Basic PostgreSQL Interview Questions & Answers

PostgreSQL Fundamentals

Q1: What is PostgreSQL and what are its key features?

Answer: PostgreSQL is an open-source, object-relational database management system (ORDBMS) that emphasizes extensibility and SQL compliance.

Key Features:

- **ACID Compliance**: Ensures data integrity through Atomicity, Consistency, Isolation, and Durability
- **Multi-Version Concurrency Control (MVCC)**: Allows multiple transactions without blocking
- **Extensibility**: Custom data types, functions, operators, and indexing methods
- **Advanced Data Types**: JSON/JSONB, Arrays, UUID, Geographic data types
- **Full-Text Search**: Built-in text search capabilities
- **Foreign Data Wrappers**: Connect to external data sources
- **Stored Procedures**: Support for multiple programming languages (PL/pgSQL, Python, etc.)
- **Replication**: Built-in streaming replication and logical replication

Q2: What is the difference between PostgreSQL and MySQL?

Answer:

Feature	PostgreSQL	MySQL
Architecture	Object-relational	Relational
ACID Compliance	Full ACID compliance	ACID with InnoDB engine
SQL Standards	More SQL standard compliant	Less strict SQL compliance
Data Types	Rich data types (JSON, Arrays, UUID)	Basic data types
Concurrency	MVCC (better for read-heavy)	Locking (better for write-heavy)
Extensibility	Highly extensible	Limited extensibility
Performance	Better for complex queries	Better for simple read/write operations
Replication	Built-in streaming replication	Master-slave replication

Q3: Explain PostgreSQL architecture.

Answer: PostgreSQL uses a **client-server architecture** with the following components:

1. Postmaster Process (Main Server Process):

- Manages client connections
- Spawns backend processes for each connection
- Handles authentication and authorization

2. Backend Processes:

- One per client connection
- Executes SQL commands
- Manages transaction isolation

3. Shared Memory:

- **Shared Buffers**: Cache for data pages

- **WAL Buffers**: Write-Ahead Logging buffers
- **Lock Tables**: Manages concurrent access

4. Background Processes:

- **Checkpointer**: Writes dirty pages to disk
- **WAL Writer**: Writes WAL buffers to disk
- **Autovacuum**: Automatic cleanup and statistics update
- **Background Writer**: Writes dirty buffers to disk
- **Archiver**: Archives WAL files for backup

5. Storage:

- Data files stored in tablespaces
- Transaction logs (WAL files)
- Configuration files

Data Types and Database Objects

Q4: What are the main data types in PostgreSQL?

Answer:

1. Numeric Types:

```sql
-- Integer types
SMALLINT       -- 2 bytes, -32,768 to 32,767
INTEGER (INT)  -- 4 bytes, -2,147,483,648 to 2,147,483,647
BIGINT         -- 8 bytes, large range

-- Decimal types
DECIMAL(p,s)   -- Exact decimal
NUMERIC(p,s)   -- Same as DECIMAL
REAL           -- 4 bytes floating point
DOUBLE PRECISION -- 8 bytes floating point

-- Auto-increment
SERIAL         -- Auto-incrementing integer
BIGSERIAL      -- Auto-incrementing bigint
```

2. Character Types:

```sql
CHAR(n)        -- Fixed-length character string
VARCHAR(n)     -- Variable-length character string
TEXT           -- Variable-length character string (unlimited)
```

3. Date/Time Types:

```sql
```

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

4. Boolean:

```
sql

BOOLEAN       -- TRUE, FALSE, NULL
```

5. Advanced Types:

```
sql

JSON          -- JSON data (stored as text)
JSONB         -- Binary JSON (faster operations)
UUID          -- Universally unique identifier
ARRAY         -- Array of any data type
HSTORE        -- Key-value pairs
```

Q5: What is the difference between CHAR, VARCHAR, and TEXT?

Answer:

Type	Storage	Use Case	Performance
CHAR(n)	Fixed-length, padded with spaces	Fixed-size data (codes, IDs)	Faster for fixed-length comparisons
VARCHAR(n)	Variable-length up to n characters	Variable data with known max length	Good balance of storage and performance
TEXT	Unlimited variable-length	Large text content, no size limit	Same performance as VARCHAR

```
sql

-- Examples
CREATE TABLE example (
    country_code CHAR(2),       -- Always 2 characters: 'US', 'IN'
    username VARCHAR(50),       -- Up to 50 characters
    description TEXT            -- Unlimited length
);
```

Key Points:

- CHAR pads with spaces: CHAR(5) storing 'hi' becomes 'hi '
- VARCHAR and TEXT have identical performance in PostgreSQL
- Use CHAR for fixed-length codes, VARCHAR when you know max length, TEXT for unlimited content

Q6: What are PostgreSQL constraints and their types?

