zero, returning NULL';
        RETURN NULL;
END;
$ LANGUAGE plpgsql;
```

Triggers

Triggers are special functions that automatically execute in response to certain events.

Creating Trigger Functions

```
-- Trigger function for updating timestamp
CREATE OR REPLACE FUNCTION update_modified_column()
RETURNS TRIGGER AS $
BEGIN
    NEW.modified_at = CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$ LANGUAGE plpgsql;

-- Trigger function for logging changes
```

```
CREATE OR REPLACE FUNCTION log_employee_changes()
RETURNS TRIGGER AS $
BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO employee_audit (action, employee_id, old_data, new_data,
changed_at)
        VALUES ('INSERT', NEW.id, NULL, row_to_json(NEW), CURRENT_TIMESTAMP);
        RETURN NEW;
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO employee_audit (action, employee_id, old_data, new_data,
changed_at)
        VALUES ('UPDATE', NEW.id, row_to_json(OLD), row_to_json(NEW),
CURRENT_TIMESTAMP);
        RETURN NEW;
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO employee_audit (action, employee_id, old_data, new_data,
changed_at)
        VALUES ('DELETE', OLD.id, row_to_json(OLD), NULL, CURRENT_TIMESTAMP);
        RETURN OLD;
    END IF;
    RETURN NULL;
END;
$ LANGUAGE plpgsql;
```

Creating Triggers

```
-- BEFORE trigger
CREATE TRIGGER tr_employee_update_modified
    BEFORE UPDATE ON employees
    FOR EACH ROW
    EXECUTE FUNCTION update_modified_column();

-- AFTER trigger
CREATE TRIGGER tr_employee_audit
    AFTER INSERT OR UPDATE OR DELETE ON employees
    FOR EACH ROW
    EXECUTE FUNCTION log_employee_changes();

-- INSTEAD OF trigger (for views)
CREATE TRIGGER tr_employee_view_insert
    INSTEAD OF INSERT ON employee_view
    FOR EACH ROW
    EXECUTE FUNCTION handle_employee_view_insert();
```

Managing Triggers

```
-- List triggers
SELECT * FROM information_schema.triggers
WHERE event_object_table = 'employees';
```

```sql
-- Disable trigger
ALTER TABLE employees DISABLE TRIGGER tr_employee_audit;

-- Enable trigger
ALTER TABLE employees ENABLE TRIGGER tr_employee_audit;

-- Drop trigger
DROP TRIGGER IF EXISTS tr_employee_audit ON employees;
```

Transactions

Transactions ensure data integrity by grouping multiple operations into a single unit.

Basic Transaction Control

```sql
-- Start transaction
BEGIN;

-- Perform operations
UPDATE employees SET salary = salary * 1.1 WHERE department = 'Sales';
INSERT INTO salary_changes (employee_id, old_salary, new_salary, change_date)
SELECT id, salary / 1.1, salary, CURRENT_DATE FROM employees WHERE department = 
'Sales';

-- Commit transaction
COMMIT;

-- Or rollback if something goes wrong
-- ROLLBACK;
```

Savepoints

```sql
-- Transaction with savepoints
BEGIN;

UPDATE employees SET salary = salary * 1.05;

SAVEPOINT after_salary_update;

DELETE FROM employees WHERE performance_rating < 2;

-- If we want to undo the delete but keep the salary update
ROLLBACK TO SAVEPOINT after_salary_update;

-- Or release the savepoint
RELEASE SAVEPOINT after_salary_update;
```

```
COMMIT;
```

Transaction Isolation Levels

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

-- Set for session
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Handling Deadlocks

```
-- Example of deadlock handling in a function
CREATE OR REPLACE FUNCTION transfer_budget(
    from_dept INTEGER,
    to_dept INTEGER,
    amount DECIMAL
) RETURNS BOOLEAN AS $
BEGIN
    BEGIN
        -- Lock departments in consistent order to prevent deadlocks
        PERFORM * FROM departments
        WHERE id IN (from_dept, to_dept)
        ORDER BY id
        FOR UPDATE;

        UPDATE departments SET budget = budget - amount WHERE id = from_dept;
        UPDATE departments SET budget = budget + amount WHERE id = to_dept;

