PostgreSQL

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Explain the difference between MVCC (Multi-Version Concurrency Control) in PostgreSQL and traditional locking mechanisms. How does it impact performance?

MVCC (Multi-Version Concurrency Control) is PostgreSQL's approach to handling concurrent transactions without traditional locking.

Key Differences:

- **Traditional Locking:** Readers block writers and writers block readers, leading to contention
- **MVCC:** Each transaction sees a snapshot of data as it existed at transaction start, allowing readers and writers to work concurrently without blocking each other

How MVCC Works:

- Each row version contains transaction IDs (xmin for creation, xmax for deletion)
- Transactions determine visibility based on their snapshot and transaction IDs
- Old row versions remain until VACUUM removes them

Performance Impact:

- **Advantages:** High concurrency, no read locks, predictable read performance
- **Trade-offs:** Increased storage for row versions, need for VACUUM maintenance, potential bloat if not managed
- **Write amplification:** Updates create new row versions rather than in-place modifications

This design makes PostgreSQL excellent for read-heavy workloads with occasional writes, but requires proper VACUUM tuning for write-heavy scenarios.

2. What are the different types of indexes in PostgreSQL and when would you use each?

PostgreSQL supports multiple index types, each optimized for specific use cases:

1. B-tree (Default)

- **Use case:** General purpose, equality and range queries
- **Best for:** Sortable data, WHERE clauses with =, <, >, <=, >=, BETWEEN
- **Example:** CREATE INDEX idx_users_email ON users(email);

2. Hash

- **Use case:** Equality comparisons only
- **Best for:** Simple equality lookups, smaller than B-tree
- **Limitation:** No range queries, not WAL-logged before PG 10

3. GiST (Generalized Search Tree)

- **Use case:** Geometric data, full-text search, range types
- **Best for:** PostGIS queries, nearest-neighbor searches, overlapping ranges

4. GIN (Generalized Inverted Index)

- **Use case:** Multiple values per row (arrays, JSONB, full-text)
- **Best for:** JSONB containment (@>), array operations, text search
- **Example:** CREATE INDEX idx_data_jsonb ON logs USING gin(data);

5. BRIN (Block Range Index)

- **Use case:** Very large tables with natural ordering
- **Best for:** Time-series data, append-only tables
- **Advantage:** Extremely small index size

6. SP-GiST (Space-Partitioned GiST)

- **Use case:** Non-balanced data structures
- **Best for:** Phone numbers, IP addresses, geometric data

3. How do you diagnose and optimize a slow PostgreSQL query? Walk through your methodology.

Systematic approach to query optimization:

1. Gather Query Execution Plan

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT * FROM orders
WHERE customer_id = 123
AND created_at > '2024-01-01';
```

2. Analyze Key Metrics

- **Execution time:** Actual time vs planning time
- **Seq Scan:** Table scans indicate missing indexes
- **Rows estimate vs actual:** Large differences suggest stale statistics
- **Buffers:** Shared hits (cache) vs reads (disk I/O)
- **Nested loops:** Can be expensive with large datasets

3. Common Issues and Fixes

- **Missing indexes:** Add appropriate index type
- **Stale statistics:** Run ANALYZE on tables
- **Wrong index choice:** Adjust random_page_cost, effective_cache_size
- **Bloated tables:** Run VACUUM FULL or pg_repack
- **Poor join order:** Rewrite query or adjust join_collapse_limit

4. Monitor with pg_stat_statements

```
SELECT query, calls, mean_exec_time,
       total_exec_time
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
```

5. Additional Tools

- **pg_stat_user_tables:** Check for table bloat and vacuum needs
- **pg_stat_user_indexes:** Identify unused indexes
- **auto_explain:** Automatically log slow queries

4. Explain the difference between VACUUM, VACUUM FULL, and AUTOVACUUM. When would you use each?

VACUUM operations manage dead tuples and table bloat in PostgreSQL's MVCC system.

VACUUM (Standard)

- **What it does:** Marks dead tuples as reusable space, updates statistics, prevents transaction ID wraparound
- **Behavior:** Non-blocking, can run concurrently with normal operations
- **Limitation:** Doesn't return disk space to OS, only makes it reusable within the table
- **When to use:** Regular maintenance, especially after large DELETE/UPDATE operations

```
VACUUM VERBOSE users;
VACUUM (ANALYZE) orders;
```

VACUUM FULL

- **What it does:** Rewrites entire table, compacts data, returns space to OS
- **Behavior:** Takes exclusive lock (blocks all operations), requires extra disk space
- **Impact:** Rebuilds indexes, can take hours on large tables
- **When to use:** Severe bloat (>50%), one-time cleanup after major data purge
- **Alternative:** Consider pg_repack for online reorganization

AUTOVACUUM (Automated)

- **What it does:** Background process that automatically runs VACUUM and ANALYZE
- **Triggers:** Based on autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor
- **Configuration:** Can be tuned per-table or globally

```
ALTER TABLE large_table SET (
  autovacuum_vacuum_scale_factor = 0.05,
  autovacuum_vacuum_threshold = 1000
);
```

Best Practices

- Keep autovacuum enabled (default)
- Tune for write-heavy tables
- Monitor bloat with pg_stat_user_tables
- Use manual VACUUM during maintenance windows for critical tables
- Avoid VACUUM FULL in production; use pg_repack instead

5. How do you implement and optimize JSONB queries in PostgreSQL? What are the performance considerations?

JSONB is PostgreSQL's binary JSON format optimized for querying and indexing.

Basic JSONB Operations

```
-- Containment query
SELECT * FROM events
WHERE data @> '{"user_id": 123}';

-- Path extraction
SELECT data->'user'->>'name' AS name
FROM events;

-- Existence check
SELECT * FROM events
WHERE data ? 'transaction_id';
```

Indexing Strategies

1. GIN Index (General Purpose)

```
CREATE INDEX idx_events_data
ON events USING gin(data);

-- For specific paths
CREATE INDEX idx_events_user
ON events USING gin((data->'user'));
```

2. Expression Index (Specific Fields)

```
CREATE INDEX idx_events_user_id
ON events ((data->>'user_id'));
```

3. Partial Index (Filtered)

```
CREATE INDEX idx_active_events
```

```
ON events USING gin(data)
WHERE data->>'status' = 'active';
```

Performance Considerations

- **GIN index size:** Can be large; monitor disk usage
- **Write overhead:** GIN updates are expensive; tune fastupdate
- **Query specificity:** Use expression indexes for frequently queried paths
- **Data normalization:** Consider extracting frequently queried fields to columns
- **jsonb_path_ops:** Smaller, faster GIN variant for containment queries only

Optimization Tips

- Use @> (containment) instead of ->> with WHERE when possible
- Combine JSONB with traditional columns for best performance
- Use jsonb_path_ops for containment-only queries
- Monitor index bloat and rebuild periodically

6. Explain PostgreSQL's transaction isolation levels and provide real-world scenarios where each is appropriate.

PostgreSQL implements SQL standard isolation levels to control transaction visibility and consistency.

1. Read Uncommitted (Not Really Implemented)

- PostgreSQL treats this as Read Committed
- No dirty reads possible in PostgreSQL

2. Read Committed (Default)

- **Behavior:** Each statement sees committed data as of statement start
- **Prevents:** Dirty reads
- **Allows:** Non-repeatable reads, phantom reads
- **Use case:** Most applications, web requests, general OLTP

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL
  READ COMMITTED;
SELECT balance FROM accounts
  WHERE id = 1;
COMMIT;
```

3. Repeatable Read

- **Behavior:** Transaction sees snapshot from first query
- **Prevents:** Dirty reads, non-repeatable reads, phantom reads (in PostgreSQL)
- **Caveat:** Can cause serialization failures on concurrent updates
- **Use case:** Reports, data exports, multi-step calculations requiring consistency

Example scenario: Financial report generation where totals must be consistent across multiple queries.

