SQL Database

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 clustered and non-clustered indexes, and when would you use each?

Clustered vs Non-Clustered Indexes

Clustered Index:

- Determines the physical order of data in the table
- Only one per table (the table IS the index)
- Leaf nodes contain actual data rows
- Typically created on primary key by default
- Faster for range queries and sorting operations

Non-Clustered Index:

- Separate structure from the table data
- Multiple non-clustered indexes allowed per table
- Leaf nodes contain pointers (row locators) to actual data
- Additional lookup required to retrieve full row data
- Ideal for columns frequently used in WHERE, JOIN, or ORDER BY clauses

Usage Guidelines:

- Use clustered indexes on columns with sequential values (timestamps, auto-increment IDs) or frequently used in range queries
- Use non-clustered indexes on foreign keys, frequently searched columns, and columns used in JOIN conditions
- Avoid clustered indexes on frequently updated columns as physical reordering is expensive

2. What are the ACID properties and how do different isolation levels affect them?

ACID Properties

Atomicity: Transactions are all-or-nothing. Either all operations succeed or all fail and rollback.

Consistency: Database moves from one valid state to another, maintaining all defined rules and constraints.

Isolation: Concurrent transactions don't interfere with each other.

Durability: Committed transactions persist even after system failures.

Isolation Levels and Their Impact

READ UNCOMMITTED:

- Lowest isolation, allows dirty reads
- Can read uncommitted changes from other transactions
- Highest concurrency, lowest data integrity

READ COMMITTED:

- Prevents dirty reads
- Can still experience non-repeatable reads and phantom reads
- Default in most databases (PostgreSQL, SQL Server)

REPEATABLE READ:

- Prevents dirty and non-repeatable reads
- Still susceptible to phantom reads
- Locks rows read during transaction

SERIALIZABLE:

- Highest isolation, prevents all anomalies
- Transactions execute as if serially, one after another
- Lowest concurrency, potential for deadlocks

3. Explain query execution plans and how you would optimize a slow-running query.

Query Execution Plans

An execution plan is the roadmap the database engine creates to execute a SQL query, showing operations like table scans, index seeks, joins, and sorts.

Key Components to Analyze:

- **Scan vs Seek:** Table/Index Scans read entire tables; Index Seeks use indexes efficiently
- **Join Types:** Nested Loop, Hash Join, Merge Join - each optimal for different scenarios
- **Cost Estimates:** Relative resource consumption for each operation
- **Cardinality:** Estimated vs actual rows processed

Optimization Approach

```
-- Example: Analyzing execution plan
EXPLAIN ANALYZE
SELECT o.order_id, c.name
FROM orders o
JOIN customers c ON o.customer_id = c.id
WHERE o.order_date > '2024-01-01';
```

Optimization Steps:

- Identify table scans and replace with index seeks by adding appropriate indexes
- Check for missing indexes on JOIN and WHERE clause columns
- Eliminate unnecessary columns in SELECT (avoid SELECT *)
- Rewrite subqueries as JOINs when possible
- Update statistics to ensure accurate cardinality estimates
- Consider partitioning for large tables
- Use covering indexes to avoid key lookups

4. What is database normalization? Explain 1NF, 2NF, 3NF, and when denormalization is appropriate.

Database Normalization

Normalization is the process of organizing data to minimize redundancy and dependency, improving data integrity and reducing anomalies.

First Normal Form (1NF):

- Eliminate repeating groups
- Each column contains atomic (indivisible) values
- Each row is unique (has primary key)

Second Normal Form (2NF):

- Must be in 1NF
- Remove partial dependencies
- All non-key attributes fully depend on the entire primary key
- Applies to tables with composite keys

Third Normal Form (3NF):

- Must be in 2NF

- Remove transitive dependencies
- Non-key attributes depend only on the primary key, not on other non-key attributes

When to Denormalize

Appropriate Scenarios:

- **Read-heavy workloads:** When query performance is critical and writes are infrequent
- **Data warehousing:** Star/snowflake schemas for analytical queries
- **Calculated fields:** Storing aggregated values to avoid expensive calculations
- **Reducing joins:** When multiple joins significantly impact performance

Trade-offs:

- Faster reads at the cost of data redundancy
- Increased storage requirements
- More complex update logic to maintain consistency
- Higher risk of data anomalies

5. Explain the differences between INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, and CROSS JOIN with use cases.

