PostgreSQL Cheatsheet

PostgreSQL Cheatsheet

PostgreSQL is a powerful, open-source object-relational database system. This guide provides deep technical reference for administration, advanced SQL, and performance tuning.

Installation & Setup

```
# Install PostgreSQL (Debian/Ubuntu)
sudo apt update
sudo apt install postgresql postgresql-contrib

# Switch to postgres user
sudo -i -u postgres
psql

# Access a specific database with a user
psql -d my_database -U my_user -h localhost

# List all databases
\l

# List all tables in current database
\dt

# List all users/roles
\du

# Show table structure (describe)
\d table_name

# Show table of columns
\d+ table_name
```

Basic SQL Operations

CRUD Operations

```
-- Create a table
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
```

```
    username TEXT NOT NULL,
    email VARCHAR(255) UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert data
INSERT INTO users (username, email)
VALUES ('johndoe', 'john@example.com');

-- Select data
SELECT * FROM users WHERE username = 'johndoe';
SELECT id, username FROM users LIMIT 10;

-- Update data
UPDATE users SET email = 'john_new@example.com' WHERE id = 1;

-- Delete data
DELETE FROM users WHERE id = 1;
```

Joins & Relationships

```
-- Inner Join
SELECT orders.id, users.username
FROM orders
JOIN users ON orders.user_id = users.id;

-- Left Join (all users, even without orders)
SELECT users.username, orders.amount
FROM users
LEFT JOIN orders ON users.id = orders.user_id;

-- Self Join
SELECT a.name, b.name
FROM employees a, employees b
WHERE a.manager_id = b.employee_id;
```

Advanced SQL & Data Structures

JSONB Deep Dive

```
-- Create a table with JSONB
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    metadata JSONB
);

-- Insert JSONB data
INSERT INTO products (metadata) VALUES ('{"brand": "Apple", "specs": {"cpu": "M2", "ram": "16

-- Query JSONB with containment operator (@>)
SELECT * FROM products WHERE metadata @> '{"brand": "Apple"}';

-- Access nested values with ->>
SELECT metadata->'specs'->>'cpu' AS cpu_type FROM products;

-- Use GIN index for JSONB performance
```

```
CREATE INDEX idx_metadata_gin ON products USING GIN (metadata);

-- JSONB path expressions
SELECT * FROM products
WHERE metadata @@ '$.specs.cpu == "M2"';
```

Common Table Expressions (CTEs)

```
-- Simple CTE
WITH regional_sales AS (
    SELECT region, SUM(amount) as total_sales
    FROM sales
    GROUP BY region
)
SELECT region, total_sales
FROM regional_sales
WHERE total_sales > 10000;

-- Recursive CTE (Hierarchical Data)
WITH RECURSIVE sub_tree AS (
    SELECT id, name, parent_id FROM categories WHERE parent_id IS NULL
  UNION ALL
    SELECT c.id, c.name, c.parent_id
    FROM categories c
    JOIN sub_tree st ON c.parent_id = st.id
)
SELECT * FROM sub_tree;
```

Window Functions

```
-- Row Numbering
SELECT name, salary,
       ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as row_num,
       RANK() OVER (PARTITION BY department ORDER BY salary DESC) as rank,
       DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dense_rank
FROM employees;

-- Moving Averages & Lags
SELECT date, revenue,
       AVG(revenue) OVER (ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) as moving_a
       LAG(revenue) OVER (ORDER BY date) as prev_day_revenue,
       LEAD(revenue) OVER (ORDER BY date) as next_day_revenue
FROM daily_sales;
```

Administration & Maintenance

User & Role Management

```
-- Create a user with password
CREATE USER web_user WITH PASSWORD 'strong_password';

-- Grant privileges on a table
GRANT SELECT, INSERT, UPDATE ON users TO web_user;
```

```
-- Grant all privileges on a database
GRANT ALL PRIVILEGES ON DATABASE my_db TO web_user;

-- Role Hierarchy
CREATE ROLE readonly_role;
GRANT USAGE ON SCHEMA public TO readonly_role;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly_role;

-- Change a user's password
ALTER USER web_user WITH PASSWORD 'new_password';
```

Database Maintenance

```
-- Vacuuming & Analyzing
VACUUM;                      -- Standard vacuum
VACUUM FULL;                 -- Compacts the database (requires exclusive lock)
ANALYZE;                     -- Updates statistics
VACUUM ANALYZE users;       -- Both vacuum and analyze a specific table

-- Reindex
REINDEX TABLE users;         -- Rebuilds indexes for the table
REINDEX DATABASE my_db;      -- Rebuilds all indexes in the database

-- Maintenance stats
SELECT relname, reltuples, relsize FROM pg_class WHERE relname = 'users';
```

