PostgreSQL Coding Challenges

Interview Questions and Answers

Core Concepts

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

1. Write a query to pivot dynamic columns based on distinct category values

Solution:

```
SELECT *
FROM crosstab(
  'SELECT row_name, category, value
   FROM source_table
   ORDER BY 1,2',
  'SELECT DISTINCT category
   FROM source_table ORDER BY 1'
) AS final_result(row_name text, cat1 int, cat2 int);
```

2. Create a materialized view with concurrent refresh capability

Solution:

```
CREATE MATERIALIZED VIEW mv_sales_summary
WITH (concurrent_refresh = true) AS
SELECT date_trunc('month', sale_date) as month,
       sum(amount) as total
FROM sales
GROUP BY 1;
```

3. Implement a custom aggregate function to calculate median

Solution:

```
CREATE AGGREGATE median(numeric) (
  SFUNC=array_append,
  STYPE=numeric[],
  FINALFUNC=array_median,
  INITCOND='{}'
);
```

4. Write a query to handle slowly changing dimensions (SCD) Type 2

Solution:

```
INSERT INTO dim_customer (
  SELECT *, current_timestamp, '9999-12-31'
  FROM staged_customers s
  WHERE NOT EXISTS (
    SELECT 1 FROM dim_customer d
    WHERE d.customer_id = s.customer_id
    AND d.valid_to = '9999-12-31'
));
```

5. Create a function to generate a UUID-based sequential ID

Solution:

```
CREATE OR REPLACE FUNCTION generate_uuid_seq()
RETURNS uuid AS $$
DECLARE
```

```
  seq_id bigint;
BEGIN
  seq_id := nextval('global_id_seq');
  RETURN uuid_in(overlay(md5(seq_id::text) placing '4' from 13))::uuid;
END;
$$ LANGUAGE plpgsql;
```

6. Implement a query to find the longest streak of consecutive events

Solution:

```
WITH groups AS (
  SELECT event_date,
        event_date - ROW_NUMBER() OVER (ORDER BY event_date) AS grp
  FROM events
)
SELECT COUNT(*) as streak
FROM groups GROUP BY grp ORDER BY 1 DESC LIMIT 1;
```

7. Implement a recursive CTE to generate a hierarchical view of employees and their managers

Solution:

```
WITH RECURSIVE emp_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, h.level + 1
  FROM employees e JOIN emp_hierarchy h
  ON e.manager_id = h.id
) SELECT * FROM emp_hierarchy;
```

8. Write a query to find running totals of sales by date with a 7-day moving average

Solution:

```
SELECT
  date,
  amount,
  SUM(amount) OVER (ORDER BY date),
  AVG(amount) OVER (
    ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  ) as moving_avg
FROM sales;
```

9. Create a function to generate and maintain an audit trail for table changes

Solution:

```
CREATE OR REPLACE FUNCTION audit_changes()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO audit_log(table_name, action, changed_by)
  VALUES (TG_TABLE_NAME, TG_OP, current_user);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

10. Implement a query to find gaps in a sequence of consecutive numbers

Solution:

```
SELECT n + 1 as gap_start, next_n - 1 as gap_end
FROM (
  SELECT n, LEAD(n) OVER (ORDER BY n) as next_n
  FROM numbers
```

```
) t WHERE next_n - n > 1;
```

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 Cache using PostgreSQL?

LRU Cache Implementation

We can implement an LRU Cache using a combination of a table and triggers:

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

To maintain the LRU property, create a trigger that removes oldest entries when cache size exceeds limit:

```
CREATE OR REPLACE FUNCTION maintain_cache_size()
RETURNS TRIGGER AS $$
BEGIN
  DELETE FROM lru_cache WHERE key IN (
    SELECT key FROM lru_cache ORDER BY last_accessed
    LIMIT 1
  );
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

2. Implement a sliding window function to calculate moving averages over time series data

Sliding Window Implementation

Here's how to calculate a 7-day moving average:

```
SELECT date,
  AVG(value) OVER (
    ORDER BY date
    ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  ) as moving_avg
FROM time_series
ORDER BY date;
```

Time Complexity: O(n) where n is the number of rows in the window

3. How would you implement a stack data structure in PostgreSQL?

Stack Implementation

```
CREATE TABLE stack (
  id SERIAL,
  value TEXT,
  push_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Basic Operations:

- Push: INSERT INTO stack(value) VALUES ('item');

- Pop: DELETE FROM stack WHERE id = (SELECT MAX(id) FROM stack) RETURNING value;
- Peek: SELECT value FROM stack ORDER BY id DESC LIMIT 1;

4. Implement a solution to find pairs of numbers that sum to a target value

Pair Sum Implementation