Answer:

Constraints enforce rules on data to maintain data integrity:

1. PRIMARY KEY:

sql

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL
);
```

2. FOREIGN KEY:

sql

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INTEGER REFERENCES users(id),
  -- or with explicit constraint name
  customer_id INTEGER,
  CONSTRAINT fk_customer FOREIGN KEY (customer_id) REFERENCES users(id)
);
```

3. UNIQUE:

sql

```
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  sku VARCHAR(50) UNIQUE,
  name VARCHAR(255)
);
```

4. NOT NULL:

sql

```
CREATE TABLE employees (
  id SERIAL PRIMARY KEY,
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL,
  email VARCHAR(255) NOT NULL
);
```

5. CHECK:

sql

```
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  price DECIMAL(10,2) CHECK (price > 0),
  quantity INTEGER CHECK (quantity >= 0),
  status VARCHAR(20) CHECK (status IN ('active', 'inactive', 'discontinued'))
);
```

6. DEFAULT:

sql

```
CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20) DEFAULT 'draft'
);
```

Basic SQL Operations

Q7: How do you create, alter, and drop tables?

Answer:

Create Table:

```sql
-- Basic table creation
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    phone VARCHAR(20),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Table with foreign key
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    customer_id INTEGER REFERENCES customers(id),
    order_date DATE DEFAULT CURRENT_DATE,
    total_amount DECIMAL(10,2) CHECK (total_amount >= 0),
    status VARCHAR(20) DEFAULT 'pending'
);
```

Alter Table:

```sql
-- Add column
ALTER TABLE customers ADD COLUMN address TEXT;

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

-- Add constraint
ALTER TABLE customers ADD CONSTRAINT chk_email CHECK (email ~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]

-- Drop column
ALTER TABLE customers DROP COLUMN address;

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

-- Rename table
ALTER TABLE customers RENAME TO client;
```

Drop Table:

```sql
-- Drop single table
DROP TABLE IF EXISTS temp_table;

-- Drop table with dependencies (cascade)
DROP TABLE customers CASCADE;

-- Drop multiple tables
DROP TABLE table1, table2, table3;
```

Q8: Explain different types of JOINs with examples.

Answer:

Sample Tables:

```sql
-- Users table
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(255)
);

-- Orders table
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INTEGER,
    amount DECIMAL(10,2)
);

-- Sample data
INSERT INTO users VALUES (1, 'John', 'john@email.com'), (2, 'Jane', 'jane@email.com'), (3, 'Bob', 'bob@email.com');
INSERT INTO orders VALUES (1, 1, 100.00), (2, 1, 150.00), (3, 2, 200.00), (4, 4, 75.00);
```

1. INNER JOIN (Only matching records):

```sql
SELECT u.name, o.amount
FROM users u
INNER JOIN orders o ON u.id = o.user_id;

-- Result: John (100.00), John (150.00), Jane (200.00)
```

2. LEFT JOIN (All from left table):

```sql
SELECT u.name, o.amount
FROM users u
LEFT JOIN orders o ON u.id = o.user_id;

-- Result: John (100.00), John (150.00), Jane (200.00), Bob (NULL)
```

3. RIGHT JOIN (All from right table):

```
sql

SELECT u.name, o.amount
FROM users u
RIGHT JOIN orders o ON u.id = o.user_id;

-- Result: John (100.00), John (150.00), Jane (200.00), NULL (75.00)
```

4. FULL OUTER JOIN (All records from both tables):

```
sql

SELECT u.name, o.amount
FROM users u
FULL OUTER JOIN orders o ON u.id = o.user_id;

-- Result: John (100.00), John (150.00), Jane (200.00), Bob (NULL), NULL (75.00)
```

5. CROSS JOIN (Cartesian product):

```
sql

SELECT u.name, o.amount
FROM users u
CROSS JOIN orders o;

-- Result: Every user paired with every order (12 rows total)
```

Q9: What are aggregate functions and window functions?

Answer:

Aggregate Functions (operate on multiple rows, return single value):

```
sql

-- Common aggregate functions
SELECT
  COUNT(*) as total_orders,
  COUNT(DISTINCT user_id) as unique_customers,
  SUM(amount) as total_revenue,
  AVG(amount) as average_order,
  MIN(amount) as smallest_order,
  MAX(amount) as largest_order
FROM orders;

-- With GROUP BY
SELECT
  user_id,
  COUNT(*) as order_count,
  SUM(amount) as total_spent
FROM orders
GROUP BY user_id
HAVING SUM(amount) > 100;
```

Window Functions (operate on rows related to current row):

sql

```sql
-- ROW_NUMBER, RANK, DENSE_RANK
SELECT
  name,
  amount,
  ROW_NUMBER() OVER (ORDER BY amount DESC) as row_num,
  RANK() OVER (ORDER BY amount DESC) as rank,
  DENSE_RANK() OVER (ORDER BY amount DESC) as dense_rank
FROM orders o
JOIN users u ON o.user_id = u.id;

-- Partitioned window functions
SELECT
  user_id,
  amount,
  AVG(amount) OVER (PARTITION BY user_id) as user_avg,
  SUM(amount) OVER (PARTITION BY user_id ORDER BY id) as running_total
FROM orders;

-- LAG and LEAD
SELECT
  amount,
  LAG(amount, 1) OVER (ORDER BY id) as previous_order,
  LEAD(amount, 1) OVER (ORDER BY id) as next_order
FROM orders;
```

Indexes and Performance

Q10: What are indexes and why are they important?

Answer:

Indexes are database objects that improve query performance by creating shortcuts to data rows.