SQL Join Types

INNER JOIN:

- Returns only matching rows from both tables
- Most common join type
- Use case: Get orders with customer information (exclude orders without customers)

```
SELECT o.*, c.name
FROM orders o
INNER JOIN customers c ON o.customer_id = c.id;
```

LEFT JOIN (LEFT OUTER JOIN):

- Returns all rows from left table, matching rows from right table
- NULL for non-matching right table columns
- Use case: Get all customers and their orders (including customers with no orders)

```
SELECT c.name, COUNT(o.id) as order_count
FROM customers c
LEFT JOIN orders o ON c.id = o.customer_id
GROUP BY c.name;
```

RIGHT JOIN:

- Opposite of LEFT JOIN - all rows from right table
- Less commonly used (can be rewritten as LEFT JOIN)

FULL OUTER JOIN:

- Returns all rows from both tables
- NULL where no match exists
- Use case: Reconciliation queries, finding unmatched records in both tables

CROSS JOIN:

- Cartesian product of both tables
- Every row from first table paired with every row from second
- Use case: Generating combinations, test data, or calendar tables

6. What are window functions and how do they differ from GROUP BY? Provide practical examples.

Window Functions

Window functions perform calculations across a set of rows related to the current row without collapsing the result set, unlike GROUP BY which aggregates rows.

Key Differences from GROUP BY:

- Window functions retain individual rows while adding calculated columns
- GROUP BY collapses rows into groups
- Window functions can access other rows in the partition
- Multiple window functions can be used with different partitions in one query

Common Window Functions

```
-- ROW_NUMBER: Assign unique sequential numbers
SELECT
  employee_id,
  salary,
  ROW_NUMBER() OVER (ORDER BY salary DESC) as rank
FROM employees;

-- RANK with PARTITION BY
SELECT
  department,
  employee_name,
  salary,
  RANK() OVER (PARTITION BY department
               ORDER BY salary DESC) as dept_rank
FROM employees;
```

Practical Use Cases:

- **Running totals:** SUM() OVER (ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
- **Moving averages:** AVG() OVER (ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
- **Top N per group:** Using ROW_NUMBER() with PARTITION BY
- **Lag/Lead analysis:** LAG() and LEAD() for comparing with previous/next rows

7. Explain deadlocks in databases. How do they occur and what strategies prevent them?

Database Deadlocks

A deadlock occurs when two or more transactions are waiting for each other to release locks, creating a circular dependency that prevents any transaction from proceeding.

Classic Deadlock Scenario:

- Transaction A locks Row 1, needs Row 2
- Transaction B locks Row 2, needs Row 1
- Both wait indefinitely for the other to release

How Deadlocks Occur

- **Lock ordering:** Transactions acquire locks in different orders
- **Lock escalation:** Row locks escalate to page or table locks
- **Long transactions:** Extended lock hold times increase collision probability
- **High concurrency:** Many simultaneous transactions competing for resources

Prevention Strategies

1. Consistent Lock Ordering:

- Always access tables/rows in the same order across all transactions
- Establish application-wide conventions

2. Keep Transactions Short:

- Minimize time locks are held
- Avoid user interaction within transactions

3. Use Appropriate Isolation Levels:

- Lower isolation when consistency requirements allow

- READ COMMITTED instead of SERIALIZABLE when possible

4. Index Optimization:

- Proper indexes reduce lock scope and duration
- Avoid table scans that lock many rows

5. Deadlock Detection and Retry:

```
BEGIN TRY
  BEGIN TRANSACTION
  -- operations here
  COMMIT
END TRY
BEGIN CATCH
  IF ERROR_NUMBER() = 1205 -- Deadlock
    ROLLBACK AND RETRY
END CATCH
```

8. What are CTEs (Common Table Expressions) and how do recursive CTEs work? When would you use them?

Common Table Expressions (CTEs)

CTEs are temporary named result sets that exist only during query execution, improving readability and enabling recursive queries.