4. Serializable

- **Behavior:** Guarantees truly serial execution semantics
- **Prevents:** All anomalies including write skew
- **Implementation:** Uses Serializable Snapshot Isolation (SSI)
- **Trade-off:** Higher chance of serialization failures requiring retry logic
- **Use case:** Critical financial transactions, inventory systems with strict constraints

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL
  SERIALIZABLE;
-- Transfer between accounts
UPDATE accounts SET balance = balance - 100
  WHERE id = 1;
```

```
UPDATE accounts SET balance = balance + 100
  WHERE id = 2;
COMMIT;
```

Real-World Recommendations

- **Web applications:** Read Committed (default)
- **Batch processing:** Repeatable Read
- **Banking transfers:** Serializable with retry logic
- **Analytics queries:** Repeatable Read for consistency

7. How does PostgreSQL replication work? Compare and contrast logical vs physical replication.

PostgreSQL offers two main replication approaches with different use cases and characteristics.

Physical Replication (Streaming Replication)

How it works:

- Replicates WAL (Write-Ahead Log) segments at binary level
- Standby replays WAL to maintain exact copy
- Byte-for-byte identical replica

Configuration:

```
-- Primary: postgresql.conf
wal_level = replica
max_wal_senders = 5

-- Standby: recovery.conf (PG <12)
primary_conninfo = 'host=primary...'
hot_standby = on
```

Characteristics:

- **Pros:** Low overhead, simple setup, entire cluster replicated
- **Cons:** Same PostgreSQL version required, all-or-nothing (entire cluster)
- **Use cases:** High availability, disaster recovery, read replicas

Logical Replication (PG 10+)

How it works:

- Replicates data changes at logical level (INSERT/UPDATE/DELETE)
- Uses publications (primary) and subscriptions (replica)
- Decodes WAL to logical changes

Configuration:

```
-- Primary
wal_level = logical
CREATE PUBLICATION pub_orders
  FOR TABLE orders;

-- Subscriber
CREATE SUBSCRIPTION sub_orders
  CONNECTION 'host=primary...'
  PUBLICATION pub_orders;
```

Characteristics:

- **Pros:** Selective replication, cross-version, cross-platform, allows writes on subscriber
- **Cons:** Higher overhead, DDL not replicated, no sequence replication
- **Use cases:** Multi-master setups, data consolidation, version upgrades, partial replication

Comparison Table

- **Granularity:** Physical (cluster) vs Logical (table-level)
- **Performance:** Physical (faster) vs Logical (more overhead)
- **Flexibility:** Physical (rigid) vs Logical (flexible)
- **Version compatibility:** Physical (same version) vs Logical (cross-version)

8. What are CTEs (Common Table Expressions) and when should you use them vs subqueries? Explain materialization in PostgreSQL 12+.

CTEs provide a way to write auxiliary statements for use in a larger query.

Basic CTE Syntax

```
WITH recent_orders AS (
  SELECT customer_id, order_date, amount
  FROM orders
  WHERE order_date > CURRENT_DATE - 30
)
SELECT c.name, COUNT(*) as order_count
FROM customers c
JOIN recent_orders ro ON c.id = ro.customer_id
GROUP BY c.name;
```

CTEs vs Subqueries

Advantages of CTEs:

- **Readability:** Named, reusable, easier to understand
- **Recursion:** Support recursive queries (subqueries don't)
- **Multiple references:** Can reference the same CTE multiple times
- **Debugging:** Easier to test individual CTEs

When to use subqueries:

- Simple, one-time use expressions
- When you want optimizer to inline the logic
- Correlated subqueries for row-by-row operations

Materialization Behavior (PostgreSQL 12+)

Before PG 12: CTEs were always materialized (optimization fence) **PG 12+:** CTEs are automatically inlined unless:

- CTE is referenced multiple times
- CTE contains side effects (INSERT/UPDATE/DELETE)
- Explicitly marked with MATERIALIZED keyword

```
-- Force materialization
WITH MATERIALIZED expensive_calc AS (
  SELECT customer_id,
    complex_calculation() as result
  FROM large_table
)
SELECT * FROM expensive_calc
WHERE result > 100;

-- Prevent materialization
WITH NOT MATERIALIZED simple_filter AS (
  SELECT * FROM orders
  WHERE status = 'pending'
)
SELECT * FROM simple_filter
LIMIT 10;
```

Use Cases

- **Recursive queries:** Hierarchical data, graph traversal
- **Data modification:** WITH ... INSERT/UPDATE/DELETE
- **Complex calculations:** Break down into logical steps

- **Multiple aggregations:** Different grouping levels

9. Explain partitioning in PostgreSQL. What are the different strategies and how do you choose between them?

Table partitioning divides large tables into smaller, more manageable pieces while appearing as a single table.

Partitioning Strategies

1. Range Partitioning

- **Use case:** Time-series data, sequential data
- **Best for:** Date ranges, numeric ranges

```
CREATE TABLE measurements (
  id BIGSERIAL,
  recorded_at TIMESTAMP,
  value NUMERIC
) PARTITION BY RANGE (recorded_at);

CREATE TABLE measurements_2024_01
  PARTITION OF measurements
  FOR VALUES FROM ('2024-01-01')
  TO ('2024-02-01');
```

2. List Partitioning

- **Use case:** Discrete categories, geographic regions
- **Best for:** Country codes, status values, categories

```
CREATE TABLE orders (
  id BIGSERIAL,
  region TEXT,
  amount NUMERIC
) PARTITION BY LIST (region);

CREATE TABLE orders_us
  PARTITION OF orders
  FOR VALUES IN ('US', 'USA');
```

3. Hash Partitioning

- **Use case:** Evenly distribute data without natural partitioning key
- **Best for:** Load balancing, no obvious partition key

```
CREATE TABLE users (
  id BIGSERIAL,
  email TEXT
) PARTITION BY HASH (id);

CREATE TABLE users_p0
  PARTITION OF users
  FOR VALUES WITH (MODULUS 4, REMAINDER 0);
```

Choosing a Strategy

- **Range:** Time-series, logs, events with natural ordering
- **List:** Multi-tenant apps, geographic data, categorical data
- **Hash:** No natural partition key, need even distribution

Benefits

- **Query performance:** Partition pruning eliminates irrelevant partitions
- **Maintenance:** Drop old partitions instead of DELETE
- **Bulk operations:** Load/unload data by partition
- **Index size:** Smaller indexes per partition

Best Practices

- Include partition key in indexes
- Enable constraint_exclusion or partition_pruning
- Keep partition count reasonable (<100)
- Use default partition for unmatched rows
- Automate partition creation for range partitioning

10. How do you handle connection pooling in PostgreSQL? Compare different pooling solutions and their trade-offs.

Connection pooling is critical for PostgreSQL performance since each connection consumes significant resources.

Why Connection Pooling Matters

- PostgreSQL uses process-per-connection model (not threads)
- Each connection consumes ~10MB RAM
- Connection establishment has overhead (~1-3ms)
- Recommended max_connections: 100-300 for most systems

Pooling Solutions

1. PgBouncer (Most Popular)

- **Type:** Lightweight, external pooler
- **Modes:** Session, transaction, statement
- **Pros:** Low overhead, battle-tested, simple configuration
- **Cons:** Limited protocol support, no query routing