Benefits:

- **Faster SELECT queries**: Dramatically reduce query execution time
- **Efficient sorting**: ORDER BY operations use indexes
- **Unique constraints**: Automatically created for PRIMARY KEY and UNIQUE constraints
- **JOIN optimization**: Speed up table joins

Types of Indexes:

sql

```
-- B-tree index (default)
CREATE INDEX idx_users_email ON users(email);

-- Unique index
CREATE UNIQUE INDEX idx_users_username ON users(username);

-- Composite index
CREATE INDEX idx_orders_user_date ON orders(user_id, order_date);

-- Partial index
CREATE INDEX idx_active_users ON users(email) WHERE status = 'active';

-- Functional index
CREATE INDEX idx_users_lower_email ON users(LOWER(email));
```

When to Use Indexes:

- Columns frequently used in WHERE clauses
- Columns used in JOIN conditions
- Columns used in ORDER BY
- Foreign key columns

When NOT to Use Indexes:

- Small tables (overhead not worth it)
- Columns that change frequently
- Tables with heavy INSERT/UPDATE/DELETE operations

Q11: How do you analyze query performance?

Answer:

Using EXPLAIN:

```sql
sql

-- Show query plan without execution
EXPLAIN SELECT * FROM users WHERE email = 'john@email.com';

-- Show actual execution statistics
EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'john@email.com';

-- Detailed output with timing and buffers
EXPLAIN (ANALYZE, BUFFERS, TIMING)
SELECT u.name, COUNT(o.id) as order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id, u.name;
```

Reading EXPLAIN Output:

- **Cost**: startup cost..total cost (lower is better)
- **Rows**: estimated number of rows
- **Width**: average row size in bytes
- **Actual time**: real execution time (with ANALYZE)

- **Loops**: how many times the node was executed

Common Operations:

- **Seq Scan**: Full table scan (slow for large tables)
- **Index Scan**: Using index (fast)
- **Index Only Scan**: Data from index only (fastest)
- **Nested Loop**: Join algorithm for small datasets
- **Hash Join**: Join algorithm for larger datasets

Transactions and Concurrency

Q12: What are transactions and ACID properties?

Answer:

Transaction: A sequence of database operations that are executed as a single unit of work.

```
sql

-- Basic transaction
BEGIN;
  INSERT INTO users (name, email) VALUES ('Alice', 'alice@email.com');
  INSERT INTO orders (user_id, amount) VALUES (1, 250.00);
  UPDATE users SET last_order_date = CURRENT_DATE WHERE id = 1;
COMMIT;

-- Transaction with error handling
BEGIN;
  UPDATE accounts SET balance = balance - 100 WHERE id = 1;
  UPDATE accounts SET balance = balance + 100 WHERE id = 2;
  -- If any error occurs, rollback
ROLLBACK; -- or COMMIT if successful
```

ACID Properties:

1. Atomicity: All operations succeed or all fail

```
sql

-- Either both updates happen, or neither happens
BEGIN;
  UPDATE account SET balance = balance - 100 WHERE id = 1;
  UPDATE account SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

2. Consistency: Database remains in valid state

```
sql

-- Constraints ensure consistency
ALTER TABLE accounts ADD CONSTRAINT chk_positive_balance
CHECK (balance >= 0);
```

3. Isolation: Transactions don't interfere with each other

```
sql
```

```
-- Set isolation level
BEGIN ISOLATION LEVEL READ COMMITTED;
-- or REPEATABLE READ, SERIALIZABLE
```

4. Durability: Committed changes persist even after system failure

- Implemented through Write-Ahead Logging (WAL)
- Changes written to disk before transaction commits

Q13: What are isolation levels in PostgreSQL?

Answer:

PostgreSQL supports 4 isolation levels:

1. READ UNCOMMITTED (Lowest isolation):

```
sql

BEGIN ISOLATION LEVEL READ UNCOMMITTED;
-- Can read uncommitted changes from other transactions
-- Allows: Dirty reads, non-repeatable reads, phantom reads
```

2. READ COMMITTED (Default):

```
sql

BEGIN ISOLATION LEVEL READ COMMITTED;
-- Only reads committed data
-- Allows: Non-repeatable reads, phantom reads
-- Most commonly used level
```

3. REPEATABLE READ:

```
sql

BEGIN ISOLATION LEVEL REPEATABLE READ;
-- Same data read multiple times within transaction
-- Allows: Phantom reads
-- Prevents: Dirty reads, non-repeatable reads
```

4. SERIALIZABLE (Highest isolation):

```
sql

BEGIN ISOLATION LEVEL SERIALIZABLE;
-- Complete isolation, as if transactions run serially
-- Prevents: All phenomena but may cause more deadlocks
```

Example of Isolation Issues:

```
sql
```

```
-- Session 1
BEGIN;
UPDATE products SET price = 15.00 WHERE id = 1;
-- Don't commit yet

-- Session 2 (READ COMMITTED won't see the change)
SELECT price FROM products WHERE id = 1; -- Still shows old price

-- Session 1
COMMIT; -- Now Session 2 will see the new price
```

Functions and Stored Procedures

Q14: How do you create functions in PostgreSQL?

Answer:

Basic Function:

```
sql

-- Simple function
CREATE OR REPLACE FUNCTION calculate_tax(amount DECIMAL)
RETURNS DECIMAL AS $$
BEGIN
    RETURN amount * 0.08; -- 8% tax
END;
$$ LANGUAGE plpgsql;

