PostgreSQL Performance and Query Tuning Interview Questions & Answers

Query Analysis & Execution Plans

Q1: How do you analyze query performance in PostgreSQL?

Answer: Use `EXPLAIN` and `EXPLAIN ANALYZE` to understand query execution plans:

```sql
-- Shows the planned execution without running the query
EXPLAIN SELECT * FROM users WHERE age > 25;

-- Actually executes and shows real performance metrics
EXPLAIN ANALYZE SELECT * FROM users WHERE age > 25;

-- More detailed output with buffers, timing, and costs
EXPLAIN (ANALYZE, BUFFERS, TIMING) SELECT * FROM users WHERE age > 25;
```

Key metrics to examine:

- **Cost**: Estimated expense (startup cost..total cost)
- **Rows**: Estimated vs actual rows returned
- **Width**: Average row size in bytes
- **Actual Time**: Real execution time per node
- **Buffers**: Shared/local/temp buffers hit/read/written

Q2: What are the different types of scans in PostgreSQL and when are they used?

Answer:

1. **Sequential Scan (Seq Scan)**: Reads entire table row by row
   - Used when no suitable index exists or small tables
   - Cost: O(n) where n is table size
2. **Index Scan**: Uses index to locate specific rows

- Efficient for selective queries
- Cost: O(log n) for lookup + cost of fetching rows

3. **Index Only Scan**: Data retrieved entirely from index
   - Possible when all required columns are in the index
   - Fastest for covered queries
4. **Bitmap Index Scan**: Creates bitmap of matching rows
   - Used for multiple index conditions or low selectivity
   - Combines multiple indexes efficiently

```sql
-- Example forcing different scan types
SET enable_seqscan = off;  -- Force index usage
SET enable_indexscan = off;  -- Force bitmap scan
```

Q3: How do you interpret and optimize join operations?

Answer: PostgreSQL uses three main join algorithms:

1. **Nested Loop Join**:
   - Best for small datasets or when one side is very selective
   - Cost: O(n*m)
   - Often used with indexes
2. **Hash Join**:
   - Good for medium to large datasets
   - Builds hash table from smaller relation
   - Cost: O(n+m)
3. **Merge Join**:
   - Efficient for pre-sorted data
   - Requires both inputs to be sorted
   - Cost: O(n log n + m log m) if sorting needed

Optimization strategies:

```sql
-- Ensure join columns have indexes
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
CREATE INDEX idx_customers_id ON customers(id);

-- Use appropriate data types (avoid implicit conversions)
-- Correct
WHERE customer_id = 123

-- Avoid
WHERE customer_id = '123'  -- if customer_id is integer
```

Indexing Strategies

Q4: What are the different types of indexes in PostgreSQL and when should you use each?

Answer:

1. **B-Tree Index (Default)**:

   ```sql
   CREATE INDEX idx_name ON table(column);
   ```

   - Best for equality and range queries
   - Supports sorting operations
   - Use for: =, <, <=, >, >=, BETWEEN, IN, IS NULL

2. **Hash Index**:

   ```sql
   CREATE INDEX idx_hash ON table USING HASH(column);
   ```

   - Only equality operations (=)
   - Slightly faster than B-tree for equality
   - Not WAL-logged (use with caution)

3. **GIN (Generalized Inverted Index)**:

```sql
CREATE INDEX idx_gin ON table USING GIN(column);
```

- For composite values (arrays, JSONB, full-text search)
- Excellent for "contains" operations
- Use for: @>, <@, &&, ?

4. **GiST (Generalized Search Tree)**:

```sql
CREATE INDEX idx_gist ON table USING GIST(column);
```

- For geometric data, full-text search
- Supports custom data types
- Less precise than GIN but smaller

5. **BRIN (Block Range Index)**:

```sql
CREATE INDEX idx_brin ON table USING BRIN(column);
```

- For very large tables with natural ordering
- Minimal storage overhead
- Best for time-series or sequential data

6. **SP-GiST (Space-Partitioned GiST)**:

```sql
CREATE INDEX idx_spgist ON table USING SPGIST(column);
```

- For clustered data with uneven distribution
- Good for text, IP addresses, geographic data

Q5: How do you create and optimize partial indexes?

Answer: Partial indexes include only rows meeting specified conditions:

```sql
```

```
-- Index only active users
CREATE INDEX idx_active_users ON users(email)
WHERE status = 'active';

-- Index only recent orders
CREATE INDEX idx_recent_orders ON orders(created_at)
WHERE created_at >= '2024-01-01';

-- Index only non-null values
CREATE INDEX idx_phone ON users(phone_number)
WHERE phone_number IS NOT NULL;
```

Benefits:

- Smaller index size
- Faster maintenance
- Reduced storage requirements
- Better performance for specific query patterns

Best practices:

- Use when querying a subset of data frequently
- Ensure WHERE clause in queries matches index condition
- Monitor query plans to verify index usage

Q6: Explain composite indexes and column ordering strategy.