Backup and Restore

```
# Dump a single database to a file
pg_dump -U username -d db_name > db_backup.sql

# Restore from a SQL file
psql -U username -d db_name < db_backup.sql

# Full cluster backup
pg_dumpall -U username > full_cluster_backup.sql

# Restore full cluster
psql -U username -f full_cluster_backup.sql
```

High-Performance Architecture

Indexing Strategies

```
-- B-Tree (Default)
CREATE INDEX idx_users_email ON users(email);

-- GIN (Generalized Inverted Index) - Best for JSONB/Arrays
CREATE INDEX idx_metadata_gin ON products USING GIN (metadata);

-- GiST (Generalized Search Tree) - Best for geometry/proximity
CREATE INDEX idx_location ON points USING GiST (geom);

-- BRIN (Block Range Index) - For massive, naturally ordered datasets
```

```
CREATE INDEX idx_timestamp ON logs USING BRIN (created_at);

-- Expression Index (Functional)
CREATE INDEX idx_lower_email ON users (LOWER(email));
```

Partitioning (Declarative)

```
-- Range Partitioning (By Date)
CREATE TABLE measurement (
    city_id         int,
    log_date        date NOT NULL,
    resp_time       float
) PARTITION BY RANGE (log_date);

-- Creating a partition for Jan 2023
CREATE TABLE measurement_y2023m01
    PARTITION OF measurement
    FOR VALUES FROM ('2023-01-01') TO ('2023-02-01');

-- List Partitioning (By Region)
CREATE TABLE user_logs (...) PARTITION BY LIST (region);
CREATE TABLE logs_us PART_OF user_logs FOR VALUES IN ('US', 'CA');

-- Hash Partitioning (By ID)
CREATE TABLE users_hashed (...) PARTITION BY HASH (id);
CREATE TABLE users_part_1 PARTITION OF users_hashed FOR VALUES $= 0$;
```

Performance Tuning & Optimization

The EXPLAIN Command

```
-- Simple Explain
EXPLAIN SELECT * FROM users WHERE id = 1;

-- Detailed Explain with actual execution times
EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'test@example.com';

-- Explain with Buffer Cache Statistics
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM products WHERE metadata @> '{"brand": "Apple"}';

-- Show the most expensive nodes in a plan
EXPLAIN (FORMAT JSON, ANALYZE) SELECT * FROM orders;
```

Memory & Configuration Tuning

```
-- Check current configuration
SHOW max_connections;
SHOW shared_buffers;
SHOW work_mem;
SHOW effective_cache_size;

-- Change configuration at session level
SET work_mem = '64MB';
```

```
SET client_encoding = 'UTF8';
SET lock_timeout = '1s';

-- Adjusting autovacuum settings
ALTER TABLE users SET (autovacuum_enabled = true);
```

Locking & Concurrency (MVCC)

```
-- Explicit Row-Level Locking
SELECT * FROM users WHERE id = 1 FOR UPDATE;     -- Strongest lock
SELECT * FROM users WHERE id = 1 FOR SHARE;     -- Shared lock
SELECT * FROM users WHERE id = 1 FOR UPDATE OF name;

-- Transaction Isolation Levels
BEGIN ISOLATION LEVEL SERIALIZABLE;
-- (High safety, high contention)

BEGIN ISOLATION LEVEL READ COMMITTED;
-- (Default: reads only committed data)

-- Advisory Locks (Application-level)
SELECT pg_advisory_lock(12345);
SELECT pg_advisory_unlock(12345);
```

Troubleshooting & Monitoring

System Statistics & Inspection

```
-- View active connections and long-running queries
SELECT pid, usename, query, state, wait_event
FROM pg_stat_activity
WHERE state != 'idle';

-- Check table size and bloat
SELECT pg_size_pretty(pg_relation_size('users'));
SELECT pg_size_pretty(pg_total_relation_size('users'));

-- Check current locks
SELECT l.pid, a.datname, l.mode, a.query
FROM pg_locks l
JOIN pg_stat_activity a ON l.pid = a.pid;
```

Connection Management

```
# Check connections via psql
\conninfo

# Show all active connections
SELECT count(*) FROM pg_stat_activity;

# Kill a connection by PID
SELECT pg_terminate_backend(pid);
```