        RETURN TRUE;
    EXCEPTION
        WHEN deadlock_detected THEN
            RAISE NOTICE 'Deadlock detected, transaction rolled back';
            RETURN FALSE;
    END;
END;
$ LANGUAGE plpgsql;
```

Window Functions

Window functions perform calculations across related rows without grouping.

Basic Window Functions

```sql
-- ROW_NUMBER
SELECT
    first_name,
    last_name,
    salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) as salary_rank
FROM employees;

-- RANK and DENSE_RANK
SELECT
    first_name,
    last_name,
    salary,
    RANK() OVER (ORDER BY salary DESC) as rank,
    DENSE_RANK() OVER (ORDER BY salary DESC) as dense_rank
FROM employees;

-- NTILE (divide into n groups)
SELECT
    first_name,
    last_name,
    salary,
    NTILE(4) OVER (ORDER BY salary DESC) as salary_quartile
FROM employees;
```

Window Functions with PARTITION BY

```sql
-- Partition by department
SELECT
    first_name,
    last_name,
    department,
    salary,
    ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as dept_rank,
    AVG(salary) OVER (PARTITION BY department) as dept_avg_salary
FROM employees;

-- Running totals
SELECT
    first_name,
    last_name,
    salary,
    SUM(salary) OVER (ORDER BY hire_date ROWS UNBOUNDED PRECEDING) as
running_total
FROM employees;
```

Frame Specifications

```sql
-- Moving average (3-row window)
SELECT
    first_name,
    salary,
    AVG(salary) OVER (
        ORDER BY hire_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) as moving_avg
FROM employees;

-- Cumulative sum
SELECT
    first_name,
    salary,
    SUM(salary) OVER (
        ORDER BY hire_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) as cumulative_sum
FROM employees;
```

LAG and LEAD Functions

```sql
-- Compare with previous/next row
SELECT
    first_name,
    salary,
    LAG(salary, 1) OVER (ORDER BY hire_date) as previous_salary,
    LEAD(salary, 1) OVER (ORDER BY hire_date) as next_salary,
    salary - LAG(salary, 1) OVER (ORDER BY hire_date) as salary_increase
FROM employees;

-- FIRST_VALUE and LAST_VALUE
SELECT
    first_name,
    department,
    salary,
    FIRST_VALUE(salary) OVER (
        PARTITION BY department
        ORDER BY salary DESC
        ROWS UNBOUNDED PRECEDING
    ) as highest_dept_salary,
    LAST_VALUE(salary) OVER (
        PARTITION BY department
        ORDER BY salary DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as lowest_dept_salary
FROM employees;
```

Common Table Expressions (CTEs)

CTEs provide a way to write more readable and maintainable complex queries.

Basic CTE

```sql
-- Simple CTE
WITH high_earners AS (
    SELECT first_name, last_name, salary, department
    FROM employees
    WHERE salary > 60000
)
SELECT department, COUNT(*) as high_earner_count
FROM high_earners
GROUP BY department;
```

Multiple CTEs

```sql
-- Multiple CTEs
WITH
department_stats AS (
    SELECT
        department,
        COUNT(*) as emp_count,
        AVG(salary) as avg_salary
    FROM employees
    GROUP BY department
),
high_performing_depts AS (
    SELECT department
    FROM department_stats
    WHERE avg_salary > 55000
)
SELECT e.first_name, e.last_name, e.salary
FROM employees e
INNER JOIN high_performing_depts hpd ON e.department = hpd.department;
```

Recursive CTEs

```sql
-- Recursive CTE for hierarchical data
WITH RECURSIVE employee_hierarchy AS (
    -- Base case: top-level managers
    SELECT id, first_name, last_name, manager_id, 0 as level
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL
    -- Recursive case: employees with managers
    SELECT e.id, e.first_name, e.last_name, e.manager_id, eh.level + 1
    FROM employees e
    INNER JOIN employee_hierarchy eh ON e.manager_id = eh.id
)
SELECT * FROM employee_hierarchy ORDER BY level, last_name;

-- Recursive CTE for generating series
WITH RECURSIVE date_series AS (
    SELECT DATE '2023-01-01' as date_value
    UNION ALL
    SELECT date_value + INTERVAL '1 day'
    FROM date_series
    WHERE date_value < DATE '2023-12-31'
)
SELECT date_value FROM date_series;
```

JSON and JSONB

PostgreSQL has excellent support for JSON data types.

JSON vs JSONB