```
WITH numbers AS (SELECT unnest(ARRAY[1,2,3,4,5,6]) AS num)
SELECT a.num, b.num
FROM numbers a
JOIN numbers b ON a.num + b.num = 10
WHERE a.num < b.num;
```

Time Complexity: O(n²) in worst case, but PostgreSQL's join optimization can improve performance significantly.

5. How would you implement a priority queue in PostgreSQL?

Priority Queue Implementation

```
CREATE TABLE priority_queue (
  id SERIAL,
  item TEXT,
  priority INTEGER,
  insert_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Operations:

- Enqueue: INSERT INTO priority_queue(item, priority) VALUES ('task', 1);
- Dequeue: DELETE FROM priority_queue WHERE id = (SELECT id FROM priority_queue ORDER BY priority, insert_time LIMIT 1) RETURNING *;

6. Implement a graph traversal algorithm using recursive CTEs

Graph Traversal Implementation

```
WITH RECURSIVE traverse AS (
  SELECT node_id, 1 AS level FROM graph WHERE node_id = 1
  UNION ALL
  SELECT g.node_id, t.level + 1
  FROM traverse t JOIN graph g ON t.node_id = g.parent_id
  WHERE t.level < 5
) SELECT * FROM traverse;
```

Note: This implements a breadth-first traversal limited to 5 levels deep

7. How would you implement a binary search tree traversal in PostgreSQL?

BST Traversal Implementation

```
WITH RECURSIVE tree_traverse AS (
  SELECT id, value, parent_id, ARRAY[value] as path
  FROM binary_tree WHERE parent_id IS NULL
  UNION ALL
  SELECT b.id, b.value, b.parent_id, t.path || b.value
  FROM tree_traverse t JOIN binary_tree b ON t.id = b.parent_id
) SELECT * FROM tree_traverse;
```

Time Complexity: O(n) where n is number of nodes

8. Implement a solution to find the median in a streaming data scenario

Streaming Median Implementation

```
SELECT percentile_cont(0.5) WITHIN GROUP (
  ORDER BY value
) as median,
```

```
percentile_cont(0.5) WITHIN GROUP (
  ORDER BY value) OVER (
    ROWS BETWEEN 99 PRECEDING AND CURRENT ROW
  ) as rolling_median
FROM stream_data;
```

Note: This maintains a rolling median over the last 100 rows

9. How would you implement a trie (prefix tree) structure in PostgreSQL?

Trie Implementation

```
CREATE TABLE trie_node (
  id SERIAL PRIMARY KEY,
  character CHAR(1),
  parent_id INTEGER REFERENCES trie_node(id),
  is_end BOOLEAN DEFAULT false
);
```

Search Implementation:

```
WITH RECURSIVE word_search AS (
  SELECT id, character, ARRAY[character] as word
  FROM trie_node WHERE parent_id IS NULL
  UNION ALL
  SELECT t.id, t.character, w.word || t.character
  FROM word_search w JOIN trie_node t ON w.id = t.parent_id
) SELECT array_to_string(word, '') FROM word_search;
```

10. Implement a solution for finding the longest increasing subsequence

Longest Increasing Subsequence

```
WITH RECURSIVE lis AS (
  SELECT value, ARRAY[value] as seq
  FROM numbers WHERE id = 1
  UNION ALL
  SELECT n.value,
    CASE WHEN n.value > ALL(l.seq)
    THEN l.seq || n.value ELSE l.seq END
  FROM lis l JOIN numbers n ON n.id > array_length(l.seq, 1)
) SELECT seq FROM lis ORDER BY array_length(seq, 1) DESC LIMIT 1;
```

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 using PostgreSQL. What schema and architecture would you use?