-- Usage
SELECT calculate_tax(100.00); -- Returns 8.00
```

Function with Multiple Parameters:

```
sql

CREATE OR REPLACE FUNCTION get_full_name(first_name TEXT, last_name TEXT)
RETURNS TEXT AS $$
BEGIN
    RETURN first_name || ' ' || last_name;
END;
$$ LANGUAGE plpgsql;

-- Usage
SELECT get_full_name('John', 'Doe'); -- Returns 'John Doe'
```

Function Returning Table:

```
sql
```

```
CREATE OR REPLACE FUNCTION get_user_orders(user_id INTEGER)
RETURNS TABLE(order_id INTEGER, amount DECIMAL, order_date DATE) AS $$
BEGIN
    RETURN QUERY
    SELECT o.id, o.amount, o.order_date
    FROM orders o
    WHERE o.user_id = get_user_orders.user_id;
END;
$$ LANGUAGE plpgsql;

-- Usage
SELECT * FROM get_user_orders(1);
```

Function with Conditional Logic:

sql

```
CREATE OR REPLACE FUNCTION get_discount_rate(total_amount DECIMAL)
RETURNS DECIMAL AS $$
BEGIN
    IF total_amount >= 1000 THEN
        RETURN 0.10; -- 10% discount
    ELSIF total_amount >= 500 THEN
        RETURN 0.05; -- 5% discount
    ELSE
        RETURN 0.00; -- No discount
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Q15: What are triggers and how do you use them?

Answer:

Triggers are functions that automatically execute in response to database events.

Types of Triggers:

- **BEFORE**: Execute before the triggering event
- **AFTER**: Execute after the triggering event
- **INSTEAD OF**: Replace the triggering event (views only)

Trigger Function:

sql

```
-- Create trigger function
CREATE OR REPLACE FUNCTION update_modified_time()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Create trigger
CREATE TRIGGER users_update_trigger
    BEFORE UPDATE ON users
    FOR EACH ROW
    EXECUTE FUNCTION update_modified_time();
```

Audit Trail Trigger:

```
sql

-- Audit table
CREATE TABLE user_audit (
    id SERIAL PRIMARY KEY,
    user_id INTEGER,
    action VARCHAR(10),
    old_values JSONB,
    new_values JSONB,
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Audit trigger function
CREATE OR REPLACE FUNCTION audit_user_changes()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO user_audit (user_id, action, new_values)
        VALUES (NEW.id, 'INSERT', row_to_json(NEW));
        RETURN NEW;
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO user_audit (user_id, action, old_values, new_values)
        VALUES (NEW.id, 'UPDATE', row_to_json(OLD), row_to_json(NEW));
        RETURN NEW;
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO user_audit (user_id, action, old_values)
        VALUES (OLD.id, 'DELETE', row_to_json(OLD));
        RETURN OLD;
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Create triggers for all operations
CREATE TRIGGER user_audit_trigger
    AFTER INSERT OR UPDATE OR DELETE ON users
    FOR EACH ROW
    EXECUTE FUNCTION audit_user_changes();
```

Administration and Maintenance

Q16: What is VACUUM and why is it important?

Answer:

VACUUM is a maintenance operation that reclaims storage and updates statistics.

Why VACUUM is Needed:

- PostgreSQL uses MVCC (Multi-Version Concurrency Control)
- UPDATE and DELETE operations don't immediately remove old row versions
- Dead tuples accumulate over time, causing table bloat
- Statistics become outdated, affecting query performance

Types of VACUUM:

1. VACUUM (Standard):

```sql
-- Vacuum specific table
VACUUM users;

-- Vacuum all tables in database
VACUUM;

-- Vacuum with verbose output
VACUUM VERBOSE users;
```

2. VACUUM FULL (Aggressive):

```sql
-- Reclaims all dead space but locks table
VACUUM FULL users;
-- Warning: This locks the table and can take long time
```

3. VACUUM ANALYZE:

```sql
-- Vacuum and update statistics
VACUUM ANALYZE users;

-- Just update statistics
ANALYZE users;
```

Autovacuum (Automatic Maintenance):

```sql
-- Check autovacuum settings
SHOW autovacuum;

-- Configure autovacuum for specific table
ALTER TABLE users SET (
    autovacuum_vacuum_threshold = 100,
    autovacuum_analyze_threshold = 50
);
```

Monitoring VACUUM:

```sql
-- Check last vacuum/analyze times
SELECT
    relname,
    last_vacuum,
    last_autovacuum,
    last_analyze,
    last_autoanalyze
FROM pg_stat_user_tables;

-- Check table bloat
SELECT
    schemaname,
    tablename,
    n_dead_tup,
    n_live_tup,
    round(100.0 * n_dead_tup / (n_live_tup + n_dead_tup), 2) as dead_ratio
FROM pg_stat_user_tables
WHERE n_live_tup > 0;
```

Q17: How do you backup and restore PostgreSQL databases?