Full-Text Search

```
-- Basic full-text search
SELECT title, body
FROM articles
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'PostgreSQL & performance');

-- Create a tsvector column with GIN index
ALTER TABLE articles ADD COLUMN search_vector tsvector;
CREATE INDEX idx_articles_search ON articles USING GIN (search_vector);

-- Populate the search vector (trigger-based is preferred)
UPDATE articles SET search_vector =
    setweight(to_tsvector('english', coalesce(title, '')), 'A') ||
    setweight(to_tsvector('english', coalesce(body, '')), 'B');

-- Weighted search: title matches rank higher than body matches
SELECT title, ts_rank(search_vector, query) AS rank
FROM articles, to_tsquery('english', 'database') query
WHERE search_vector @@ query
ORDER BY rank DESC;

-- Phrase search
SELECT title FROM articles
WHERE to_tsvector('english', body) @@ phraseto_tsquery('english', 'performance tuning');

-- Headline function (generates excerpts with highlighted matches)
SELECT ts_headline('english', body, to_tsquery('english', 'PostgreSQL'))
FROM articles WHERE id = 1;

-- Create a trigger to auto-update the search vector
CREATE FUNCTION articles_search_vector_update() RETURNS trigger AS $$
BEGIN
    NEW.search_vector :=
        setweight(to_tsvector('english', coalesce(NEW.title, '')), 'A') ||
        setweight(to_tsvector('english', coalesce(NEW.body, '')), 'B');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

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

Extensions

```
-- PostGIS — spatial data
CREATE EXTENSION IF NOT EXISTS postgis;

CREATE TABLE locations (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    geom GEOMETRY(Point, 4326)  -- WGS84 (GPS coordinates)
);

INSERT INTO locations (name, geom)
VALUES ('Berlin', ST_SetSRID(ST_MakePoint(13.4050, 52.5200), 4326));
```

```
-- Find locations within 10km of a point
SELECT name, ST_Distance(geom, ST_SetSRID(ST_MakePoint(13.4050, 52.5200), 4326)) AS distance
FROM locations
WHERE ST_DWithin(geom, ST_SetSRID(ST_MakePoint(13.4050, 52.5200), 4326), 10000)
ORDER BY distance;

-- pg_stat_statements — query statistics
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

-- Top 10 slowest queries
SELECT query, calls, total_exec_time, mean_exec_time, rows
FROM pg_stat_statements
ORDER BY mean_exec_time DESC
LIMIT 10;

-- pgcrypto — cryptographic functions
CREATE EXTENSION IF NOT EXISTS pgcrypto;

-- Generate UUID v4
SELECT gen_random_uuid();

-- Hash passwords
SELECT crypt('my_password', gen_salt('bf'));

-- Verify password
SELECT (crypt('my_password', stored_hash) = stored_hash) AS password_valid;

-- uuid-ossp — UUID generation
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
SELECT uuid_generate_v4();   -- random UUID
SELECT uuid_generate_v1();   -- MAC-address + timestamp

-- hstore — key-value store
CREATE EXTENSION IF NOT EXISTS hstore;
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT,
    attrs hstore  -- flexible key-value attributes
);
INSERT INTO products (name, attrs) VALUES ('Widget', '"color"=>"red", "weight"=>"1.5"');
SELECT * FROM products WHERE attrs ? 'color';
SELECT attrs -> 'weight' FROM products WHERE name = 'Widget';

-- citext — case-insensitive text
CREATE EXTENSION IF NOT EXISTS citext;
CREATE TABLE accounts (
    username CITEXT PRIMARY KEY,
    email CITEXT UNIQUE
);
INSERT INTO accounts VALUES ('JohnDoe', 'John@Example.Com');
SELECT * FROM accounts WHERE username = 'johndoe';  -- matches!
```

Streaming Replication

```
# --- Primary Server Configuration ---
# postgresql.conf
wal_level = replica
```

```
max_wal_senders = 5
wal_keep_size = 1GB
hot_standby = on
archive_mode = on
archive_command = 'cp %p /var/lib/postgresql/wal_archive/%f'

# pg_hba.conf — allow replication connections
host    replication     replicator      10.0.0.0/24      scram-sha-256

# Create replication user
CREATE USER replicator WITH REPLICATION ENCRYPTED PASSWORD 'strong_replication_password';
```

```
# --- Standby Server Setup ---
# Stop PostgreSQL on the standby
sudo systemctl stop postgresql