```
[databases]
production = host=localhost
  port=5432 dbname=mydb

[pgbouncer]
pool_mode = transaction
max_client_conn = 1000
default_pool_size = 25
```

Pooling modes:

- **Session:** Connection held for client session (safest, least efficient)
- **Transaction:** Released after transaction (recommended for most apps)
- **Statement:** Released after each statement (breaks multi-statement transactions)

2. PgPool-II

- **Type:** Full-featured middleware
- **Features:** Connection pooling, load balancing, replication, query caching
- **Pros:** Rich features, query routing, automatic failover
- **Cons:** Complex configuration, higher overhead, potential bottleneck

3. Application-Level Pooling

- **Examples:** HikariCP (Java), psycopg2.pool (Python), pg-pool (Node.js)
- **Pros:** No external dependency, application-aware
- **Cons:** Per-application pools, doesn't share across services

Sizing Guidelines

```
-- Formula
pool_size = ((core_count * 2) +
  effective_spindle_count)

-- Example: 4 cores, 1 disk
Optimal pool size: ~10 connections
```

Best Practices

- Use transaction mode in PgBouncer for web apps
- Set pool size based on actual concurrency needs
- Monitor with pg_stat_activity
- Configure statement_timeout to prevent connection hogging
- Use separate pools for different workload types

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. How would you implement an LRU (Least Recently Used) cache in PostgreSQL using appropriate data structures?

LRU Cache Implementation

An **LRU cache** can be implemented using a combination of a hash table (for O(1) lookups) and a doubly-linked list (for O(1) eviction). In PostgreSQL context, you can use:

- **hstore** or **jsonb** for key-value storage
- A separate table with timestamp tracking for access order
- Triggers to maintain the access order

```
CREATE TABLE lru_cache (
  key TEXT PRIMARY KEY,
  value TEXT,
  last_accessed TIMESTAMP DEFAULT NOW()
);
CREATE INDEX idx_lru_access ON lru_cache(last_accessed);
```

Time Complexity: O(1) for get/put operations with proper indexing. Eviction requires O(log n) due to index scan.

2. Explain how PostgreSQL's B-tree index works and its time complexity for search, insert, and delete operations.

B-tree Index Structure

PostgreSQL's default index type is **B-tree**, a self-balancing tree data structure that maintains sorted data for efficient operations:

- **Structure:** Multi-level tree with nodes containing multiple keys and child pointers
- **Balanced:** All leaf nodes are at the same depth
- **Order:** Each node has between d and 2d keys (except root)

Time Complexity:

- Search: **O(log n)**
- Insert: **O(log n)**
- Delete: **O(log n)**
- Range queries: **O(log n + k)** where k is result size

B-trees are optimal for disk-based storage due to high branching factor, minimizing disk I/O operations.

3. How would you find all pairs in a PostgreSQL table that sum to a target value efficiently?

Pair Sum Problem

To find pairs that sum to a target value, use a **hash-based approach** with a self-join:

```
SELECT a.value AS val1, b.value AS val2
FROM numbers a
JOIN numbers b ON a.value + b.value = target
  AND a.id < b.id;
```

Optimized approach using hash aggregate:

```
WITH target_pairs AS (
  SELECT value, (target - value) AS complement
  FROM numbers
)
SELECT DISTINCT t1.value, t1.complement
FROM target_pairs t1
WHERE EXISTS (SELECT 1 FROM numbers WHERE value = t1.complement);
```

Time Complexity: O(n) with hash join. **Space Complexity:** O(n) for hash table.

4. Explain PostgreSQL's GiST (Generalized Search Tree) index and when you would use it over B-tree.

GiST Index Overview

GiST (Generalized Search Tree) is a balanced tree structure that supports custom data types and operators:

- **Flexible:** Supports geometric data, full-text search, and custom types
- **Lossy:** May require rechecking tuples (unlike B-tree)
- **Extensible:** Custom operator classes can be defined

Use cases over B-tree:

- Geometric queries (points, polygons, ranges)
- Full-text search with tsvector
- Range types and overlapping conditions
- Nearest-neighbor searches (KNN)

```
CREATE INDEX idx_location ON places
USING GIST (location);

SELECT * FROM places
WHERE location && box '((0,0),(1,1))';
```

Time Complexity: O(log n) for search, but with higher constants than B-tree.

5. How would you implement a sliding window algorithm to calculate moving averages in PostgreSQL?

Sliding Window with Window Functions

PostgreSQL's **window functions** provide efficient sliding window implementations:

```
SELECT
  timestamp,
  value,
  AVG(value) OVER (
    ORDER BY timestamp
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
  ) AS moving_avg_3
FROM sensor_data;
```

Time-based window:

```
SELECT
  timestamp,
  AVG(value) OVER (
    ORDER BY timestamp
    RANGE BETWEEN INTERVAL '1 hour' PRECEDING
    AND CURRENT ROW
  ) AS hourly_avg
FROM metrics;
```

Time Complexity: O(n) for sequential scan with window. **Space Complexity:** O(w) where w is window size.

6. Describe PostgreSQL's hash index implementation and its advantages/disadvantages compared to B-tree.

Hash Index Structure

Hash indexes use a hash table data structure for equality comparisons:

- **Structure:** Hash buckets with overflow chains
- **Operations:** Only supports equality (=) operator
- **WAL-logged:** Crash-safe since PostgreSQL 10

Advantages:

- Slightly faster for equality lookups: O(1) average case
- Smaller index size for certain data types

Disadvantages:

- No range queries, sorting, or pattern matching
- Cannot be used for ORDER BY
- Worst-case O(n) with hash collisions
- B-tree often performs similarly due to caching

```
CREATE INDEX idx_hash_email ON users
USING HASH (email);
```

Recommendation: Use B-tree unless proven performance benefit exists.

7. How would you detect and remove duplicates from a large table efficiently using set operations?

Duplicate Detection and Removal

Use **window functions** with ROW_NUMBER to identify duplicates efficiently:

```
DELETE FROM users
WHERE ctid NOT IN (
  SELECT MIN(ctid)
  FROM users
  GROUP BY email
);
```

More efficient approach using CTE:

```
WITH duplicates AS (
  SELECT ctid,
    ROW_NUMBER() OVER (
      PARTITION BY email ORDER BY created_at
    ) AS rn
  FROM users
)
DELETE FROM users
WHERE ctid IN (SELECT ctid FROM duplicates WHERE rn > 1);
```

Time Complexity: O(n log n) due to sorting. **Space Complexity:** O(n) for duplicate tracking. Use batch processing for very large tables.

8. Explain how PostgreSQL implements skip lists or similar structures for concurrent access patterns.

Concurrent Data Structures

PostgreSQL doesn't use skip lists directly but employs similar concurrent structures:

- **Lock-free structures:** Used in shared memory management
- **MVCC (Multi-Version Concurrency Control):** Allows concurrent reads without blocking
- **Tuple versioning:** Each row version forms a linked list

Implementation details:

- Transaction snapshots determine visibility
- No read locks for SELECT queries
- Writers don't block readers