Key Components:

- **Database Schema**: URLs table with shortened key and original URL
- **Cache Layer**: Redis for frequently accessed URLs
- **Load Balancer**: Nginx for distribution

Schema Design:

```
CREATE TABLE urls (
  id SERIAL PRIMARY KEY,
  short_key VARCHAR(8) UNIQUE,
  original_url TEXT NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  access_count BIGINT DEFAULT 0
);
```

Key Considerations:

- Base62 encoding for short URLs
- Distributed counter for ID generation
- Write-through cache strategy
- Horizontal scaling with read replicas

2. How would you design a real-time notification system using PostgreSQL's LISTEN/NOTIFY feature?

Architecture Components:

- **PostgreSQL LISTEN/NOTIFY** for pub/sub
- **WebSocket server** for client connections
- **Message queue** for reliability

```
CREATE TABLE notifications (
  id SERIAL PRIMARY KEY,
  user_id INTEGER,
  message TEXT,
  created_at TIMESTAMP DEFAULT NOW()
);
```

Implementation:

- Trigger-based notification dispatch
- Connection pooling with pgBouncer
- Message buffering for offline users
- Horizontal scaling with partitioning

3. Design a social media feed system with PostgreSQL. How would you handle scaling reads and writes?

Core Components:

- **Post Storage**: Partitioned by user_id

- **Feed Assembly**: Materialized views
- **Caching Strategy**: Redis for hot posts

```
CREATE TABLE posts (
  id BIGINT PRIMARY KEY,
  user_id INTEGER,
  content TEXT,
  created_at TIMESTAMP
) PARTITION BY HASH (user_id);
```

Scaling Strategies:

- Fan-out on write for popular users
- Lazy loading for inactive users
- Content delivery network integration
- Read replicas for heavy query load

4. How would you implement a distributed rate limiter using PostgreSQL?

Design Approach:

- **Token Bucket Algorithm** implementation
- **Distributed coordination** via advisory locks
- **Cache layer** for performance

```
CREATE TABLE rate_limits (
  key TEXT PRIMARY KEY,
  tokens INTEGER,
  last_updated TIMESTAMP,
  rate INTEGER,
  capacity INTEGER
);
```

Implementation Details:

- Advisory locks for atomic operations
- Sliding window counting
- Automatic cleanup of expired entries
- Monitoring and alerting system

5. Design a job queue system with PostgreSQL for handling background tasks. How would you ensure reliability and scaling?

System Components:

- **Queue Table** with job metadata
- **Worker Processes** for execution
- **Status Tracking** mechanism

```
CREATE TABLE jobs (
  id SERIAL PRIMARY KEY,
  payload JSONB,
  status VARCHAR(20),
  priority INTEGER,
  scheduled_at TIMESTAMP
);
```

Reliability Features:

- Transaction-based job claiming
- Dead letter queue for failed jobs
- Retry mechanism with exponential backoff
- Job prioritization system

6. Design a real-time analytics system using PostgreSQL. How would you handle high-volume inserts and aggregations?

Architecture:

- **Time-series partitioning** for events
- **Materialized views** for aggregations
- **Parallel query execution**

```
CREATE TABLE events (
  ts TIMESTAMP,
  event_type TEXT,
  data JSONB
) PARTITION BY RANGE (ts);
```

Optimization Strategies:

- Hypertable partitioning
- Continuous aggregates
- Retention policy automation
- Query result caching

7. How would you implement a distributed session management system using PostgreSQL?

Core Features:

- **Session storage** with TTL
- **Distributed locking** mechanism
- **Cache invalidation** strategy

```
CREATE TABLE sessions (
  id UUID PRIMARY KEY,
  user_data JSONB,
  expires_at TIMESTAMP,
  last_accessed TIMESTAMP
);
```

Implementation Details:

- Automatic session cleanup
- Redis cache integration
- Session data encryption
- High availability setup

8. Design a content tagging system with efficient search capabilities using PostgreSQL.

System Components:

- **GIN indexes** for full-text search
- **Hierarchical tags** structure
- **Cache layer** for popular searches

```
CREATE TABLE content_tags (
  content_id INTEGER,
  tag_path ltree,
  metadata JSONB,
  UNIQUE(content_id, tag_path)
);
```

Search Optimizations:

- tsvector columns for text search
- Materialized path pattern
- Cached aggregate counts
- Bitmap scan optimization

9. How would you implement a distributed locking mechanism using PostgreSQL?

Implementation Approach:

- **Advisory locks** for coordination
- **Deadlock prevention** strategy

- **Lock timeout** handling

```
CREATE TABLE distributed_locks (
  lock_key TEXT PRIMARY KEY,
  owner TEXT,
  acquired_at TIMESTAMP,
  timeout INTEGER
);
```

Key Features:

- Automatic lock release
- Lock acquisition queuing
- Health check mechanism
- Monitoring and alerting

10. Design a scalable e-commerce inventory management system using PostgreSQL.

Core Components:

- **Stock tracking** with MVCC
- **Reserved inventory** handling
- **Transaction isolation** strategy

```
CREATE TABLE inventory (
  product_id INTEGER PRIMARY KEY,
  quantity INTEGER,
  reserved INTEGER,
  version INTEGER
);
```

Implementation Features:

- Optimistic locking
- Inventory reservations
- Stock level triggers
- Audit trail logging

Coding and Debugging

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

1. Implement a trigger to maintain audit logs for table changes

Solution:

```
CREATE TRIGGER audit_trigger
AFTER INSERT OR UPDATE OR DELETE ON target_table
FOR EACH ROW EXECUTE FUNCTION audit_function();
```

Audit function:

```
CREATE FUNCTION audit_function()
RETURNS trigger AS $$
BEGIN
  INSERT INTO audit_logs VALUES (NOW(), TG_OP, NEW, OLD);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

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

Solution:

```
SELECT column1, column2, COUNT(*) as count
FROM table_name
GROUP BY column1, column2
HAVING COUNT(*) > 1;
```

Key points:

- GROUP BY combines records with matching values
- HAVING filters after grouping
- COUNT(*) returns number of duplicates

3. Create a recursive CTE to generate a sequence of dates between two dates

Solution:

```
WITH RECURSIVE date_range AS (
  SELECT '2023-01-01'::date AS date
  UNION ALL
  SELECT date + 1 FROM date_range
  WHERE date < '2023-01-31'::date
)
SELECT * FROM date_range;
```

Explanation:

- Base case starts with initial date
- Recursive part adds one day until end date
- ::date ensures proper date casting

4. Write a query to pivot rows into columns dynamically

Solution:

```
SELECT *
```

```
FROM crosstab(
  'SELECT row_name, category, value
   FROM source_table
   ORDER BY 1,2',
  'SELECT DISTINCT category
   FROM source_table ORDER BY 1'
) AS ct (row_name text, col1 int, col2 int);
```

Notes:

- Requires tablefunc extension
- crosstab function handles dynamic pivoting
- Define output columns explicitly

5. Create a function to calculate running totals with window functions

Solution:

```
CREATE FUNCTION running_total()
RETURNS TABLE (date date, amount numeric, total numeric) AS $$
SELECT date, amount,
       SUM(amount) OVER (ORDER BY date)
FROM transactions;
$$ LANGUAGE SQL;
```

Key concepts:

- Window function maintains running sum
- OVER clause defines calculation window
- Returns result as a table

6. Debug a slow-performing query using EXPLAIN ANALYZE

Analysis Steps:

```
EXPLAIN ANALYZE
SELECT * FROM orders o
JOIN customers c ON c.id = o.customer_id
WHERE o.status = 'pending';
```

Optimization checklist:

- Check for sequential scans vs index scans
- Verify join types (nested loop vs hash)
- Look for high-cost operations
- Ensure proper indexing on join columns

7. Write a query to find gaps in sequential data

Solution:

```
SELECT id + 1 as gap_start, next_id - 1 as gap_end
FROM (
  SELECT id, LEAD(id) OVER (ORDER BY id) as next_id
  FROM sequence_table
) t WHERE next_id - id > 1;
```

Concepts used:

- LEAD window function for next value
- Subquery to compare consecutive values
- Gap detection logic

8. Implement a custom aggregate function for median calculation

Solution:

```
CREATE AGGREGATE median(numeric) (
  SFUNC = array_append,
```

```
  STYPE = numeric[],
  FINALFUNC = array_median,
  INITCOND = '{}'
);
```

Components:

- SFUNC: State transition function
- STYPE: State value data type
- FINALFUNC: Final calculation function

9. Debug and fix a deadlock situation in concurrent transactions

Analysis approach:

- Check pg_locks view for lock information
- Review transaction isolation levels
- Examine lock acquisition order

```
SELECT blocked_locks.pid AS blocked_pid,
       blocking_locks.pid AS blocking_pid
FROM pg_catalog.pg_locks blocked_locks
JOIN pg_catalog.pg_locks blocking_locks ON blocked_locks.transactionid = blocking_locks.transactionid;
```

10. Implement a hierarchical query for organizational structure

Solution:

```
WITH RECURSIVE org_tree 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, t.level + 1
  FROM employees e JOIN org_tree t ON e.manager_id = t.id
)
SELECT * FROM org_tree;
```

Features:

- Recursive CTE for tree traversal
- Level tracking
- Self-joining structure