# Use pg_basebackup to clone the primary
sudo -u postgres pg_basebackup \
    -h primary_host -D /var/lib/postgresql/15/main \
    -U replicator -P -v -R -X stream \
    -C -S standby_slot

# -R creates standby.signal and sets primary_conninfo in postgresql.auto.conf
# -C -S creates a replication slot on the primary

# Start the standby
sudo systemctl start postgresql
```

```
-- Verify replication status on the PRIMARY
SELECT client_addr, state, sent_lsn, write_lsn, flush_lsn, replay_lsn,
       pg_wal_lsn_diff(sent_lsn, replay_lsn) AS lag_bytes
FROM pg_stat_replication;

-- Check replication slot health
SELECT slot_name, active, restart_lsn,
       pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn) AS lag_bytes
FROM pg_replication_slots;

-- Failover: promote the standby to primary
SELECT pg_promote();
-- Or via command line:
# pg_ctl promote -D /var/lib/postgresql/15/main

-- After failover, rebuild old primary as new standby
```

Logical Replication

```
-- Publisher: create a publication
CREATE PUBLICATION my_publication FOR TABLE users, orders, products;
CREATE PUBLICATION all_tables FOR ALL TABLES;

-- Subscriber: create a subscription
CREATE SUBSCRIPTION my_subscription
    CONNECTION 'host=primary_host dbname=mydb user=replicator password=secret'
    PUBLICATION my_publication
    WITH (copy_data = true, create_slot = true);
```

```
-- Manage subscriptions
ALTER SUBSCRIPTION my_subscription DISABLE;
ALTER SUBSCRIPTION my_subscription ENABLE;
ALTER SUBSCRIPTION my_subscription REFRESH PUBLICATION;

-- View subscription status
SELECT subname, subrelid, substate FROM pg_subscription;

-- Add a table to an existing publication
ALTER PUBLICATION my_publication ADD TABLE new_table;

-- Use case: partial replication (only certain tables to a data warehouse)
CREATE PUBLICATION analytics_pub FOR TABLE orders, order_items, products;
```

Views & Materialized Views

```
-- Regular view (virtual table — query runs every time)
CREATE VIEW active_users AS
SELECT id, username, email, last_login
FROM users
WHERE last_login > CURRENT_TIMESTAMP - INTERVAL '30 days';

-- Materialized view (query result stored on disk)
CREATE MATERIALIZED VIEW monthly_sales AS
SELECT DATE_TRUNC('month', order_date) AS month,
       COUNT(*) AS order_count,
       SUM(amount) AS total_revenue
FROM orders
GROUP BY DATE_TRUNC('month', order_date)
WITH DATA;

-- Create an index on the materialized view for fast lookups
CREATE INDEX idx_monthly_sales_month ON monthly_sales (month);

-- Refresh materialized view (locks the table during refresh)
REFRESH MATERIALIZED VIEW monthly_sales;

-- Concurrent refresh (does NOT lock — available for reads)
REFRESH MATERIALIZED VIEW CONCURRENTLY monthly_sales;

-- Drop materialized view
DROP MATERIALIZED VIEW IF EXISTS monthly_sales;

-- Use case: pre-computed dashboard data
CREATE MATERIALIZED VIEW dashboard_metrics AS
SELECT
    (SELECT COUNT(*) FROM users WHERE created_at > NOW() - INTERVAL '7 days') AS new_users,
    (SELECT COUNT(*) FROM orders WHERE status = 'pending') AS pending_orders,
    (SELECT SUM(amount) FROM payments WHERE created_at > NOW() - INTERVAL '1 day') AS daily_r
```

Stored Procedures & Functions

```
-- Simple function returning a single value
CREATE OR REPLACE FUNCTION get_order_total(p_order_id INT)
RETURNS NUMERIC AS $$
```

```
DECLARE
    v_total NUMERIC;
BEGIN
    SELECT SUM(quantity * unit_price)
    INTO v_total
    FROM order_items
    WHERE order_id = p_order_id;
    RETURN COALESCE(v_total, 0);
END;
$$ LANGUAGE plpgsql;

-- Function returning a table (set-returning function)
CREATE OR REPLACE FUNCTION user_order_summary(p_user_id INT)
RETURNS TABLE (order_id INT, total NUMERIC, item_count INT) AS $$
BEGIN
    RETURN QUERY
    SELECT o.id, SUM(oi.quantity * oi.unit_price), COUNT(oi.id)
    FROM orders o
    JOIN order_items oi ON o.id = oi.order_id
    WHERE o.user_id = p_user_id
    GROUP BY o.id
    ORDER BY o.id DESC;
END;
$$ LANGUAGE plpgsql;