```sql
-- JSON column (stores exact text)
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    attributes JSON
);

-- JSONB column (binary, faster, indexable)
CREATE TABLE products_jsonb (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    attributes JSONB
);
```

Inserting JSON Data

```sql
-- Insert JSON data
INSERT INTO products (name, attributes) VALUES
('Laptop', '{"brand": "Dell", "cpu": "Intel i7", "ram": "16GB", "price": 1200}'),
('Phone', '{"brand": "Apple", "model": "iPhone 13", "storage": "128GB", "price":
800}');

-- Insert JSONB data
INSERT INTO products_jsonb (name, attributes) VALUES
```

```
('Laptop', '{"brand": "Dell", "cpu": "Intel i7", "ram": "16GB", "price": 1200}'),
('Phone', '{"brand": "Apple", "model": "iPhone 13", "storage": "128GB", "price":
800}');
```

Querying JSON Data

```
-- Extract JSON field with ->
SELECT name, attributes->'brand' as brand
FROM products;

-- Extract JSON field as text with ->>
SELECT name, attributes->>'brand' as brand
FROM products;

-- Extract nested JSON
SELECT name, attributes->'specs'->>'cpu' as cpu
FROM products;

-- JSON path queries
SELECT name, attributes #> '{specs,cpu}' as cpu
FROM products;

SELECT name, attributes #>> '{specs,cpu}' as cpu_text
FROM products;
```

JSON Operators and Functions

```
-- Check if JSON contains key
SELECT * FROM products WHERE attributes ? 'brand';

-- Check if JSON contains any of the keys
SELECT * FROM products WHERE attributes ?| array['brand', 'model'];

-- Check if JSON contains all keys
SELECT * FROM products WHERE attributes ?& array['brand', 'price'];

-- JSON containment (@> and <@)
SELECT * FROM products_jsonb
WHERE attributes @> '{"brand": "Apple"}';

-- JSON functions
SELECT
    name,
    json_extract_path_text(attributes, 'brand') as brand,
    json_extract_path_text(attributes, 'price')::numeric as price
FROM products;

-- JSONB functions
SELECT
```

```
    name,
    jsonb_extract_path_text(attributes, 'brand') as brand,
    (attributes->>'price')::numeric as price
FROM products_jsonb;
```

JSON Aggregation

```
-- Aggregate to JSON
SELECT
    jsonb_object_agg(name, attributes) as all_products
FROM products_jsonb;

-- Array of JSON objects
SELECT
    jsonb_agg(jsonb_build_object('name', name, 'brand', attributes->>'brand')) as
product_list
FROM products_jsonb;
```

JSON Indexing

```
-- GIN index on JSONB
CREATE INDEX idx_products_attributes ON products_jsonb USING gin(attributes);

-- Index on specific JSON path
CREATE INDEX idx_products_brand ON products_jsonb USING btree((attributes-
>>'brand'));

-- Partial index with JSON condition
CREATE INDEX idx_expensive_products ON products_jsonb
WHERE (attributes->>'price')::numeric > 1000;
```

Full-Text Search

PostgreSQL provides powerful full-text search capabilities.

Basic Text Search

```
-- Create table with text search
CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    title TEXT,
    content TEXT,
    search_vector tsvector
);
```

```sql
-- Insert sample data
INSERT INTO articles (title, content) VALUES
('PostgreSQL Tutorial', 'Learn PostgreSQL database management system'),
('Advanced SQL Queries', 'Master complex SQL queries and optimization'),
('Database Design Principles', 'Best practices for database schema design');
```

Creating Text Search Vectors

```sql
-- Update search vector
UPDATE articles SET search_vector =
    to_tsvector('english', coalesce(title, '') || ' ' || coalesce(content, ''));

-- Using trigger to maintain search vector
CREATE OR REPLACE FUNCTION update_search_vector()
RETURNS TRIGGER AS $
BEGIN
    NEW.search_vector := to_tsvector('english',
        coalesce(NEW.title, '') || ' ' || coalesce(NEW.content, ''));
    RETURN NEW;
END;
$ LANGUAGE plpgsql;

CREATE TRIGGER tr_articles_search_vector
    BEFORE INSERT OR UPDATE ON articles
    FOR EACH ROW
    EXECUTE FUNCTION update_search_vector();
```

Text Search Queries