Answer:

Backup Methods:

1. pg_dump (Logical Backup):

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

# Backup with custom format (smaller, faster restore)
pg_dump -U username -Fc database_name > backup.backup

# Backup specific tables
pg_dump -U username -t table1 -t table2 database_name > tables_backup.sql

# Backup with compression
pg_dump -U username -Fc -Z 9 database_name > compressed_backup.backup
```

2. pg_dumpall (All Databases):

```bash
# Backup all databases and global objects
pg_dumpall -U postgres > all_databases.sql

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

3. Physical Backup (Base Backup):

```bash
```

```
# Create base backup
pg_basebackup -U postgres -D /backup/location -Ft -z -P

# With WAL files
pg_basebackup -U postgres -D /backup/location -Ft -z -P -X stream
```

Restore Methods:

1. Restore from pg_dump:

```bash
# Restore SQL format
psql -U username -d database_name < backup.sql

# Restore custom format
pg_restore -U username -d database_name backup.backup

# Restore with specific options
pg_restore -U username -d database_name --clean --if-exists backup.backup

# Restore specific tables
pg_restore -U username -d database_name -t table1 -t table2 backup.backup
```

2. Point-in-Time Recovery:

```bash
# Stop PostgreSQL
sudo systemctl stop postgresql

# Restore base backup
tar -xf base_backup.tar -C /var/lib/postgresql/data/

# Configure recovery
echo "restore_command = 'cp /backup/wal/%f %p'" > /var/lib/postgresql/data/recovery.conf
echo "recovery_target_time = '2024-01-15 14:30:00'" >> /var/lib/postgresql/data/recovery.conf

# Start PostgreSQL
sudo systemctl start postgresql
```

Q18: What are common PostgreSQL configuration parameters?

Answer:

Connection Settings:

```bash
```

```
# postgresql.conf

# Maximum number of concurrent connections
max_connections = 100

# Listening addresses
listen_addresses = '*'  # or specific IP addresses

# Port number
port = 5432
```

Memory Settings:

```bash
# Shared memory for caching data
shared_buffers = '256MB'  # 25% of RAM for dedicated server

# Memory for complex operations per connection
work_mem = '4MB'

# Memory for maintenance operations
maintenance_work_mem = '64MB'

# Kernel buffer cache size estimate
effective_cache_size = '1GB'  # 50-75% of total RAM
```

Write-Ahead Logging (WAL):

```bash
# WAL level for replication
wal_level = replica

# WAL buffer size
wal_buffers = '16MB'

# Checkpoint settings
checkpoint_timeout = '5min'
checkpoint_completion_target = 0.5
```

Logging Settings:

```bash
# Log directory
log_directory = 'pg_log'

# Log file naming
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'

# What to log
log_statement = 'none'  # none, ddl, mod, all
log_min_duration_statement = 1000  # Log slow queries (ms)
log_line_prefix = '%t [%p-%l] %q%u@%d '
```

Performance Settings:

```bash
# Query planner cost parameters
random_page_cost = 4.0  # Lower for SSDs (1.1)
effective_io_concurrency = 1  # Higher for SSDs (200)

# Parallel query settings
max_parallel_workers_per_gather = 2
max_parallel_workers = 8
```

Applying Configuration Changes:

```sql
-- Reload configuration without restart
SELECT pg_reload_conf();

-- Check current settings
SHOW shared_buffers;
SHOW work_mem;

-- Show all settings
SELECT name, setting, unit, context
FROM pg_settings
WHERE name LIKE '%buffer%';
```

Security and User Management

Q19: How do you manage users and permissions in PostgreSQL?

Answer:

Creating Users/Roles:

```sql
-- Create role (user)
CREATE ROLE username WITH LOGIN PASSWORD 'secure_password';

-- Create user (same as role with LOGIN)
CREATE USER username WITH PASSWORD 'secure_password';

-- Create role with specific privileges
CREATE ROLE app_user WITH
    LOGIN
    PASSWORD 'password'
    CREATEDB
    VALID UNTIL '2025-12-31';

-- Create role without login (group role)
CREATE ROLE developers;
```

Granting Permissions:

```sql
```

```
-- Grant database access
GRANT CONNECT ON DATABASE myapp TO app_user;

-- Grant schema usage
GRANT USAGE ON SCHEMA public TO app_user;

-- Grant table permissions
GRANT SELECT, INSERT, UPDATE, DELETE ON users TO app_user;
GRANT SELECT ON orders TO app_user;

-- Grant permissions on all tables in schema
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO app_user;

-- Grant permissions on future tables
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO app_user;

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

Role Inheritance and Groups:

```
sql

-- Create group roles
CREATE ROLE developers;
CREATE ROLE managers;

-- Grant group permissions
GRANT SELECT, INSERT, UPDATE ON users TO developers;
GRANT ALL PRIVILEGES ON users TO managers;

-- Add users to groups
GRANT developers TO john, jane;
GRANT managers TO alice;

-- Role with inheritance (default)
CREATE ROLE bob WITH LOGIN PASSWORD 'password' INHERIT;
GRANT developers TO bob;  -- bob inherits developers permissions
```

Viewing Permissions:

```
sql
```

```
-- List all roles
SELECT rolname, rolsuper, rolcreaterole, rolcreatedb, rolcanlogin
FROM pg_roles;

-- Check table permissions
SELECT grantee, privilege_type
FROM information_schema.role_table_grants
WHERE table_name = 'users';