-- Function with OUT parameters
CREATE OR REPLACE FUNCTION calculate_discount(
    p_amount NUMERIC,
    p_tier TEXT,
    OUT discounted_amount NUMERIC,
    OUT discount_pct NUMERIC
) AS $$
BEGIN
    discount_pct := CASE p_tier
        WHEN 'gold'   THEN 0.15
        WHEN 'silver' THEN 0.10
        WHEN 'bronze' THEN 0.05
        ELSE 0
    END;
    discounted_amount := p_amount * (1 - discount_pct);
END;
$$ LANGUAGE plpgsql;

-- SECURITY DEFINER: runs with privileges of the function owner
CREATE OR REPLACE FUNCTION admin_reset_password(p_user_id INT, p_new_hash TEXT)
RETURNS VOID
SECURITY DEFINER
SET search_path = public
LANGUAGE plpgsql AS $$
BEGIN
    UPDATE users SET password_hash = p_new_hash WHERE id = p_user_id;
END;
$$;

-- Usage
SELECT get_order_total(42);
SELECT * FROM user_order_summary(7);
SELECT * FROM calculate_discount(100.00, 'gold');
```

Triggers

```
-- Trigger function: audit log for all changes to a table
CREATE TABLE audit_log (
    id BIGSERIAL PRIMARY KEY,
    table_name TEXT NOT NULL,
    operation TEXT NOT NULL,
    old_data JSONB,
    new_data JSONB,
    changed_by TEXT DEFAULT current_user,
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION audit_trigger_func()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'DELETE' THEN
        INSERT INTO audit_log (table_name, operation, old_data)
        VALUES (TG_TABLE_NAME, TG_OP, to_jsonb(OLD));
        RETURN OLD;
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO audit_log (table_name, operation, old_data, new_data)
        VALUES (TG_TABLE_NAME, TG_OP, to_jsonb(OLD), to_jsonb(NEW));
        RETURN NEW;
    ELSIF TG_OP = 'INSERT' THEN
        INSERT INTO audit_log (table_name, operation, new_data)
        VALUES (TG_TABLE_NAME, TG_OP, to_jsonb(NEW));
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Attach the trigger to a table
CREATE TRIGGER users_audit_trigger
    AFTER INSERT OR UPDATE OR DELETE ON users
    FOR EACH ROW EXECUTE FUNCTION audit_trigger_func();

-- BEFORE trigger: enforce business rules
CREATE OR REPLACE FUNCTION prevent_negative_balance()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.balance < 0 THEN
        RAISE EXCEPTION 'Account balance cannot be negative (attempted: %)', NEW.balance;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_balance
    BEFORE INSERT OR UPDATE ON accounts
    FOR EACH ROW EXECUTE FUNCTION prevent_negative_balance();

-- Trigger timing options: BEFORE, AFTER, INSTEAD OF
-- Row-level: FOR EACH ROW (runs per row)
-- Statement-level: FOR EACH STATEMENT (runs once per statement)
```

Advanced Data Types

```
-- ARRAY
CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    title TEXT,
    tags TEXT[]
);
INSERT INTO posts (title, tags) VALUES ('Intro to SQL', ARRAY['database', 'sql', 'beginner'])
SELECT * FROM posts WHERE 'sql' = ANY(tags);
SELECT unnest(tags) FROM posts WHERE id = 1;

-- UUID
CREATE TABLE sessions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id INT NOT NULL,
    expires_at TIMESTAMP NOT NULL
);
INSERT INTO sessions (user_id, expires_at) VALUES (42, NOW() + INTERVAL '24 hours');

-- MACADDR and INET/CIDR
CREATE TABLE network_devices (
    id SERIAL PRIMARY KEY,
    hostname TEXT,
    ip_address INET,
    mac_address MACADDR
);
INSERT INTO network_devices (hostname, ip_address, mac_address)
VALUES ('web-01', '192.168.1.10', '00:1a:2b:3c:4d:5e');
SELECT * FROM network_devices WHERE ip_address << '192.168.1.0/24';  -- within subnet

-- ENUM
CREATE TYPE order_status AS ENUM ('pending', 'processing', 'shipped', 'delivered', 'cancelled
ALTER TABLE orders ADD COLUMN status order_status DEFAULT 'pending';
SELECT * FROM orders WHERE status = 'shipped';