```sql
-- Basic text search
SELECT title, content
FROM articles
WHERE search_vector @@ to_tsquery('english', 'PostgreSQL');

-- Search with multiple terms
SELECT title, content
FROM articles
WHERE search_vector @@ to_tsquery('english', 'PostgreSQL & database');

-- Search with OR
SELECT title, content
FROM articles
WHERE search_vector @@ to_tsquery('english', 'PostgreSQL | MySQL');

-- Phrase search
SELECT title, content
FROM articles
WHERE search_vector @@ phraseto_tsquery('english', 'database management');
```

```
-- Plain text search (simpler)
SELECT title, content
FROM articles
WHERE search_vector @@ plainto_tsquery('english', 'PostgreSQL database');
```

Search Ranking

```
-- Rank search results
SELECT
    title,
    content,
    ts_rank(search_vector, to_tsquery('english', 'PostgreSQL')) as rank
FROM articles
WHERE search_vector @@ to_tsquery('english', 'PostgreSQL')
ORDER BY rank DESC;

-- Weighted ranking
SELECT
    title,
    content,
    ts_rank_cd(
        setweight(to_tsvector('english', title), 'A') ||
        setweight(to_tsvector('english', content), 'B'),
        to_tsquery('english', 'PostgreSQL')
    ) as rank
FROM articles
WHERE search_vector @@ to_tsquery('english', 'PostgreSQL')
ORDER BY rank DESC;
```

Full-Text Search Index

```
-- GIN index for full-text search
CREATE INDEX idx_articles_search ON articles USING gin(search_vector);

-- GiST index (smaller, slower updates)
CREATE INDEX idx_articles_search_gist ON articles USING gist(search_vector);
```

Performance Optimization

Query Analysis

```
-- EXPLAIN shows query execution plan
EXPLAIN SELECT * FROM employees WHERE salary > 50000;

-- EXPLAIN ANALYZE shows actual execution statistics
```

```
EXPLAIN ANALYZE SELECT * FROM employees WHERE salary > 50000;

-- More detailed analysis
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT e.first_name, d.department_name
FROM employees e
JOIN departments d ON e.dept_id = d.id
WHERE e.salary > 50000;
```

Query Optimization Techniques

```
-- Use appropriate indexes
CREATE INDEX idx_employees_salary ON employees(salary);
CREATE INDEX idx_employees_dept_salary ON employees(dept_id, salary);

-- Avoid SELECT * in production
-- Instead of: SELECT * FROM employees;
SELECT id, first_name, last_name, salary FROM employees;

-- Use LIMIT for large result sets
SELECT * FROM employees ORDER BY salary DESC LIMIT 10;

-- Use EXISTS instead of IN for large datasets
-- Less efficient:
SELECT * FROM employees WHERE dept_id IN (SELECT id FROM departments WHERE
location = 'NY');

-- More efficient:
SELECT * FROM employees e WHERE EXISTS (
    SELECT 1 FROM departments d WHERE d.id = e.dept_id AND d.location = 'NY'
);
```

Table Partitioning

```
-- Range partitioning by date
CREATE TABLE sales (
    id SERIAL,
    sale_date DATE,
    amount DECIMAL(10,2),
    customer_id INTEGER
) PARTITION BY RANGE (sale_date);

-- Create partitions
CREATE TABLE sales_2023 PARTITION OF sales
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE sales_2024 PARTITION OF sales
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
```

```
-- Hash partitioning
CREATE TABLE customers (
    id SERIAL,
    name VARCHAR(100),
    email VARCHAR(100)
) PARTITION BY HASH (id);

CREATE TABLE customers_p1 PARTITION OF customers
    FOR VALUES WITH (modulus 4, remainder 0);

CREATE TABLE customers_p2 PARTITION OF customers
    FOR VALUES WITH (modulus 4, remainder 1);
```

Maintenance Commands

```
-- Update table statistics
ANALYZE employees;
ANALYZE; -- All tables

-- Vacuum to reclaim space
VACUUM employees;
VACUUM FULL employees; -- More thorough but locks table

-- Vacuum and analyze together
VACUUM ANALYZE employees;