```
SELECT xmin, xmax, * FROM users
WHERE id = 1;
```

The **xmin** and **xmax** system columns track transaction IDs for versioning. **Time Complexity:** O(1) for visibility checks with snapshot isolation.

9. How would you implement a priority queue or heap structure using PostgreSQL for task scheduling?

Priority Queue Implementation

Implement a **priority queue** using a table with proper indexing:

```
CREATE TABLE task_queue (
  id SERIAL PRIMARY KEY,
  priority INTEGER NOT NULL,
  created_at TIMESTAMP DEFAULT NOW(),
  status TEXT DEFAULT 'pending'
);
CREATE INDEX idx_priority ON task_queue(priority DESC, created_at)
WHERE status = 'pending';
```

Dequeue operation (atomic):

```
UPDATE task_queue
SET status = 'processing'
WHERE id = (
  SELECT id FROM task_queue
  WHERE status = 'pending'
  ORDER BY priority DESC, created_at
  LIMIT 1 FOR UPDATE SKIP LOCKED
) RETURNING *;
```

Time Complexity: O(log n) for dequeue with index. **SKIP LOCKED** ensures lock-free concurrent access.

10. Describe how to implement and optimize a trie (prefix tree) for autocomplete functionality in PostgreSQL.

Trie Implementation for Autocomplete

While PostgreSQL doesn't have native trie structures, you can implement prefix search efficiently:

Using B-tree with text_pattern_ops:

```
CREATE INDEX idx_prefix ON dictionary(word text_pattern_ops);

SELECT word FROM dictionary
WHERE word LIKE 'pre%'
ORDER BY word
LIMIT 10;
```

Using trigram indexes (pg_trgm):

```
CREATE EXTENSION pg_trgm;
CREATE INDEX idx_trgm ON dictionary
USING GIN (word gin_trgm_ops);

SELECT word FROM dictionary
WHERE word % 'prefix'
ORDER BY similarity(word, 'prefix') DESC;
```

Time Complexity: O(log n + k) for prefix search with B-tree, O(k) for trigram where k is result size. Trigram indexes excel at fuzzy matching.

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly. How would you architect it using PostgreSQL?

Architecture Overview

A URL shortener requires handling high read traffic, unique short code generation, and reliable redirects.

Key Components

- **Database Schema:** Store mappings between short codes and original URLs with metadata (creation time, expiry, click count)
- **Short Code Generation:** Use base62 encoding of auto-incrementing IDs or hash-based approach
- **Caching Layer:** Redis/Memcached in front of PostgreSQL to handle 90%+ of reads
- **Load Balancing:** Distribute requests across multiple application servers
- **CDN:** Cache redirect responses at edge locations

PostgreSQL Schema

```
CREATE TABLE urls (
  id BIGSERIAL PRIMARY KEY,
  short_code VARCHAR(10) UNIQUE NOT NULL,
  original_url TEXT NOT NULL,
  created_at TIMESTAMP DEFAULT NOW(),
  expires_at TIMESTAMP,
  click_count BIGINT DEFAULT 0
);
CREATE INDEX idx_short_code ON urls(short_code);
CREATE INDEX idx_expires ON urls(expires_at) WHERE expires_at IS NOT NULL;
```

Scaling Considerations

- **Read Replicas:** Use PostgreSQL streaming replication for read-heavy workloads
- **Partitioning:** Partition by creation date for time-based data management
- **Connection Pooling:** Use PgBouncer to manage database connections efficiently
- **Analytics:** Separate analytics database or use PostgreSQL logical replication to data warehouse

CAP Theorem Trade-offs

Favor **Availability and Partition Tolerance** over strict consistency. Eventual consistency for click counts is acceptable. Use PostgreSQL for durability of mappings while caching for performance.

2. How would you design a social media news feed system with PostgreSQL? Discuss the fan-out approach and performance optimization.

Feed Generation Approaches

Two primary strategies exist for generating social feeds:

- **Fan-out on Write (Push):** Pre-compute feeds when posts are created
- **Fan-out on Read (Pull):** Compute feeds on-demand when users request them
- **Hybrid:** Push for active users, pull for inactive users

PostgreSQL Schema Design

```
CREATE TABLE posts (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT NOT NULL,
  content TEXT,
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE feed_items (
  user_id BIGINT,
  post_id BIGINT,
  created_at TIMESTAMP,
  PRIMARY KEY (user_id, post_id)
) PARTITION BY HASH(user_id);
```

Performance Optimizations

- **Partitioning:** Hash partition feed_items by user_id for better query performance
- **Materialized Views:** Pre-aggregate popular content for discovery feeds
- **BRIN Indexes:** Use for time-series data on created_at columns
- **Batch Inserts:** Use COPY or batch INSERT for fan-out operations
- **Read Replicas:** Route feed reads to replicas, writes to primary

Scaling Strategy

For users with millions of followers, use **fan-out on read** with aggressive caching. Store only recent feed items (30 days) in PostgreSQL, archive older data to object storage. Use Redis sorted sets for hot feeds with PostgreSQL as source of truth.

Query Example

```
SELECT p.* FROM feed_items f
JOIN posts p ON f.post_id = p.id
WHERE f.user_id = $1
ORDER BY f.created_at DESC
LIMIT 20;
```

3. Design a real-time chat application backend. How would you use PostgreSQL for message persistence while ensuring low latency?

Architecture Components

- **WebSocket Servers:** Handle real-time connections (stateful)
- **Message Queue:** RabbitMQ/Kafka for message routing
- **PostgreSQL:** Persistent storage for message history
- **Redis:** Online user presence, unread counts, recent messages cache
- **Object Storage:** Media files (images, videos)

PostgreSQL Schema

```
CREATE TABLE conversations (
  id BIGSERIAL PRIMARY KEY,
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE messages (
  id BIGSERIAL PRIMARY KEY,
  conversation_id BIGINT,
  sender_id BIGINT,
  content TEXT,
  created_at TIMESTAMP DEFAULT NOW()
) PARTITION BY RANGE (created_at);
```

Latency Optimization

- **Asynchronous Writes:** Acknowledge message delivery immediately, persist to PostgreSQL asynchronously
- **Write-Behind Cache:** Buffer writes in Redis, batch flush to PostgreSQL every few seconds
- **Partitioning:** Range partition messages by month for efficient queries and archival

- **Indexes:** Create composite index on (conversation_id, created_at) for pagination

Consistency Model

Use **eventual consistency** for message persistence. Messages are delivered via WebSocket first (low latency), then persisted to PostgreSQL. If database write fails, retry from message queue. PostgreSQL ensures durability but doesn't block real-time delivery.

Read Pattern Optimization

```
SELECT * FROM messages
WHERE conversation_id = $1
  AND created_at < $2
ORDER BY created_at DESC
LIMIT 50;
```

Cache recent 100 messages per conversation in Redis. Query PostgreSQL only for history scrollback.

4. Design a distributed rate limiting system. How would you implement it with PostgreSQL and handle high-concurrency scenarios?

Rate Limiting Strategies

- **Token Bucket:** Allows bursts, refills at fixed rate
- **Leaky Bucket:** Smooths traffic, processes at constant rate
- **Fixed Window:** Simple but can allow 2x limit at boundaries
- **Sliding Window:** More accurate, higher complexity

PostgreSQL Implementation Challenges

PostgreSQL alone is **not ideal** for rate limiting due to:

- High write contention on counter updates
- Network latency for distributed systems
- Lock contention in high-concurrency scenarios

Hybrid Architecture

Recommended: Use Redis for rate limiting logic, PostgreSQL for configuration and audit logs.