-- DOMAIN (custom type with constraints)
CREATE DOMAIN email_address TEXT
CHECK (VALUE ~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$');

CREATE TABLE contacts (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email email_address NOT NULL
);

-- Composite type
CREATE TYPE address AS (
    street TEXT,
    city TEXT,
    zip_code TEXT,
    country TEXT DEFAULT 'US'
);

CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name TEXT,
    billing_addr address
);
INSERT INTO customers (name, billing_addr)
```

```
VALUES ('Acme Corp', ROW('123 Main St', 'Springfield', '62701', 'US'));
SELECT (billing_addr).city FROM customers WHERE name = 'Acme Corp';
```

Constraints Deep Dive

```
-- CHECK constraint
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    price NUMERIC(10,2) CHECK (price >= 0),
    discount_pct INT CHECK (discount_pct BETWEEN 0 AND 100),
    CHECK (price > discount_pct * price / 100)  -- table-level constraint
);

-- UNIQUE constraint (single and composite)
ALTER TABLE users ADD CONSTRAINT users_email_unique UNIQUE (email);
CREATE TABLE enrollments (
    student_id INT,
    course_id INT,
    UNIQUE (student_id, course_id)
);

-- EXCLUDE constraint (prevent overlapping ranges)
CREATE TABLE room_bookings (
    room_id INT NOT NULL,
    during TSTZRANGE NOT NULL,
    EXCLUDE USING gist (
        room_id WITH =,
        during WITH &&
    )
);

-- DEFERRABLE constraints (check at end of transaction)
CREATE TABLE accounts (
    id SERIAL PRIMARY KEY,
    balance NUMERIC CHECK (balance >= 0) DEFERRABLE INITIALLY DEFERRED
);

BEGIN;
UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;
COMMIT;  -- constraints checked here, not at each UPDATE

-- Foreign key actions
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(id)
        ON DELETE CASCADE          -- delete orders when user is deleted
        ON UPDATE CASCADE,
    status TEXT DEFAULT 'pending'
);

CREATE TABLE logs (
    id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(id)
        ON DELETE SET NULL         -- keep logs, clear user reference
```

```
        ON UPDATE SET NULL
);
```

COPY & Bulk Data

```
# Export to CSV
psql -c "COPY (SELECT * FROM users) TO STDOUT WITH CSV HEADER" > users.csv

# Import from CSV
psql -c "COPY users FROM STDIN WITH CSV HEADER" < users.csv

# Export with custom delimiter and NULL handling
psql -c "COPY products TO STDOUT WITH (FORMAT csv, DELIMITER '|', NULL 'NULL', HEADER true)"

# Import specific columns
psql -c "COPY users(username, email, created_at) FROM STDIN WITH CSV" < new_users.csv

# Binary format (faster for large datasets)
psql -c "COPY large_table TO STDOUT WITH BINARY" > large_table.bin
psql -c "COPY large_table FROM STDIN WITH BINARY" < large_table.bin
```

```
-- SQL COPY (server-side, requires superuser)
COPY users TO '/tmp/users.csv' WITH CSV HEADER;
COPY users FROM '/tmp/users.csv' WITH CSV HEADER;

-- \copy (client-side, no superuser needed)
\copy users TO 'users.csv' CSV HEADER
\copy users FROM 'users.csv' CSV HEADER
```

Connection Pooling (PgBouncer)

```
# Install PgBouncer
sudo apt install pgbouncer

# /etc/pgbouncer/pgbouncer.ini
[databases]
mydb = host=127.0.0.1 port=5432 dbname=mydb

[pgbouncer]
listen_addr = 127.0.0.1
listen_port = 6432
auth_type = scram-sha-256
auth_file = /etc/pgbouncer/userlist.txt
pool_mode = transaction
max_client_conn = 1000
default_pool_size = 20
reserve_pool_size = 5
reserve_pool_timeout = 3
server_idle_timeout = 600
```

```
# Add users to userlist.txt
echo '"mydb_user" "password_hash"' | sudo tee -a /etc/pgbouncer/userlist.txt

# Get password hash from PostgreSQL
```

```
psql -c "SELECT username || ':' || passwd FROM pg_shadow WHERE username = 'mydb_user'"

# Pool modes comparison
# session    — connection reused only after client disconnects (safest, most connections)
# transaction — connection returned after transaction ends (recommended for most apps)
# statement  — connection returned after each statement (not compatible with prepared stateme