-- Check role memberships
SELECT r.rolname, m.rolname as member_of
FROM pg_roles r
JOIN pg_auth_members am ON r.oid = am.member
JOIN pg_roles m ON am.roleid = m.oid;
```

Q20: What are some PostgreSQL security best practices?

Answer:

1. Authentication and Access Control:

```
bash

# pg_hba.conf configuration
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# Local connections
local   all             postgres                                peer
local   all             all                                     md5

# Remote connections
host    all             all             192.168.1.0/24          md5
hostssl myapp           app_user        0.0.0.0/0               md5

# Reject all other connections
host    all             all             0.0.0.0/0               reject
```

2. Network Security:

```
bash

# postgresql.conf
# Only listen on specific interfaces
listen_addresses = 'localhost,192.168.1.100'

# Use SSL/TLS
ssl = on
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
```

3. Password Security:

```
sql
```

```
-- Strong password policy
CREATE ROLE user1 WITH LOGIN PASSWORD 'ComplexP@ssw0rd123!';

-- Password expiration
ALTER ROLE user1 VALID UNTIL '2025-12-31';

-- Disable password authentication for specific users
ALTER ROLE admin_user WITH PASSWORD NULL;
```

4. Principle of Least Privilege:

```
sql

-- Create specific roles for different access levels
CREATE ROLE read_only;
GRANT CONNECT ON DATABASE myapp TO read_only;
GRANT USAGE ON SCHEMA public TO read_only;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO read_only;

CREATE ROLE app_writer;
GRANT CONNECT ON DATABASE myapp TO app_writer;
GRANT USAGE ON SCHEMA public TO app_writer;
GRANT SELECT, INSERT, UPDATE ON specific_tables TO app_writer;

-- Don't grant unnecessary privileges
-- Revoke public schema access if not needed
REVOKE ALL ON SCHEMA public FROM PUBLIC;
```

5. Audit and Monitoring:

```
sql

-- Enable logging of connections and statements
-- In postgresql.conf:
log_connections = on
log_disconnections = on
log_statement = 'all'  -- or 'mod' for modifications only
log_min_duration_statement = 0  -- Log all statements

-- Create audit triggers for sensitive tables
CREATE OR REPLACE FUNCTION audit_trigger()
RETURNS TRIGGER AS $
BEGIN
    INSERT INTO audit_log (table_name, operation, user_name, timestamp, old_values, new_values)
    VALUES (TG_TABLE_NAME, TG_OP, current_user, now(),
            CASE WHEN TG_OP = 'DELETE' THEN row_to_json(OLD) ELSE NULL END,
            CASE WHEN TG_OP IN ('INSERT', 'UPDATE') THEN row_to_json(NEW) ELSE NULL END);
    RETURN CASE WHEN TG_OP = 'DELETE' THEN OLD ELSE NEW END;
END;
$ LANGUAGE plpgsql;
```

6. Data Protection:

```
sql
```

```
-- Use views to restrict column access
CREATE VIEW user_public_info AS
SELECT id, username, email, created_at
FROM users;  -- Hide sensitive columns like password_hash

GRANT SELECT ON user_public_info TO public_role;
REVOKE ALL ON users FROM public_role;

-- Row Level Security (RLS)
ALTER TABLE user_data ENABLE ROW LEVEL SECURITY;

-- Create policy for RLS
CREATE POLICY user_data_policy ON user_data
  FOR ALL TO app_user
  USING (user_id = current_user_id());
```

Common Interview Scenarios

Q21: How would you find and optimize a slow query?

Answer:

Step 1: Identify the Slow Query

sql

```
-- Enable slow query logging
-- In postgresql.conf: log_min_duration_statement = 1000

-- Or use pg_stat_statements extension
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

-- Find slowest queries
SELECT
  query,
  calls,
  total_time,
  mean_time,
  rows
FROM pg_stat_statements
ORDER BY mean_time DESC
LIMIT 5;
```

Step 2: Analyze the Query

sql

```
-- Let's say we found this slow query:
SELECT u.name, u.email, COUNT(o.id) as order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE u.created_at >= '2024-01-01'
GROUP BY u.id, u.name, u.email
ORDER BY order_count DESC;

-- Analyze execution plan
EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT)
SELECT u.name, u.email, COUNT(o.id) as order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE u.created_at >= '2024-01-01'
GROUP BY u.id, u.name, u.email
ORDER BY order_count DESC;
```

Step 3: Optimization Strategies

```sql
-- 1. Add missing indexes
CREATE INDEX idx_users_created_at ON users(created_at);
CREATE INDEX idx_orders_user_id ON orders(user_id);

-- 2. Consider composite index
CREATE INDEX idx_users_created_name ON users(created_at, name, email);

-- 3. Rewrite query if needed
-- Instead of LEFT JOIN with COUNT, use subquery:
SELECT
  u.name,
  u.email,
  COALESCE(o.order_count, 0) as order_count
FROM users u
LEFT JOIN (
  SELECT user_id, COUNT(*) as order_count
  FROM orders
  GROUP BY user_id
) o ON u.id = o.user_id
WHERE u.created_at >= '2024-01-01'
ORDER BY order_count DESC;

-- 4. Update statistics
ANALYZE users;
ANALYZE orders;
```

Q22: How do you handle a database that's running out of space?

Answer:

Step 1: Identify Space Usage