```
CREATE TABLE rate_limits (
  user_id BIGINT PRIMARY KEY,
  limit_per_hour INT NOT NULL,
  limit_per_day INT NOT NULL
);

CREATE TABLE rate_limit_violations (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT,
  endpoint VARCHAR(255),
  violated_at TIMESTAMP DEFAULT NOW()
);
```

PostgreSQL-Only Approach (Limited Scale)

For moderate traffic, use advisory locks and atomic operations:

```
UPDATE user_quotas
SET requests = requests + 1,
    last_reset = CASE
      WHEN last_reset < NOW() - INTERVAL '1 hour'
      THEN NOW() ELSE last_reset END
WHERE user_id = $1
  AND requests < limit
RETURNING requests;
```

Distributed Considerations

- **CAP Theorem:** Choose AP (availability + partition tolerance) for rate limiting - temporary over-limit is acceptable
- **Local Counters:** Each application server maintains local counters, sync periodically
- **Consensus:** Use Redis with Lua scripts for atomic operations across distributed nodes

5. Design an e-commerce inventory management system. How would you handle concurrent purchases and prevent overselling using PostgreSQL?

Core Challenge

Prevent race conditions where multiple users purchase the last item simultaneously. Must maintain **ACID properties** while handling high concurrency.

Schema Design

```
CREATE TABLE products (
  id BIGSERIAL PRIMARY KEY,
  name VARCHAR(255),
  price DECIMAL(10,2)
);

CREATE TABLE inventory (
  product_id BIGINT PRIMARY KEY,
  quantity INT NOT NULL CHECK (quantity >= 0),
  reserved INT DEFAULT 0,
  version BIGINT DEFAULT 0
);
```

Solution 1: Pessimistic Locking

```
BEGIN;
SELECT quantity FROM inventory
WHERE product_id = $1
FOR UPDATE;

UPDATE inventory
SET quantity = quantity - $2
WHERE product_id = $1
  AND quantity >= $2;
COMMIT;
```

Pros: Guarantees consistency. **Cons:** Lock contention reduces throughput.

Solution 2: Optimistic Locking

```
UPDATE inventory
SET quantity = quantity - $2,
    version = version + 1
WHERE product_id = $1
  AND quantity >= $2
  AND version = $3;
```

Check affected rows. If 0, retry transaction. **Better for high read/low write scenarios.**

Solution 3: Reservation System

Two-phase approach: reserve inventory during checkout, commit on payment success.

- Reserve: Increment reserved count, decrement available
- Commit: Decrement reserved, record sale
- Timeout: Background job releases expired reservations

Scaling Strategies

- **Sharding:** Partition inventory by product category or warehouse
- **Read Replicas:** Route inventory checks to replicas, writes to primary
- **Caching:** Cache product info (not exact quantities) to reduce database load
- **Queue System:** Use message queue for order processing, database for state persistence

6. Design a multi-tenant SaaS application database architecture. What are the PostgreSQL strategies for data isolation and performance?

Multi-Tenancy Approaches

Three primary strategies with different trade-offs:

1. Separate Database Per Tenant

- **Isolation:** Complete data and performance isolation
- **Pros:** Easiest to secure, backup, and migrate individual tenants
- **Cons:** High operational overhead, expensive for many tenants
- **Use case:** Enterprise customers with strict compliance requirements

2. Shared Database, Separate Schemas

```
CREATE SCHEMA tenant_1001;
CREATE TABLE tenant_1001.users (...);

CREATE SCHEMA tenant_1002;
CREATE TABLE tenant_1002.users (...);

SET search_path TO tenant_1001;
```

- **Pros:** Good isolation, easier management than separate databases
- **Cons:** PostgreSQL has schema limits, shared connection pool
- **Use case:** Mid-market SaaS with hundreds of tenants

3. Shared Schema with Tenant ID

```
CREATE TABLE users (
  id BIGSERIAL,
  tenant_id BIGINT NOT NULL,
  email VARCHAR(255),
  PRIMARY KEY (tenant_id, id)
) PARTITION BY LIST (tenant_id);

CREATE INDEX idx_tenant ON users(tenant_id);
```

- **Pros:** Maximum density, easiest to manage
- **Cons:** Risk of data leakage, noisy neighbor problems
- **Use case:** SMB SaaS with thousands of small tenants

Security with Row-Level Security

```
ALTER TABLE users ENABLE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation ON users
USING (tenant_id = current_setting('app.tenant_id')::BIGINT);
```

Performance Optimization

- **Partitioning:** Partition large tables by tenant_id for large tenants
- **Connection Pooling:** Use PgBouncer with session pooling per tenant
- **Resource Limits:** Use cgroups or separate read replicas for large tenants

7. Design a notification system that supports multiple channels (email, SMS, push). How would you architect the queuing and delivery tracking with PostgreSQL?

System Architecture

- **API Layer:** Receives notification requests
- **PostgreSQL:** Persistent queue and delivery tracking
- **Worker Processes:** Poll queue and send notifications
- **External Services:** SendGrid, Twilio, FCM for actual delivery
- **Redis:** Rate limiting and deduplication

Database Schema

```
CREATE TYPE notification_channel AS ENUM ('email', 'sms', 'push');
CREATE TYPE notification_status AS ENUM ('pending', 'processing', 'sent', 'failed');

CREATE TABLE notifications (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT NOT NULL,
  channel notification_channel,
  status notification_status DEFAULT 'pending',
  payload JSONB,
  scheduled_at TIMESTAMP DEFAULT NOW(),
  sent_at TIMESTAMP,
  retry_count INT DEFAULT 0
) PARTITION BY RANGE (scheduled_at);
```

Queue Implementation

Use PostgreSQL as a durable queue with SKIP LOCKED for concurrent workers:

```
BEGIN;
SELECT * FROM notifications
WHERE status = 'pending'
  AND scheduled_at <= NOW()
ORDER BY scheduled_at
LIMIT 100
FOR UPDATE SKIP LOCKED;

UPDATE notifications
SET status = 'processing'
WHERE id = ANY($1);
COMMIT;
```

Delivery Tracking

- **Idempotency:** Store external provider message IDs to prevent duplicates
- **Retry Logic:** Exponential backoff for failed deliveries
- **Dead Letter Queue:** Move permanently failed notifications to separate table
- **Webhooks:** Receive delivery status from providers, update PostgreSQL

Performance Considerations

- **Partitioning:** Range partition by scheduled_at, drop old partitions
- **Indexes:** Create partial index on (status, scheduled_at) WHERE status = 'pending'
- **Archival:** Move sent notifications older than 30 days to archive table
- **Monitoring:** Track queue depth, processing rate, failure rate

Scaling Strategy

For high volume, consider hybrid approach: PostgreSQL for configuration and audit trail, Redis or RabbitMQ for actual message queue. Use logical replication to sync delivery status back to PostgreSQL for analytics.

8. Design a content delivery and caching system. How would you use PostgreSQL alongside caching layers to optimize content delivery?

System Architecture Layers

- **CDN:** Edge caching for static assets (CloudFront, Cloudflare)
- **Application Cache:** Redis/Memcached for hot data
- **PostgreSQL:** Source of truth for content metadata
- **Object Storage:** S3 for actual content files

PostgreSQL Schema