# Reload configuration
sudo systemctl reload pgbouncer
sudo systemctl restart pgbouncer

# Connect through PgBouncer (same psql, different port)
psql -h 127.0.0.1 -p 6432 -U mydb_user -d mydb

# Monitor
psql -h 127.0.0.1 -p 6432 -d pgbouncer -c "SHOW POOLS;"
psql -h 127.0.0.1 -p 6432 -d pgbouncer -c "SHOW STATS;"
```

pg_hba.conf & Authentication

```
# /etc/postgresql/15/main/pg_hba.conf

# TYPE  DATABASE   USER      ADDRESS             METHOD

# Local connections
local   all        all                           peer

# IPv4 local connections (password auth)
host    all        all       127.0.0.1/32        scram-sha-256

# IPv6 local connections
host    all        all       ::1/128             scram-sha-256

# Application servers (password auth)
host    mydb       app_user  10.0.1.0/24         scram-sha-256

# Replication connections
host    replication replicator 10.0.0.0/24       scram-sha-256

# Trust local development (insecure!)
# host    all        all       192.168.1.0/24     trust

# Certificate-based authentication
# hostssl all        all       0.0.0.0/0          cert

# Reload pg_hba.conf without restart
SELECT pg_reload_conf();
```

WAL & Point-in-Time Recovery

```
# postgresql.conf — enable WAL archiving
wal_level = replica
archive_mode = on
archive_command = 'cp %p /var/lib/postgresql/wal_archive/%f'
archive_timeout = 300  # force archive every 5 minutes even if WAL is not full
```

```
# Enable WAL compression (PostgreSQL 14+)
wal_compression = lz4

# --- Recovery (PITR) ---
# 1. Stop PostgreSQL
sudo systemctl stop postgresql

# 2. Clear the data directory
rm -rf /var/lib/postgresql/15/main/*

# 3. Restore from base backup
sudo -u postgres pg_restore -D /var/lib/postgresql/15/main /backups/base_backup

# 4. Create recovery signal file
sudo -u postgres touch /var/lib/postgresql/15/main/recovery.signal

# 5. Configure recovery target in postgresql.auto.conf
# restore_command = 'cp /var/lib/postgresql/wal_archive/%f %p'
# recovery_target_time = '2026-04-20 14:30:00'
# recovery_target_action = 'promote'

# 6. Start PostgreSQL (recovery begins automatically)
sudo systemctl start postgresql
```

```
# pg_restore options
pg_restore -d mydb -U username backup.dump          # restore from custom format
pg_restore -d mydb -U username -c backup.dump       # clean (drop) existing objects first
pg_restore -d mydb -U username -j 4 backup.dump     # parallel restore with 4 jobs
pg_restore -l backup.dump > toc.txt                 # list contents
pg_restore -d mydb -L toc.txt backup.dump           # restore using table of contents (select
```

Tablespaces

```
-- Create a tablespace on a specific directory
CREATE TABLESPACE fast_storage LOCATION '/mnt/nvme/postgresql';

-- Create a tablespace for indexes
CREATE TABLESPACE index_storage LOCATION '/mnt/ssd/postgresql_indexes';

-- Create a table in a specific tablespace
CREATE TABLE large_analytics (
    id BIGSERIAL,
    event_data JSONB,
    created_at TIMESTAMP DEFAULT NOW()
) TABLESPACE fast_storage;

-- Move an existing table to a different tablespace
ALTER TABLE large_analytics SET TABLESPACE fast_storage;

-- Move all indexes of a table to a different tablespace
ALTER INDEX idx_analytics_created SET TABLESPACE index_storage;

-- Set default tablespace for new objects
SET default_tablespace = fast_storage;

-- View tablespaces
SELECT spcname, pg_tablespace_location(oid) AS location
```

```
FROM pg_tablespace;

-- Set default temp tablespace
SET temp_tablespaces = 'fast_storage';
```

Monitoring Queries

```
-- pg_stat_activity deep dive
SELECT pid, usename, datname, state, wait_event_type, wait_event,
       query_start, state_change, EXTRACT(EPOCH FROM (NOW() - query_start)) AS duration_s,
       LEFT(query, 80) AS query_preview
FROM pg_stat_activity
WHERE state != 'idle'
ORDER BY duration_s DESC;

-- Top queries by total execution time (requires pg_stat_statements)
SELECT query, calls, total_exec_time AS total_ms,
       mean_exec_time AS avg_ms, rows
FROM pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 20;