Basic CTE Syntax:

```
WITH sales_summary AS (
  SELECT
    product_id,
    SUM(amount) as total_sales
  FROM orders
  GROUP BY product_id
)
SELECT p.name, s.total_sales
FROM products p
JOIN sales_summary s ON p.id = s.product_id;
```

Advantages Over Subqueries:

- Better readability and maintainability
- Can be referenced multiple times in the same query
- Easier to debug and test incrementally

Recursive CTEs

Used for hierarchical or graph data structures. Contains anchor member and recursive member.

```
WITH RECURSIVE employee_hierarchy AS (
  -- Anchor: starting point
  SELECT id, name, manager_id, 1 as level
  FROM employees
  WHERE manager_id IS NULL

  UNION ALL

  -- Recursive: join with CTE itself
  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;
```

Use Cases:

- Organizational hierarchies (managers/employees)
- Bill of materials (parts/subparts)

- File system traversal
- Social network connections
- Category trees

9. Explain the difference between DELETE, TRUNCATE, and DROP. What are the performance and transactional implications?

DELETE vs TRUNCATE vs DROP

DELETE:

- DML (Data Manipulation Language) operation
- Removes rows based on WHERE clause (or all rows if no WHERE)
- Can be rolled back (transactional)
- Triggers fire for each deleted row
- Logs each row deletion (slow for large datasets)
- Does not reset identity/auto-increment values
- Table structure remains intact

```
DELETE FROM orders WHERE order_date < '2020-01-01';
-- Can rollback
ROLLBACK;
```

TRUNCATE:

- DDL (Data Definition Language) operation
- Removes all rows from table (no WHERE clause)
- Cannot be rolled back in most databases (auto-commits)
- Triggers do not fire
- Minimal logging (deallocates data pages)
- Resets identity/auto-increment to seed value
- Much faster than DELETE for entire table
- Requires ALTER TABLE permission

```
TRUNCATE TABLE orders;
-- Fast, minimal logging
```

DROP:

- DDL operation
- Removes entire table structure and data
- Cannot be rolled back (auto-commits)
- Frees storage space completely
- All indexes, triggers, constraints removed
- Permissions on table are also dropped

```
DROP TABLE orders;
-- Table no longer exists
```

Performance Comparison:

- DELETE: Slowest, row-by-row logging
- TRUNCATE: Fastest for removing all data
- DROP: Instant, removes everything

10. What are covering indexes and include columns? How do they improve query performance?

Covering Indexes

A covering index contains all columns needed by a query, allowing the database to satisfy the query entirely from the index without accessing the base table.

Key Concept:

- Index "covers" the query - no additional lookups required
- Eliminates expensive key lookup operations
- Significantly faster for read operations

Traditional Index Problem

```
-- Index on customer_id only
CREATE INDEX idx_orders_customer
  ON orders(customer_id);

-- Query needs order_date too
SELECT order_id, order_date
FROM orders
WHERE customer_id = 123;
-- Requires: index seek + key lookup to table
```

Covering Index Solution

```
-- Include additional columns
CREATE INDEX idx_orders_customer_covering
  ON orders(customer_id)
  INCLUDE (order_date, order_id);

-- Now fully covered - no table access needed
```

INCLUDE Columns (SQL Server, PostgreSQL):

- Columns stored at leaf level only (not in B-tree structure)
- Don't affect index key size or sort order
- Useful for columns in SELECT but not in WHERE/JOIN
- Reduces index size compared to adding to key columns

When to Use:

- Frequently executed queries with predictable column access
- Queries filtering on few columns but selecting many
- Read-heavy workloads where index size trade-off is acceptable

Trade-offs:

- Larger index size (more storage)
- Slower INSERT/UPDATE/DELETE operations
- Increased maintenance overhead

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 SQL?

LRU Cache Implementation

An **LRU cache** can be implemented using a combination of a hash table and doubly linked list. In SQL databases, you can simulate this using:

- **Timestamp-based approach:** Store access timestamps and evict oldest entries
- **Sequence counter:** Use an incrementing counter to track access order
- **Materialized view:** Maintain a ranked list of most recently used items

```
CREATE TABLE cache_store (
  key VARCHAR(255) PRIMARY KEY,
  value TEXT,
  last_accessed BIGINT,
  INDEX idx_accessed (last_accessed)
);

DELETE FROM cache_store
WHERE last_accessed < (SELECT MIN(last_accessed) FROM
  (SELECT last_accessed FROM cache_store ORDER BY last_accessed DESC LIMIT 100) t);
```

Time complexity: O(1) for get/put with proper indexing, O(log n) for eviction queries.

2. Explain how B-Tree and B+Tree data structures work in database indexing. What are their time complexities?

B-Tree vs B+Tree

B-Tree: A self-balancing tree where each node can have multiple keys and children. Data is stored in both internal and leaf nodes.

B+Tree: A variant where all data is stored only in leaf nodes, with internal nodes containing only keys for navigation. Leaf nodes are linked for efficient range queries.