```sql
```

```
-- Check database sizes
SELECT
    datname,
    pg_size_pretty(pg_database_size(datname)) as size
FROM pg_database
ORDER BY pg_database_size(datname) DESC;

-- Check table sizes
SELECT
    schemaname,
    tablename,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) as total_size,
    pg_size_pretty(pg_relation_size(schemaname||'.'||tablename)) as table_size,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename) - pg_relation_size(schemaname||'.'||tablename)) as
FROM pg_tables
ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC
LIMIT 10;

-- Check for bloated tables
SELECT
    schemaname,
    tablename,
    n_dead_tup,
    n_live_tup,
    ROUND(100.0 * n_dead_tup / (n_live_tup + n_dead_tup), 2) as dead_ratio
FROM pg_stat_user_tables
WHERE n_live_tup > 0
ORDER BY n_dead_tup DESC;
```

Step 2: Immediate Actions

sql

```
-- 1. Run VACUUM to reclaim space
VACUUM VERBOSE;  -- For all tables
VACUUM VERBOSE large_table;  -- For specific table

-- 2. For heavily bloated tables, consider VACUUM FULL (locks table)
VACUUM FULL bloated_table;

-- 3. Clean up temporary files
-- Check for large temp files in postgresql.conf temp directory

-- 4. Archive old data
-- Move old records to archive tables
CREATE TABLE orders_archive AS
SELECT * FROM orders WHERE created_at < '2023-01-01';

DELETE FROM orders WHERE created_at < '2023-01-01';
VACUUM ANALYZE orders;
```

Step 3: Long-term Solutions

sql

```
-- 1. Implement table partitioning for large tables
CREATE TABLE orders_new (
    id SERIAL,
    user_id INTEGER,
    created_at DATE,
    amount DECIMAL
) PARTITION BY RANGE (created_at);

-- Create monthly partitions
CREATE TABLE orders_2024_01 PARTITION OF orders_new
FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

-- 2. Set up automated archiving
CREATE OR REPLACE FUNCTION archive_old_data()
RETURNS void AS $
BEGIN
    -- Archive data older than 2 years
    INSERT INTO orders_archive
    SELECT * FROM orders
    WHERE created_at < CURRENT_DATE - INTERVAL '2 years';

    DELETE FROM orders
    WHERE created_at < CURRENT_DATE - INTERVAL '2 years';
END;
$ LANGUAGE plpgsql;

-- Schedule with pg_cron extension or external scheduler
```

Q23: How do you troubleshoot connection issues?

Answer:

Common Connection Problems and Solutions:

1. "Connection refused" Error

bash

```bash
# Check if PostgreSQL is running
sudo systemctl status postgresql
# or
ps aux | grep postgres

# Check listening addresses and ports
sudo netstat -tlnp | grep :5432

# Start PostgreSQL if stopped
sudo systemctl start postgresql
```

2. "Too many connections" Error

sql

```
-- Check current connections
SELECT count(*) FROM pg_stat_activity;

-- Check max connections
SHOW max_connections;

-- See who's connected
SELECT
    pid,
    usename,
    application_name,
    client_addr,
    state,
    query_start
FROM pg_stat_activity;

-- Kill idle connections if needed
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE state = 'idle'
  AND query_start < now() - interval '1 hour';

-- Increase max_connections in postgresql.conf
max_connections = 200  -- Restart required
```

3. Authentication Issues

```bash
# Check pg_hba.conf for authentication rules
# Common configurations:

# Local connections
local   all         all                         peer
local   all         all                         md5

# Remote connections
host    all         all         127.0.0.1/32        md5
host    all         all         ::1/128             md5
host    all         all         192.168.1.0/24      md5

# After changes, reload configuration
sudo systemctl reload postgresql
```

4. Permission Issues

```sql
```

```
-- Check user permissions
SELECT
  r.rolname,
  r.rolsuper,
  r.rolcreaterole,
  r.rolcreatedb,
  r.rolcanlogin,
  r.rolconnlimit
FROM pg_roles r
WHERE r.rolname = 'username';

-- Grant connection permission
GRANT CONNECT ON DATABASE myapp TO username;

-- Check database-specific permissions
SELECT
  datname,
  has_database_privilege('username', datname, 'CONNECT') as can_connect
FROM pg_database;
```

Q24: Explain the differences between DELETE, TRUNCATE, and DROP.

Answer:

Operation	DELETE	TRUNCATE	DROP
Purpose	Remove rows	Remove all rows	Remove table structure
Speed	Slow (row-by-row)	Fast (deallocates pages)	Fast
WHERE Clause	Yes	No	No
Rollback	Yes	Yes*	Yes*
Triggers	Fires	No triggers	No triggers
Identity/Serial	Doesn't reset	Resets	N/A
Foreign Keys	Checks constraints	Can't truncate if referenced	CASCADE option
Transaction Log	Full logging	Minimal logging	Minimal logging

Examples:

DELETE:

```sql
-- Delete specific rows
DELETE FROM orders WHERE created_at < '2023-01-01';

-- Delete all rows (slow)
DELETE FROM temp_table;

-- Delete with JOIN
DELETE o FROM orders o
JOIN users u ON o.user_id = u.id
WHERE u.status = 'inactive';