Answer: Composite indexes span multiple columns. Column order significantly affects performance:

```sql
-- Create composite index
CREATE INDEX idx_user_search ON users(status, created_at, email);
```

Column Ordering Rules:

1. **Most Selective First**: Place columns with highest cardinality first

2. **Equality Before Range**: Put equality conditions before range conditions
3. **Query Pattern Matching**: Order based on most common query patterns

```sql
-- Good: equality first, then range
CREATE INDEX idx_orders ON orders(status, customer_id, created_at);

-- Query that uses this index efficiently
SELECT * FROM orders
WHERE status = 'pending'
  AND customer_id = 123
  AND created_at >= '2024-01-01';
```

Index Usage Rules:

- PostgreSQL can use leftmost columns of composite index
- Cannot skip columns in the middle efficiently
- Range conditions stop further column usage

Performance Monitoring

Q7: What are the key performance metrics to monitor in PostgreSQL?

Answer:

System-Level Metrics:

```sql
```

```sql
-- Database connections
SELECT count(*) as active_connections
FROM pg_stat_activity
WHERE state = 'active';

-- Cache hit ratio (should be >99%)
SELECT
  sum(heap_blks_hit) / (sum(heap_blks_hit) + sum(heap_blks_read)) as cache_hit_ratio
FROM pg_statio_user_tables;

-- Index usage ratio
SELECT
  schemaname,
  tablename,
  round(100.0 * idx_scan / (seq_scan + idx_scan), 2) as index_usage_ratio
FROM pg_stat_user_tables
WHERE seq_scan + idx_scan > 0;
```

Query-Level Metrics:

```
sql
```

```sql
-- Top slow queries (requires pg_stat_statements)
SELECT
  query,
  calls,
  total_time,
  mean_time,
  rows
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;

-- Lock monitoring
SELECT
  blocked_locks.pid AS blocked_pid,
  blocked_activity.usename AS blocked_user,
  blocking_locks.pid AS blocking_pid,
  blocking_activity.usename AS blocking_user,
  blocked_activity.query AS blocked_statement
FROM pg_catalog.pg_locks blocked_locks
JOIN pg_catalog.pg_stat_activity blocked_activity ON blocked_activity.pid = blocked_locks.pid
JOIN pg_catalog.pg_locks blocking_locks ON blocking_locks.locktype = blocked_locks.locktype
JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid = blocking_locks.pid
WHERE NOT blocked_locks.granted;
```

Q8: How do you identify and resolve common performance bottlenecks?

Answer:

1. I/O Bottlenecks:

```sql
```

```
-- Check table and index 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
FROM pg_tables
ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC;

-- Solutions:
-- - Add appropriate indexes
-- - Partition large tables
-- - Archive old data
-- - Increase shared_buffers
```

2. CPU Bottlenecks:

```sql
-- Identify expensive queries
SELECT
  query,
  calls,
  total_time / calls as avg_time,
  rows / calls as avg_rows
FROM pg_stat_statements
WHERE calls > 100
ORDER BY total_time / calls DESC;

-- Solutions:
-- - Optimize query logic
-- - Add indexes
-- - Rewrite complex queries
-- - Use materialized views
```

3. Memory Issues:

```sql
```

```sql
-- Check work_mem usage
SHOW work_mem;

-- Monitor temp file usage
SELECT
  schemaname,
  tablename,
  n_tup_ins,
  n_tup_upd,
  n_tup_del
FROM pg_stat_user_tables;

-- Solutions:
-- - Increase work_mem for complex queries
-- - Optimize sort operations
-- - Use streaming replication for read replicas
```

Configuration Tuning

Q9: What are the most important PostgreSQL configuration parameters for performance?