-- Reindex
REINDEX TABLE employees;
REINDEX DATABASE mydatabase;
```

Backup and Restore

Using pg_dump

```
# Backup single database
pg_dump -U username -h localhost -d database_name > backup.sql

# Backup with compression
pg_dump -U username -h localhost -d database_name | gzip > backup.sql.gz

# Backup in custom format
pg_dump -U username -h localhost -d database_name -Fc > backup.dump

# Backup specific tables
pg_dump -U username -h localhost -d database_name -t employees -t departments >
tables_backup.sql

# Backup schema only
```

```
pg_dump -U username -h localhost -d database_name --schema-only >
schema_backup.sql

# Backup data only
pg_dump -U username -h localhost -d database_name --data-only > data_backup.sql
```

Using pg_restore

```
# Restore from SQL file
psql -U username -h localhost -d database_name < backup.sql

# Restore from custom format
pg_restore -U username -h localhost -d database_name backup.dump

# Restore with options
pg_restore -U username -h localhost -d database_name --clean --if-exists
backup.dump

# Restore specific tables
pg_restore -U username -h localhost -d database_name -t employees backup.dump
```

Using pg_dumpall

```
# Backup all databases
pg_dumpall -U postgres > all_databases.sql

# Backup only global objects (users, roles, etc.)
pg_dumpall -U postgres --globals-only > globals.sql

# Restore all databases
psql -U postgres < all_databases.sql
```

Point-in-Time Recovery (PITR)

```
-- Enable WAL archiving in postgresql.conf
-- wal_level = replica
-- archive_mode = on
-- archive_command = 'cp %p /path/to/archive/%f'

-- Create base backup
SELECT pg_start_backup('backup_label');
-- Copy data directory
SELECT pg_stop_backup();

-- Restore to specific time
-- Create recovery.conf with:
```

```
-- restore_command = 'cp /path/to/archive/%f %p'
-- recovery_target_time = '2023-12-01 12:00:00'
```

User Management and Security

Creating Users and Roles

```
-- Create user
CREATE USER john_doe WITH PASSWORD 'secure_password';

-- Create role
CREATE ROLE developer;

-- Create user with specific attributes
CREATE USER admin_user WITH
    PASSWORD 'admin_password'
    CREATEDB
    CREATEROLE
    LOGIN;

-- Create role with inheritance
CREATE ROLE manager INHERIT;
```

Granting Permissions

```
-- Grant database permissions
GRANT CONNECT ON DATABASE company_db TO john_doe;
GRANT USAGE ON SCHEMA public TO john_doe;

-- Grant table permissions
GRANT SELECT ON employees TO john_doe;
GRANT INSERT, UPDATE ON employees TO john_doe;
GRANT ALL PRIVILEGES ON employees TO admin_user;

-- Grant permissions on all tables in schema
GRANT SELECT ON ALL TABLES IN SCHEMA public TO developer;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin_user;

-- Grant permissions on future tables
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT ON TABLES TO developer;

-- Grant sequence permissions (for SERIAL columns)
GRANT USAGE ON ALL SEQUENCES IN SCHEMA public TO john_doe;
```

Role Management

```
-- Assign role to user
GRANT developer TO john_doe;

-- Remove role from user
REVOKE developer FROM john_doe;

-- Create role hierarchy
CREATE ROLE employee;
CREATE ROLE manager;
CREATE ROLE admin;

GRANT employee TO manager;
GRANT manager TO admin;
```

Row Level Security (RLS)

```
-- Enable RLS on table
ALTER TABLE employees ENABLE ROW LEVEL SECURITY;

-- Create policy
CREATE POLICY employee_policy ON employees
    FOR SELECT
    USING (user_id = current_user_id());

-- Create policy for specific role
CREATE POLICY manager_policy ON employees
    FOR ALL
    TO manager_role
    USING (department_id IN (
        SELECT department_id FROM managers WHERE user_id = current_user_id()
    ));

-- Disable RLS for specific user
ALTER TABLE employees FORCE ROW LEVEL SECURITY;
```

Security Best Practices

```
-- Change default passwords
ALTER USER postgres PASSWORD 'new_secure_password';

-- Revoke public permissions
REVOKE ALL ON DATABASE company_db FROM public;
REVOKE ALL ON SCHEMA public FROM public;

-- Create application-specific roles
CREATE ROLE app_read;
CREATE ROLE app_write;
```