```
CREATE TABLE content (
  id BIGSERIAL PRIMARY KEY,
  slug VARCHAR(255) UNIQUE,
  title TEXT,
  body TEXT,
  s3_key VARCHAR(512),
  cache_ttl INT DEFAULT 3600,
  version INT DEFAULT 1,
  published_at TIMESTAMP,
  updated_at TIMESTAMP DEFAULT NOW()
);
CREATE INDEX idx_slug ON content(slug);
```

Cache Strategy

Multi-Level Caching:

- **L1 - CDN:** Cache rendered pages at edge (TTL: 5-60 minutes)
- **L2 - Redis:** Cache content objects (TTL: 15 minutes)
- **L3 - PostgreSQL:** Query when cache misses

Cache Invalidation

Use PostgreSQL triggers and NOTIFY for cache invalidation:

```
CREATE FUNCTION notify_content_change()
RETURNS TRIGGER AS $$
BEGIN
  PERFORM pg_notify('content_updates',
    json_build_object('id', NEW.id, 'slug', NEW.slug)::text);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER content_change_trigger
AFTER UPDATE ON content
FOR EACH ROW EXECUTE FUNCTION notify_content_change();
```

Cache-Aside Pattern

Application logic:

1. Check Redis for content by slug
2. If miss, query PostgreSQL
3. Store in Redis with TTL
4. Return to client

Consistency Considerations

- **Versioning:** Include version number in cache keys to handle updates
- **Soft Deletes:** Mark content as deleted rather than removing immediately
- **Eventual Consistency:** Accept brief staleness in caches (CAP: AP over C)
- **Cache Warming:** Pre-populate cache for popular content using PostgreSQL analytics

Performance Optimization

- **Read Replicas:** Route cache population queries to replicas
- **Materialized Views:** Pre-aggregate popular content lists
- **Partial Indexes:** Index only published content for faster queries

9. Design a job scheduling and execution system for background tasks. How would you implement reliable job queuing with PostgreSQL?

System Requirements

- Reliable job persistence and execution
- Priority-based scheduling
- Retry logic with exponential backoff
- Concurrent worker support
- Job status tracking and observability

Database Schema

```
CREATE TYPE job_status AS ENUM ('pending', 'running', 'completed', 'failed', 'cancelled');

CREATE TABLE jobs (
  id BIGSERIAL PRIMARY KEY,
  job_type VARCHAR(100) NOT NULL,
  payload JSONB,
  priority INT DEFAULT 5,
  status job_status DEFAULT 'pending',
  scheduled_at TIMESTAMP DEFAULT NOW(),
  started_at TIMESTAMP,
  completed_at TIMESTAMP,
  retry_count INT DEFAULT 0,
  max_retries INT DEFAULT 3,
  error_message TEXT
);
```

Worker Implementation

Workers poll for jobs using SKIP LOCKED to prevent conflicts:

```
BEGIN;
SELECT * FROM jobs
WHERE status = 'pending'
  AND scheduled_at <= NOW()
ORDER BY priority DESC, scheduled_at ASC
LIMIT 1
FOR UPDATE SKIP LOCKED;

UPDATE jobs SET status = 'running', started_at = NOW()
WHERE id = $1;
COMMIT;
```

Retry Logic

Implement exponential backoff for failed jobs:

```
UPDATE jobs
SET status = 'pending',
    retry_count = retry_count + 1,
    scheduled_at = NOW() + (INTERVAL '1 minute' * POWER(2, retry_count))
WHERE id = $1
  AND retry_count < max_retries;
```

Advanced Features

- **Job Dependencies:** Add parent_job_id column, only schedule when parent completes
- **Recurring Jobs:** Use cron_expression column, reschedule after completion
- **Job Cancellation:** Workers check status before each step, abort if cancelled
- **Dead Letter Queue:** Move permanently failed jobs to separate table for investigation

Performance Optimization

- **Partitioning:** Partition by status or scheduled_at for faster queries
- **Partial Indexes:** CREATE INDEX ON jobs(priority, scheduled_at) WHERE status = 'pending'
- **Archival:** Periodically archive completed jobs older than 30 days
- **Connection Pooling:** Use PgBouncer to manage worker connections efficiently

Monitoring

Track queue depth, job latency, failure rates, and worker utilization using PostgreSQL statistics and custom metrics tables.

10. Design a real-time analytics dashboard system. How would you architect data ingestion, aggregation, and querying with PostgreSQL?

Architecture Overview

- **Data Ingestion:** High-throughput event stream (Kafka, Kinesis)

- **Stream Processing:** Real-time aggregation (Flink, Spark Streaming)
- **PostgreSQL:** Time-series storage with TimescaleDB extension
- **Materialized Views:** Pre-computed aggregations
- **Caching:** Redis for dashboard query results

Schema Design with TimescaleDB

```
CREATE EXTENSION IF NOT EXISTS timescaledb;

CREATE TABLE events (
  time TIMESTAMPTZ NOT NULL,
  user_id BIGINT,
  event_type VARCHAR(50),
  properties JSONB,
  value NUMERIC
);

SELECT create_hypertable('events', 'time');
```

Real-Time Aggregation

Use continuous aggregates for real-time materialized views:

```
CREATE MATERIALIZED VIEW hourly_metrics
WITH (timescaledb.continuous) AS
SELECT time_bucket('1 hour', time) AS hour,
       event_type,
       COUNT(*) as event_count,
       AVG(value) as avg_value
FROM events
GROUP BY hour, event_type;
```

Query Optimization

- **Time-based Partitioning:** Automatic with TimescaleDB hypertables
- **Compression:** Enable compression for older data to save storage
- **Data Retention:** Automatically drop old partitions based on retention policy
- **Indexes:** Create indexes on frequently filtered columns (user_id, event_type)

Dashboard Query Pattern

```
SELECT hour, event_type, event_count
FROM hourly_metrics
WHERE hour >= NOW() - INTERVAL '24 hours'
  AND event_type = $1
ORDER BY hour DESC;
```

Scaling Strategies

- **Write Optimization:** Batch inserts, use COPY for bulk loading
- **Read Replicas:** Route dashboard queries to dedicated read replicas
- **Distributed TimescaleDB:** Use multi-node setup for massive scale
- **Tiered Storage:** Hot data in PostgreSQL, cold data in data warehouse (Redshift, BigQuery)

CAP Theorem Considerations

Favor **Availability and Partition Tolerance**. Accept eventual consistency for real-time metrics. Use asynchronous replication and allow slight data lag in dashboards for better performance.

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a SQL query to find the second highest salary from an employees table without using LIMIT or TOP.

Solution Using Subquery

This approach uses a correlated subquery to find the second highest salary:

```
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

Key Points:

- The inner query finds the maximum salary
- The outer query finds the maximum salary that is less than the overall maximum
- Returns NULL if there's no second highest salary
- Works across all PostgreSQL versions

2. How would you debug a slow PostgreSQL query? What tools and techniques would you use?

Debugging Approach

Primary Tools:

- **EXPLAIN ANALYZE:** Shows actual execution plan with timing
- **pg_stat_statements:** Tracks query performance statistics
- **auto_explain:** Automatically logs slow query plans
- **pg_stat_activity:** Shows currently running queries

Technique Example:

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT * FROM orders WHERE customer_id = 123;
```

Key Metrics to Check:

- Sequential scans vs index scans
- High loop counts in nested loops
- Buffer hits vs reads (cache efficiency)
- Actual rows vs estimated rows (statistics accuracy)

3. Write a query to find duplicate records in a table based on multiple columns.

Finding Duplicates

Use GROUP BY with HAVING to identify duplicates:

```
SELECT email, phone, COUNT(*) as duplicate_count
FROM users
GROUP BY email, phone
HAVING COUNT(*) > 1
ORDER BY duplicate_count DESC;
```

To Get Full Duplicate Rows:

```
SELECT u.*
FROM users u
```

```
INNER JOIN (
  SELECT email, phone FROM users
  GROUP BY email, phone HAVING COUNT(*) > 1
) d ON u.email = d.email AND u.phone = d.phone;
```

4. How do you handle deadlocks in PostgreSQL? Provide an example of detecting and preventing them.

Deadlock Detection and Prevention

Detection: PostgreSQL automatically detects deadlocks and aborts one transaction with error code 40P01.

Monitoring Query:

```
SELECT blocked.pid, blocked.query, blocking.pid AS blocking_pid
FROM pg_stat_activity blocked
JOIN pg_stat_activity blocking
  ON blocking.pid = ANY(pg_blocking_pids(blocked.pid));
```

Prevention Strategies:

- **Consistent lock ordering:** Always acquire locks in the same order
- **Keep transactions short:** Minimize lock hold time
- **Use appropriate isolation levels:** READ COMMITTED when possible
- **Use SELECT FOR UPDATE NOWAIT:** Fail fast instead of waiting

5. Write a query using window functions to calculate running totals and rank employees by salary within departments.

Window Functions Solution

```
SELECT
  employee_id,
  department_id,
  salary,
  SUM(salary) OVER (PARTITION BY department_id
    ORDER BY employee_id) AS running_total,
  RANK() OVER (PARTITION BY department_id
    ORDER BY salary DESC) AS salary_rank
FROM employees;
```

Key Concepts:

- **PARTITION BY:** Creates separate windows per department
- **ORDER BY:** Defines the order for calculations
- **RANK():** Assigns rank with gaps for ties
- Use DENSE_RANK() for ranks without gaps

6. How would you optimize a query that performs poorly due to missing indexes? Show how to identify which indexes to create.

Index Optimization Process

Step 1: Analyze Query Plan

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM orders
WHERE customer_id = 100 AND status = 'pending';
```

Step 2: Check for Sequential Scans

Look for 'Seq Scan' in output - indicates missing index.

Step 3: Create Appropriate Index

```
CREATE INDEX idx_orders_customer_status
ON orders(customer_id, status);
```

Best Practices:

- Put most selective column first in composite indexes
- Use INCLUDE for covering indexes
- Monitor index usage with pg_stat_user_indexes
- Consider partial indexes for filtered queries

7. Explain VACUUM and ANALYZE in PostgreSQL. Write a query to check table bloat and when to run VACUUM.

VACUUM and ANALYZE

VACUUM: Reclaims storage from dead tuples created by UPDATE/DELETE operations due to MVCC.

ANALYZE: Updates statistics for query planner optimization.

Check Table Bloat:

```
SELECT schemaname, tablename,
  pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) AS size,
  n_dead_tup, n_live_tup,
  ROUND(n_dead_tup * 100.0 / NULLIF(n_live_tup + n_dead_tup, 0), 2) AS dead_pct
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC;
```

When to Run:

- When dead_pct > 20%
- After bulk DELETE/UPDATE operations
- Use autovacuum for automation

8. Write a recursive CTE query to traverse a hierarchical employee-manager relationship.

Recursive CTE Solution

```
WITH RECURSIVE employee_hierarchy AS (
  SELECT id, name, manager_id, 1 AS level
  FROM employees WHERE manager_id IS NULL
  UNION ALL
  SELECT e.id, e.name, e.manager_id, eh.level + 1
  FROM employees e
  JOIN employee_hierarchy eh ON e.manager_id = eh.id
)
SELECT * FROM employee_hierarchy ORDER BY level, id;
```

How It Works:

- **Base case:** Selects top-level managers (NULL manager_id)
- **Recursive case:** Joins employees to their managers
- **level:** Tracks depth in hierarchy
- Terminates when no more child records found

9. How do you handle connection pooling issues in PostgreSQL? What parameters would you tune?

Connection Pooling Management

Common Issues:

- Connection exhaustion (max_connections reached)
- Idle connections consuming resources
- Connection storms during traffic spikes

Key Parameters to Tune:

```
-- PostgreSQL configuration
max_connections = 200
idle_in_transaction_session_timeout = 60000
statement_timeout = 30000
```

Check Active Connections:

```
SELECT state, COUNT(*)
FROM pg_stat_activity
GROUP BY state;
```

Solutions:

- Use **PgBouncer** for connection pooling (transaction/session mode)
- Set appropriate pool size: (core_count * 2) + effective_spindle_count
- Implement connection timeouts in application
- Monitor with pg_stat_activity regularly

10. Write a query to pivot rows into columns and explain when you would use crosstab versus CASE statements.

Pivoting Data in PostgreSQL

Using CASE Statements:

```
SELECT
  product_id,
  SUM(CASE WHEN month = 'Jan' THEN sales ELSE 0 END) AS jan_sales,
  SUM(CASE WHEN month = 'Feb' THEN sales ELSE 0 END) AS feb_sales,
  SUM(CASE WHEN month = 'Mar' THEN sales ELSE 0 END) AS mar_sales
FROM monthly_sales
GROUP BY product_id;
```

Using crosstab (requires tablefunc extension):

```
SELECT * FROM crosstab(
  'SELECT product_id, month, sales FROM monthly_sales ORDER BY 1,2'
) AS ct(product_id int, jan int, feb int, mar int);
```

When to Use Each:

- **CASE:** Known, fixed number of columns; simpler syntax
- **crosstab:** Dynamic columns; cleaner for many pivots

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you optimized a slow PostgreSQL query that was impacting production performance.

Situation: Our e-commerce platform was experiencing page load times exceeding 8 seconds during checkout, with customer complaints increasing by 40%.

Task: I was assigned to identify and resolve the database bottleneck causing the slowdown within 48 hours.

Action: I used pg_stat_statements to identify the problematic query, which was performing a sequential scan on a 10M row orders table. I analyzed the execution plan using EXPLAIN ANALYZE, created a composite index on (user_id, created_at, status), and rewrote the query to eliminate unnecessary JOINs. I also implemented connection pooling with PgBouncer to reduce connection overhead.

Result: Query execution time dropped from 6.2 seconds to 180ms, page load times decreased to under 2 seconds, and we saw a 15% increase in checkout completion rates within the first week.

2. Describe a situation where you had to handle a PostgreSQL database failure or data corruption issue.

Situation: At 3 AM, our monitoring system alerted us that the primary PostgreSQL database became unresponsive, and the application was returning 500 errors to all users.

Task: As the on-call database engineer, I needed to restore service immediately and prevent data loss while investigating the root cause.

Action: I first promoted the streaming replication standby to primary to restore service within 5 minutes. Then I investigated the failed primary and discovered disk corruption in the WAL directory. I initiated point-in-time recovery (PITR) from our base backup and replayed WAL archives. I also implemented automated health checks using pg_isready and set up synchronous replication for critical transactions.

Result: Service was restored with zero data loss, downtime was limited to 5 minutes, and I documented the incident with a runbook. The automated monitoring prevented two subsequent failures from impacting users.

3. Can you share an experience where you had to migrate a large PostgreSQL database with minimal downtime?

Situation: Our company needed to migrate a 2TB PostgreSQL database from on-premises infrastructure to AWS RDS with a maximum downtime window of 15 minutes during a planned maintenance.