Answer:

Memory Settings:

```bash
```

```
# postgresql.conf

# Shared memory for caching (25% of RAM)
shared_buffers = '4GB'

# Memory for sorting and hash operations
work_mem = '64MB'

# Memory for maintenance operations
maintenance_work_mem = '1GB'

# Memory for WAL buffers
wal_buffers = '64MB'

# Effective cache size (50-75% of RAM)
effective_cache_size = '12GB'
```

I/O and Checkpoints:

```bash
# WAL settings for performance
wal_level = replica
checkpoint_timeout = '15min'
checkpoint_completion_target = 0.9
max_wal_size = '4GB'
min_wal_size = '1GB'

# Background writer settings
bgwriter_delay = '200ms'
bgwriter_lru_maxpages = 100
bgwriter_lru_multiplier = 2.0
```

Query Planner:

```bash
```

```
# Cost parameters (tune based on hardware)
seq_page_cost = 1.0
random_page_cost = 2.0  # Lower for SSDs (1.1-1.5)
cpu_tuple_cost = 0.01
cpu_index_tuple_cost = 0.005
cpu_operator_cost = 0.0025

# Statistics target for better estimates
default_statistics_target = 100
```

Q10: How do you optimize PostgreSQL for different workload types?

Answer:

OLTP Workloads (High Concurrency, Short Transactions):

```bash
# Optimize for many concurrent connections
max_connections = 200
shared_buffers = '2GB'        # Lower ratio (15-25% of RAM)
work_mem = '16MB'             # Smaller per-connection
effective_cache_size = '12GB'

# Fast commit settings
synchronous_commit = on
wal_buffers = '64MB'
checkpoint_timeout = '5min'

# Connection pooling recommended (PgBouncer)
```

OLAP Workloads (Analytics, Large Queries):

```bash
```

```
# Optimize for complex queries
shared_buffers = '8GB'        # Higher ratio (25-40% of RAM)
work_mem = '256MB'            # Larger for sorting/hashing
maintenance_work_mem = '2GB'
effective_cache_size = '24GB'

# Allow longer operations
statement_timeout = '30min'
lock_timeout = '10min'

# Parallel query settings
max_parallel_workers = 8
max_parallel_workers_per_gather = 4
```

Mixed Workloads:

```bash
# Balanced configuration
shared_buffers = '4GB'
work_mem = '64MB'
effective_cache_size = '16GB'

# Use connection pooling
# Separate read replicas for analytics
# Partition large tables by time/range
```

Advanced Optimization Techniques

Q11: Explain table partitioning strategies and their performance benefits.

Answer:

Range Partitioning (Most Common):

```sql
```

```sql
-- Parent table
CREATE TABLE sales (
    id SERIAL PRIMARY KEY,
    sale_date DATE NOT NULL,
    amount DECIMAL,
    customer_id INTEGER
) PARTITION BY RANGE (sale_date);

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

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

-- Indexes on partitions
CREATE INDEX idx_sales_2024_customer ON sales_2024(customer_id);
```

Hash Partitioning:

```sql
-- For distributing data evenly
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255),
    created_at TIMESTAMP
) PARTITION BY HASH (id);

-- Create 4 hash partitions
CREATE TABLE users_0 PARTITION OF users FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE users_1 PARTITION OF users FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE users_2 PARTITION OF users FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE users_3 PARTITION OF users FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Benefits:

- Partition pruning: Only scan relevant partitions
- Parallel processing: Operations can run on multiple partitions

- Maintenance efficiency: Operations on individual partitions
- Improved query performance for time-based queries

Q12: What are materialized views and when should you use them?

Answer:

Materialized views store query results physically and can be refreshed:

```sql
-- Create materialized view for expensive aggregation
CREATE MATERIALIZED VIEW monthly_sales_summary AS
SELECT
    DATE_TRUNC('month', sale_date) as month,
    COUNT(*) as total_sales,
    SUM(amount) as total_revenue,
    AVG(amount) as avg_sale_amount
FROM sales
GROUP BY DATE_TRUNC('month', sale_date);

-- Create index on materialized view
CREATE INDEX idx_monthly_sales_month ON monthly_sales_summary(month);

-- Refresh strategies
REFRESH MATERIALIZED VIEW monthly_sales_summary;                -- Blocking
REFRESH MATERIALIZED VIEW CONCURRENTLY monthly_sales_summary;   -- Non-blocking (needs unique
```

Use Cases:

- Complex aggregations that don't change frequently
- Joining multiple large tables
- Dashboard and reporting queries
- Data warehouse scenarios

Best Practices:

- Create unique indexes for concurrent refresh
- Schedule refreshes during low-traffic periods

- Monitor refresh times and storage overhead
- Consider incremental refresh strategies

Troubleshooting Performance Issues

Q13: How do you diagnose and fix slow queries?

Answer:

Step 1: Identify Slow Queries

```sql
-- Enable query logging in postgresql.conf
log_min_duration_statement = 1000  -- Log queries > 1 second

-- Or use pg_stat_statements
SELECT
    query,
    calls,
    total_time,
    mean_time,
    stddev_time,
    rows
FROM pg_stat_statements
ORDER BY mean_time DESC
LIMIT 10;
```

Step 2: Analyze Execution Plan