-- Returns number of affected rows
-- Triggers fire for each deleted row
-- Can be rolled back
-- Doesn't reset SERIAL sequences
```

TRUNCATE:

```sql
-- Remove all rows quickly
TRUNCATE TABLE temp_table;

-- Multiple tables
TRUNCATE TABLE table1, table2, table3;

-- With CASCADE (to handle foreign key constraints)
TRUNCATE TABLE parent_table CASCADE;

-- Reset identity columns
TRUNCATE TABLE users RESTART IDENTITY;

-- Fast operation - deallocates data pages
-- Resets AUTO_INCREMENT/SERIAL counters
-- Cannot use WHERE clause
-- No triggers fired
-- Cannot truncate if table is referenced by FK (unless CASCADE)
```

DROP:

```sql
-- Remove table structure and all data
DROP TABLE temp_table;

-- Remove if exists (no error if doesn't exist)
DROP TABLE IF EXISTS temp_table;

-- Remove with CASCADE (removes dependent objects)
DROP TABLE users CASCADE;

-- Multiple tables
DROP TABLE table1, table2, table3;

-- Completely removes table from database
-- All data, indexes, triggers, constraints removed
-- Cannot be undone after COMMIT (in most cases)
-- Dependent objects must be dropped first or use CASCADE
```

When to Use Each:

- **DELETE**: When you need to remove specific rows, need triggers to fire, or want to keep table structure
- **TRUNCATE**: When you need to quickly remove all data and reset counters, but keep table structure
- **DROP**: When you no longer need the table and want to remove it completely

Q25: What are some common PostgreSQL error messages and their solutions?

Answer:

1. "relation does not exist"

```sql
```

```
-- Error: relation "users" does not exist

-- Common causes and solutions:
-- a) Wrong schema
SET search_path TO schema_name, public;
-- or use qualified name
SELECT * FROM schema_name.users;

-- b) Case sensitivity
SELECT * FROM "Users";  -- If table name has capitals

-- c) Table doesn't exist
CREATE TABLE users (...);

-- d) Check if table exists
SELECT tablename FROM pg_tables WHERE tablename = 'users';
```

2. "column does not exist"

sql

```
-- Error: column "name" does not exist

-- Common causes:
-- a) Typo in column name
SELECT user_name FROM users;  -- Instead of 'name'

-- b) Case sensitivity
SELECT "Name" FROM users;  -- If column has capitals

-- c) Wrong table alias
SELECT u.full_name FROM users u;  -- Instead of just 'full_name'

-- Check column names
SELECT column_name FROM information_schema.columns
WHERE table_name = 'users';
```

3. "duplicate key value violates unique constraint"

sql

```
-- Error when inserting duplicate values

-- Solutions:
-- a) Use INSERT ... ON CONFLICT
INSERT INTO users (email, name)
VALUES ('john@email.com', 'John')
ON CONFLICT (email) DO UPDATE SET name = EXCLUDED.name;

-- b) Use INSERT ... ON CONFLICT DO NOTHING
INSERT INTO users (email, name)
VALUES ('john@email.com', 'John')
ON CONFLICT (email) DO NOTHING;

-- c) Check existing data first
INSERT INTO users (email, name)
SELECT 'john@email.com', 'John'
WHERE NOT EXISTS (
  SELECT 1 FROM users WHERE email = 'john@email.com'
);
```

4. "permission denied"

```
sql
-- Error: permission denied for table users

-- Solutions:
-- a) Grant necessary permissions
GRANT SELECT, INSERT, UPDATE, DELETE ON users TO username;

-- b) Check current permissions
SELECT grantee, privilege_type
FROM information_schema.role_table_grants
WHERE table_name = 'users';

-- c) Connect as superuser and fix permissions
-- d) Check if user exists and can login
SELECT rolname, rolcanlogin FROM pg_roles WHERE rolname = 'username';
```

5. "deadlock detected"

```
sql
```

```
-- Error: deadlock detected

-- Prevention strategies:
-- a) Always acquire locks in same order
BEGIN;
LOCK TABLE table1 IN SHARE MODE;
LOCK TABLE table2 IN SHARE MODE;
-- ... operations
COMMIT;

-- b) Keep transactions short
-- c) Use appropriate isolation levels
-- d) Handle deadlock in application code with retry logic

-- Monitor deadlocks
SELECT * FROM pg_stat_database WHERE datname = 'your_db';
```

6. "out of shared memory"

bash

```
# Error: out of shared memory

# Solutions in postgresql.conf:
shared_buffers = '256MB'  # Increase if you have RAM
max_connections = 100     # Reduce if too high

# Calculate memory usage:
# shared_buffers + (max_connections * work_mem) should be < available RAM
```

7. "could not extend file"

bash

```
# Error: could not extend file ... No space left on device

# Solutions:
# a) Free up disk space
df -h  # Check disk usage
du -sh /var/lib/postgresql/  # Check PostgreSQL directory size

# b) Run VACUUM to reclaim space
VACUUM FULL;

# c) Archive old data
# d) Move to larger disk
# e) Add tablespace on different disk
```

This comprehensive guide covers the fundamental PostgreSQL concepts that are commonly tested in interviews. Each answer includes practical examples and real-world scenarios that demonstrate not just theoretical knowledge, but hands-on experience with PostgreSQL.