Task: I was responsible for planning and executing the migration strategy while ensuring data consistency and minimal business disruption.

Action: I implemented a multi-phase approach: set up logical replication using pglogical to continuously sync data to the new RDS instance, ran parallel data validation checks using custom scripts, performed multiple dry-run cutover tests, and coordinated with application teams to implement database connection failover logic. During the cutover window, I stopped writes, verified replication lag was zero, and switched DNS to point to the new instance.

Result: The migration completed with only 12 minutes of downtime, zero data loss, and no post-migration issues. The new RDS setup provided 40% better query performance and reduced operational overhead by eliminating manual backup management.

**4. Tell me about a time when you had to make a difficult decision regarding database

schema design or refactoring.

Situation: Our analytics team requested denormalized tables for faster reporting, but this conflicted with our normalized schema design principles and would introduce data redundancy across 15 tables.

Task: I needed to balance performance requirements with data integrity concerns while considering long-term maintainability.

Action: I conducted a thorough analysis comparing multiple approaches: materialized views, separate OLAP database, and partial denormalization. I created POC implementations for each solution and benchmarked them with production-like data volumes. I presented findings to stakeholders showing that materialized views with CONCURRENTLY refresh provided 85% of the performance benefit without schema changes. I also implemented partitioning on time-series data and set up automated refresh schedules during low-traffic periods.

Result: Report generation time decreased from 45 seconds to 4 seconds, we maintained schema integrity, and the solution was easier to maintain. The approach became our standard pattern for similar requirements across other teams.

5. Describe a situation where you had to troubleshoot and resolve PostgreSQL connection pooling or concurrency issues.

Situation: Our application started experiencing intermittent "connection pool exhausted" errors during peak traffic, causing 503 errors for 20% of requests despite having 200 max connections configured.

Task: I needed to identify why connections were being exhausted and implement a sustainable solution without simply increasing connection limits.

Action: I analyzed pg_stat_activity and discovered that long-running transactions were holding connections for minutes due to unoptimized ORM queries. I implemented statement_timeout and idle_in_transaction_session_timeout at the database level, configured PgBouncer in transaction pooling mode to multiplex connections more efficiently, and worked with developers to fix N+1 query problems in the application code. I also added monitoring dashboards for connection metrics.

Result: Connection pool exhaustion errors dropped to zero, we reduced max_connections from 200 to 100 while handling 3x more traffic, and average connection hold time decreased from 8 seconds to 400ms. Database CPU utilization dropped by 35%.

6. Share an experience where you implemented PostgreSQL security best practices or resolved a security vulnerability.

Situation: During a security audit, we discovered that our PostgreSQL databases had overly permissive access controls, with multiple applications using the superuser account and no encryption for data at rest.

Task: I was tasked with implementing comprehensive security hardening across 12 production databases within one month without disrupting services.

Action: I created a phased implementation plan: established role-based access control (RBAC) with specific privileges for each application using GRANT statements, implemented SSL/TLS for all client connections, enabled transparent data encryption using pgcrypto for sensitive columns, configured pg_hba.conf to restrict access by IP and require certificate authentication, and set up audit logging with pgaudit to track all DDL and sensitive DML operations. I also conducted training sessions for the development team.

Result: We passed the follow-up security audit with zero critical findings, eliminated superuser access from applications, and achieved compliance with SOC 2 requirements. The audit trail helped us quickly investigate a suspected data access incident, which turned out to be a false alarm.

7. Tell me about a time when you had to balance competing priorities between database performance and cost optimization.

Situation: Our AWS RDS PostgreSQL costs were increasing 25% month-over-month, reaching $18K monthly, while the business was pushing for cost reduction, but users were also complaining about slow dashboard load times.

Task: I needed to reduce database costs by at least 30% while maintaining or improving query performance.

Action: I conducted a comprehensive analysis using Performance Insights and identified that 60% of queries were running against historical data. I implemented a tiered storage strategy: kept recent 90 days on the primary RDS instance, moved older data to partitioned tables, and archived data beyond 1 year to S3 using pg_dump with custom scripts. I also right-sized the RDS instance from db.r5.4xlarge to db.r5.2xlarge after optimizing poorly performing queries and implementing Redis caching for frequently accessed data.

Result: Monthly costs decreased from $18K to $11K (39% reduction), query performance for recent data improved by 45% due to smaller table sizes, and dashboard load times decreased from 12 seconds to 3 seconds. The solution scaled well as data volumes continued to grow.

8. Describe a time when you had to mentor or guide junior developers on PostgreSQL best practices.

Situation: Our team hired three junior developers who were creating database migrations without proper review, resulting in production incidents including table locks during peak hours and missing indexes causing performance degradation.

Task: I was asked to establish a mentorship program and create guidelines to prevent future database-related incidents.

Action: I developed a comprehensive PostgreSQL best practices guide covering migration safety, indexing strategies, and query optimization. I implemented mandatory code reviews for all database changes and created a staging environment that mirrored production data volumes for testing. I conducted weekly knowledge-sharing sessions covering topics like EXPLAIN plans, transaction isolation levels, and VACUUM strategies. I also created SQL templates and linting rules to catch common mistakes before deployment.

Result: Database-related production incidents decreased by 90% over three months, junior developers became confident in writing efficient queries, and two of them successfully handled complex schema migrations independently. The documentation became a standard onboarding resource for all new engineering hires.

9. Tell me about a challenging PostgreSQL upgrade or version migration you managed.

Situation: Our production databases were running PostgreSQL 9.6, which was approaching end-of-life, and we needed to upgrade to PostgreSQL 13 to access performance improvements and new features like parallel query enhancements.

Task: I was responsible for planning and executing the upgrade across 8 production databases totaling 5TB of data with minimal risk and downtime.

Action: I created a detailed upgrade plan including compatibility testing using pg_upgrade --check, identified deprecated features in our codebase, and set up a staging environment running PostgreSQL 13 for extensive testing. I used logical replication to minimize downtime by keeping the old and new versions in sync, validated application compatibility through automated test suites, and prepared rollback procedures. I scheduled the cutover during a low-traffic window and had the entire team on standby.

Result: The upgrade completed successfully with only 8 minutes of downtime, we immediately benefited from 25% faster query performance on complex analytical queries, and encountered zero post-upgrade issues. The process documentation I created was reused for upgrading our development and staging environments.

10. Describe a situation where you had to work with cross-functional teams to resolve a complex database-related problem.

Situation: Our application was experiencing random transaction deadlocks during order processing, causing failed checkouts and requiring manual intervention. The issue involved complex interactions between the application layer, message queue, and database.

Task: I needed to collaborate with backend developers, DevOps, and QA teams to identify the root cause and implement a comprehensive solution.

Action: I organized daily sync meetings and set up shared monitoring dashboards. I enabled deadlock logging in PostgreSQL and analyzed pg_stat_database to identify lock contention patterns. Working with developers, I discovered that concurrent updates to inventory and orders tables were happening in different transaction orders. I proposed implementing explicit row-level locking using SELECT FOR UPDATE with consistent ordering, and worked with the QA team to create load tests that reproduced the deadlock scenarios. I also implemented retry logic with exponential backoff at the application level.

Result: Deadlock incidents dropped from 15-20 per day to zero, checkout success rate improved from 94% to 99.7%, and the collaborative approach strengthened relationships between teams. We documented the locking strategy as a standard pattern for similar concurrent update scenarios.