```sql
-- Get detailed execution plan
EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
SELECT u.name, 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;
```

Step 3: Common Fixes

1. **Missing Indexes:**

```sql
-- Add index for WHERE clause
CREATE INDEX idx_users_created_at ON users(created_at);

-- Add index for JOIN
CREATE INDEX idx_orders_user_id ON orders(user_id);
```

2. **Inefficient JOINs:**

```sql
-- Bad: N+1 query pattern
-- Fix with proper JOIN and batch processing

-- Bad: SELECT * with large result sets
-- Fix: SELECT only needed columns
SELECT u.name, COUNT(o.id) -- Instead of SELECT *
```

3. **Suboptimal WHERE Clauses:**

```sql
-- Bad: Function calls prevent index usage
WHERE UPPER(name) = 'JOHN'

-- Good: Use functional index or case-insensitive comparison
CREATE INDEX idx_name_upper ON users(UPPER(name));
-- OR
WHERE name ILIKE 'john'
```

Q14: How do you handle deadlock situations?

Answer:

Understanding Deadlocks: Deadlocks occur when two or more transactions wait for each other to release locks.

Detection and Monitoring:

```sql
-- Check current locks
SELECT
    t.schemaname,
    t.tablename,
    l.locktype,
    l.mode,
    l.granted,
    a.query,
    a.query_start,
    age(now(), a.query_start) AS duration
FROM pg_locks l
JOIN pg_stat_activity a ON l.pid = a.pid
JOIN pg_tables t ON l.relation = t.schemaname||'.'||t.tablename::regclass
ORDER BY a.query_start;

-- Deadlock detection settings
deadlock_timeout = '1s'  -- How long to wait before checking for deadlock
```

Prevention Strategies:

1. **Consistent Lock Ordering:**

```sql
-- Always acquire locks in the same order across transactions
BEGIN;
LOCK TABLE users IN SHARE MODE;
LOCK TABLE orders IN SHARE MODE;
-- ... perform operations
COMMIT;
```

2. **Shorter Transactions:**

```sql
```

```
-- Keep transactions brief and avoid long-running operations
BEGIN;
-- Quick operations only
UPDATE users SET last_login = NOW() WHERE id = 123;
COMMIT;
```

3. **Appropriate Isolation Levels:**

```sql
-- Use appropriate isolation level
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; -- Default, usually sufficient
-- Avoid SERIALIZABLE unless necessary
```

Resolution:

- PostgreSQL automatically detects and resolves deadlocks
- One transaction is aborted (deadlock victim)
- Application should retry the aborted transaction
- Monitor deadlock frequency and patterns

Best Practices Summary

Q15: What are the top 10 PostgreSQL performance best practices?

Answer:

1. **Index Strategy:**
    - Create indexes on frequently queried columns
    - Use composite indexes for multi-column queries
    - Monitor and remove unused indexes
    - Use partial indexes for subset queries
2. **Query Optimization:**
    - Use EXPLAIN ANALYZE to understand query plans
    - Avoid SELECT * in application code
    - Use appropriate WHERE clauses with indexed columns

- Optimize JOINs with proper indexing

3. **Configuration Tuning:**
    - Set shared_buffers to 25% of RAM
    - Configure work_mem based on concurrent users
    - Set effective_cache_size to 50-75% of RAM
    - Tune checkpoint settings for workload
4. **Table Design:**
    - Use appropriate data types (smaller is better)
    - Normalize appropriately (avoid over-normalization)
    - Consider partitioning for large tables
    - Use constraints to help query planner
5. **Maintenance:**
    - Regular VACUUM and ANALYZE operations
    - Monitor table bloat and reindex when needed
    - Update table statistics regularly
    - Archive old data to keep tables manageable
6. **Connection Management:**
    - Use connection pooling (PgBouncer, pgpool)
    - Set appropriate max_connections
    - Monitor connection usage
    - Close connections properly in applications
7. **Hardware Optimization:**
    - Use SSDs for better random I/O
    - Separate WAL files on different disks
    - Ensure adequate RAM for caching
    - Configure OS-level parameters
8. **Monitoring:**
    - Enable pg_stat_statements extension
    - Monitor slow query log

- Track key performance metrics
- Set up alerting for performance degradation

9. **Backup and Recovery:**
    - Use streaming replication for high availability
    - Configure WAL archiving properly
    - Test recovery procedures regularly
    - Monitor replication lag
10. **Application-Level Optimization:**
    - Implement proper error handling and retries
    - Use batch operations for bulk data changes
    - Cache frequently accessed data at application level
    - Design database schema with query patterns in mind