```sql
GRANT CONNECT ON DATABASE company_db TO app_read;
GRANT USAGE ON SCHEMA public TO app_read;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO app_read;

-- Use connection limits
ALTER USER john_doe CONNECTION LIMIT 5;

-- Set password expiration
ALTER USER john_doe VALID UNTIL '2024-12-31';
```

PostgreSQL-Specific Features

Arrays

```sql
-- Create table with array column
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    grades INTEGER[],
    subjects TEXT[]
);

-- Insert array data
INSERT INTO students (name, grades, subjects) VALUES
('John Doe', ARRAY[85, 90, 78], ARRAY['Math', 'Science', 'English']),
('Jane Smith', '{92, 88, 95}', '{"History", "Art", "Music"}');

-- Query array data
SELECT name, grades[1] as first_grade FROM students;
SELECT name FROM students WHERE 85 = ANY(grades);
SELECT name FROM students WHERE grades @> ARRAY[90];
SELECT name FROM students WHERE 'Math' = ANY(subjects);

-- Array functions
SELECT name, array_length(grades, 1) as num_grades FROM students;
SELECT name, unnest(subjects) as subject FROM students;
SELECT array_agg(name) as all_students FROM students;
```

Custom Data Types

```sql
-- Create enum type
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

CREATE TABLE person (
    name VARCHAR(100),
    current_mood mood
);
```

```sql
INSERT INTO person VALUES ('Alice', 'happy'), ('Bob', 'sad');

-- Create composite type
CREATE TYPE address AS (
    street VARCHAR(100),
    city VARCHAR(50),
    state VARCHAR(2),
    zip VARCHAR(10)
);

CREATE TABLE companies (
    name VARCHAR(100),
    headquarters address
);

INSERT INTO companies VALUES
('Tech Corp', ROW('123 Main St', 'San Francisco', 'CA', '94105'));
```

Extensions

```sql
-- List available extensions
SELECT * FROM pg_available_extensions;

-- Install extension
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "hstore";
CREATE EXTENSION IF NOT EXISTS "pg_trgm";

-- Use UUID extension
SELECT uuid_generate_v4();

-- Use hstore (key-value store)
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    attributes hstore
);

INSERT INTO products (name, attributes) VALUES
('Laptop', 'brand => "Dell", cpu => "i7", ram => "16GB"');

SELECT name, attributes->'brand' as brand FROM products;
```

Advanced Data Types

```sql
-- Range types
CREATE TABLE reservations (
    id SERIAL PRIMARY KEY,
    room_id INTEGER,
    during tsrange
);

INSERT INTO reservations (room_id, during) VALUES
(1, '[2023-12-01 10:00, 2023-12-01 12:00)'),
(2, '[2023-12-01 14:00, 2023-12-01 16:00)');

-- Check for overlapping reservations
SELECT * FROM reservations
WHERE during && '[2023-12-01 11:00, 2023-12-01 13:00)';

-- Network address types
CREATE TABLE servers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    ip_address inet,
    network cidr
);

INSERT INTO servers (name, ip_address, network) VALUES
('Web Server', '192.168.1.100', '192.168.1.0/24');

SELECT * FROM servers WHERE ip_address << '192.168.1.0/24';
```

Regular Expressions

```
-- Pattern matching with regex
SELECT * FROM employees WHERE first_name ~ '^[A-M]'; -- Starts with A-M
SELECT * FROM employees WHERE email ~* '\.com;     -- Ends with .com (case-
insensitive)

-- Extract with regex
SELECT
    email,
    substring(email from '([^@]+)@') as username,
    substring(email from '@(.+)') as domain
FROM employees;

-- Replace with regex
SELECT regexp_replace(phone, '[^0-9]', '', 'g') as clean_phone
FROM employees;
```

Common Use Cases and Examples

Data Migration

```
-- Migrate data between tables
INSERT INTO new_employees (first_name, last_name, email, salary)
```

```sql
SELECT first_name, last_name, email, salary
FROM old_employees
WHERE is_active = true;

-- Bulk update with JOIN
UPDATE employees
SET department_id = d.id
FROM departments d
WHERE employees.department_name = d.name;

-- Copy table structure
CREATE TABLE employees_backup (LIKE employees INCLUDING ALL);