-- Table and index sizes
SELECT relname AS table_name,
       pg_size_pretty(pg_table_size(relid)) AS table_size,
       pg_size_pretty(pg_indexes_size(relid)) AS index_size,
       pg_size_pretty(pg_total_relation_size(relid)) AS total_size,
       n_dead_tup AS dead_tuples,
       n_live_tup AS live_tuples,
       CASE WHEN n_dead_tup > 0.1 * n_live_tup THEN 'NEEDS VACUUM' ELSE 'OK' END AS vacuum_st
FROM pg_stat_user_tables
ORDER BY pg_total_relation_size(relid) DESC;

-- Index usage (find unused indexes)
SELECT schemaname, relname AS table_name, indexrelname AS index_name,
       idx_scan AS index_scans, pg_size_pretty(pg_relation_size(indexrelid)) AS size
FROM pg_stat_user_indexes
WHERE idx_scan = 0 AND indexrelname NOT LIKE '%_pkey'
ORDER BY pg_relation_size(indexrelid) DESC;

-- Lock wait analysis
SELECT blocked.pid AS blocked_pid,
       blocked.query AS blocked_query,
       blocking.pid AS blocking_pid,
       blocking.query AS blocking_query,
       EXTRACT(EPOCH FROM (NOW() - blocked.query_start)) AS blocked_duration_s
FROM pg_stat_activity blocked
JOIN pg_locks bl ON bl.pid = blocked.pid
JOIN pg_locks kl ON kl.locktype = bl.locktype
    AND kl.database IS NOT DISTINCT FROM bl.database
    AND kl.relation IS NOT DISTINCT FROM bl.relation
    AND kl.page IS NOT DISTINCT FROM bl.page
    AND kl.tuple IS NOT DISTINCT FROM bl.tuple
    AND kl.pid != bl.pid
JOIN pg_stat_activity blocking ON blocking.pid = kl.pid
WHERE NOT bl.granted;

-- Bloat detection (tables with significant wasted space)
SELECT schemaname, tablename,
```

```
       pg_size_pretty(pg_total_relation_size(schemaname || '.' || tablename)) AS total_size,
       pg_stat_get_dead_tuples(c.oid) AS dead_tuples
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
JOIN pg_stat_user_tables s ON s.relid = c.oid
WHERE c.relkind = 'r'
ORDER BY pg_stat_get_dead_tuples(c.oid) DESC;

-- Database-wide summary
SELECT pg_database.datname,
       pg_size_pretty(pg_database_size(pg_database.datname)) AS size,
       pg_stat_activity.numbackends AS active_connections
FROM pg_database
LEFT JOIN pg_stat_activity ON pg_database.datname = pg_stat_activity.datname
GROUP BY pg_database.datname
ORDER BY pg_database_size(pg_database.datname) DESC;
```

Best Practices

1. **Always use `EXPLAIN ANALYZE`** for critical queries to verify performance.
2. **Prefer `JSONB` over `JSON`** for better performance and indexing support.
3. **Handle transactions carefully** to avoid deadlocks in high-concurrency environments.
4. **Implement partitioning** for tables exceeding hundreds of millions of rows.
5. **Use `VACUUM ANALYZE`** after bulk data loads to refresh statistics.
6. **Implement Connection Pooling** (e.g., PgBouncer) to prevent connection exhaustion.
7. **Use `LOWER()` and functional indexes** to optimize case-insensitive searches.
8. **Leverage `CTE`s** for readable and complex hierarchical queries.
9. **Design for MVCC**: Understand that updates/deletes create "dead" versions of rows.
10. **Always use explicit transaction boundaries** for multi-step operations.
11. **Use materialized views** for expensive queries that don't need real-time data.
12. **Enable `pg_stat_statements`** in production to track query performance over time.
13. **Use `scram-sha-256`** authentication instead of `md5` for all connections.
14. **Archive WAL** and test PITR recovery regularly — backups are only as good as your restore.
15. **Monitor replication lag** and set up alerts before it impacts read-after-write consistency.

Next Steps

- Bash CLI Tools Cheat Sheet — Unix command-line utilities
- Docker CLI Cheat Sheet — Containerize PostgreSQL deployments
- Systemd Cheat Sheet — Service management for PostgreSQL
- Redis Cheat Sheet — In-memory data store
- Git CLI Cheat Sheet — Version control commands
- Nginx Cheat Sheet — Reverse proxy for PostgreSQL
- Terraform Cheat Sheet — Infrastructure as Code