-- Copy table with data
CREATE TABLE employees_backup AS SELECT * FROM employees;
```

Reporting Queries

```sql
-- Monthly sales report
SELECT
    DATE_TRUNC('month', order_date) as month,
    COUNT(*) as total_orders,
    SUM(amount) as total_revenue,
    AVG(amount) as avg_order_value
FROM orders
WHERE order_date >= CURRENT_DATE - INTERVAL '12 months'
GROUP BY DATE_TRUNC('month', order_date)
ORDER BY month;

-- Top performers by department
WITH dept_rankings AS (
    SELECT
        department,
        first_name,
        last_name,
        salary,
        ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as rank
    FROM employees
)
SELECT * FROM dept_rankings WHERE rank <= 3;

-- Year-over-year comparison
SELECT
    product_id,
    SUM(CASE WHEN EXTRACT(YEAR FROM sale_date) = 2023 THEN amount ELSE 0 END) as
sales_2023,
    SUM(CASE WHEN EXTRACT(YEAR FROM sale_date) = 2024 THEN amount ELSE 0 END) as
sales_2024,
    ROUND(
        (SUM(CASE WHEN EXTRACT(YEAR FROM sale_date) = 2024 THEN amount ELSE 0 END)
-
```

```
         SUM(CASE WHEN EXTRACT(YEAR FROM sale_date) = 2023 THEN amount ELSE 0 
END)) * 100.0 /
        NULLIF(SUM(CASE WHEN EXTRACT(YEAR FROM sale_date) = 2023 THEN amount ELSE 
0 END), 0), 2
    ) as growth_percentage
FROM sales
GROUP BY product_id;
```

Data Validation

```
-- Find duplicate records
SELECT email, COUNT(*)
FROM employees
GROUP BY email
HAVING COUNT(*) > 1;

-- Find orphaned records
SELECT e.*
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.id
WHERE d.id IS NULL;

-- Data quality checks
SELECT
    'employees' as table_name,
    COUNT(*) as total_records,
    COUNT(CASE WHEN email IS NULL OR email = '' THEN 1 END) as missing_email,
    COUNT(CASE WHEN salary <= 0 THEN 1 END) as invalid_salary,
    COUNT(CASE WHEN hire_date > CURRENT_DATE THEN 1 END) as future_hire_date
FROM employees;
```

Quick Reference Commands

psql Commands

```
\l                  # List databases
\c dbname           # Connect to database
\dt                 # List tables
\d table_name       # Describe table
\du                 # List users
\dp                 # List table permissions
\df                 # List functions
\dv                 # List views
\di                 # List indexes
\q                  # Quit psql
\h                  # Help with SQL commands
\?                  # Help with psql commands
```

```
\timing on          # Show query execution time
\x                  # Toggle expanded output
```

Useful System Tables

```sql
-- Database information
SELECT * FROM pg_database;

-- Table information
SELECT * FROM information_schema.tables WHERE table_schema = 'public';

-- Column information
SELECT * FROM information_schema.columns WHERE table_name = 'employees';

-- Index information
SELECT * FROM pg_indexes WHERE tablename = 'employees';

-- Current connections
SELECT * FROM pg_stat_activity;

-- Database sizes
SELECT
    datname,
    pg_size_pretty(pg_database_size(datname)) as size
FROM pg_database;

-- Table sizes
SELECT
    schemaname,
    tablename,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) as size
FROM pg_tables
WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC;
```

Performance Tips Summary

1. **Use appropriate indexes** - Create indexes on frequently queried columns
2. **Avoid SELECT *** - Only select columns you need
3. **Use LIMIT** - Limit result sets when possible
4. **Use EXPLAIN** - Analyze query execution plans
5. **Keep statistics updated** - Run ANALYZE regularly
6. **Use connection pooling** - For high-traffic applications
7. **Partition large tables** - Split large tables into smaller ones
8. **Use appropriate data types** - Choose the most efficient data types
9. **Avoid unnecessary JOINs** - Only join tables when needed
10. **Monitor slow queries** - Use pg_stat_statements extension

---

This guide covers PostgreSQL from basic concepts to advanced features. Practice with real data and gradually work through the examples to build your expertise. Remember that PostgreSQL is a powerful database system with many more features than covered here - this guide provides a solid foundation for further exploration.