- **Search:** O(log n) - traverse from root to leaf
- **Insert:** O(log n) - may require node splits
- **Delete:** O(log n) - may require node merges
- **Range Query:** O(log n + k) where k is result size

Why databases prefer B+Trees:

- Better cache locality (sequential leaf access)
- Higher fanout (more keys per node)
- Efficient full scans via leaf node links
- All data at same depth ensures consistent performance

3. How do you find all pairs in a table that sum to a target value using SQL?

Pair Sum Problem in SQL

To find pairs that sum to a target value, use a **self-join** with appropriate conditions:

```
SELECT a.id AS id1, b.id AS id2, 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;
```

```
-- With hash-based approach for better performance:
SELECT a.value, (@target - a.value) AS pair_value
FROM numbers a
WHERE EXISTS (
  SELECT 1 FROM numbers b
  WHERE b.value = @target - a.value AND b.id > a.id
);
```

Time Complexity: O(n²) for nested loop join, O(n) with hash join if database optimizer uses hash-based execution. Space complexity: O(1) for self-join approach.

4. Implement a sliding window algorithm in SQL to calculate moving averages.

Sliding Window with Window Functions

SQL window functions provide efficient **sliding window** operations:

```
-- Moving average over last 3 rows
SELECT
  date,
  value,
  AVG(value) OVER (
    ORDER BY date
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
  ) AS moving_avg_3
FROM sales_data;
```

Time-based sliding window:

```
SELECT
  timestamp,
  amount,
  SUM(amount) OVER (
    ORDER BY timestamp
    RANGE BETWEEN INTERVAL '7' DAY PRECEDING AND CURRENT ROW
  ) AS sum_last_7_days
FROM transactions;
```

Complexity: O(n log n) for sorting, O(n) for window computation. Modern databases optimize this to single-pass algorithms.

5. How are hash tables implemented in database hash indexes? What are the collision resolution strategies?

Hash Index Implementation

Hash indexes use hash functions to map keys to bucket locations for O(1) average-case lookups.

Collision Resolution Strategies:

- **Chaining:** Each bucket contains a linked list of entries (most common in databases)
- **Open Addressing:** Linear probing, quadratic probing, or double hashing
- **Extendible Hashing:** Dynamic directory-based approach that grows gracefully

Time Complexity:

- Average case: O(1) for search, insert, delete
- Worst case: O(n) when all keys hash to same bucket
- Load factor $\alpha = n/m$ affects performance (typically kept < 0.75)

Limitations: Hash indexes don't support range queries, sorting, or prefix matching - only equality comparisons.

6. Explain how to detect cycles in hierarchical data stored in SQL tables.

Cycle Detection in Hierarchical Data

For **parent-child relationships**, detect cycles using recursive CTEs with path tracking:

```
WITH RECURSIVE hierarchy_path AS (
  SELECT id, parent_id, ARRAY[id] AS path, 0 AS depth
  FROM categories WHERE parent_id IS NULL
  UNION ALL
  SELECT c.id, c.parent_id, h.path || c.id, h.depth + 1
  FROM categories c
  JOIN hierarchy_path h ON c.parent_id = h.id
  WHERE NOT c.id = ANY(h.path) AND h.depth < 100
)
SELECT * FROM hierarchy_path;
```

Cycle Detection Query:

```
SELECT id, parent_id FROM categories c
WHERE EXISTS (
  WITH RECURSIVE check_cycle AS (...)
  SELECT 1 FROM check_cycle WHERE id = ANY(path[1:array_length(path,1)-1])
);
```

Complexity: O(V + E) using depth-first traversal with visited set tracking.

7. How do you implement a stack data structure using SQL tables? Provide push and pop operations.

Stack Implementation in SQL

A **stack (LIFO)** can be implemented using an auto-incrementing sequence as position tracker:

```
CREATE TABLE stack (
  position BIGSERIAL PRIMARY KEY,
  value TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- PUSH operation
INSERT INTO stack (value) VALUES ('item1');

-- POP operation
DELETE FROM stack
WHERE position = (SELECT MAX(position) FROM stack)
RETURNING value;

-- PEEK operation
SELECT value FROM stack
ORDER BY position DESC LIMIT 1;
```

Time Complexity: O(1) for push with indexed position, O(log n) for pop due to MAX() operation. Can optimize to O(1) by maintaining a separate counter table.

8. What is the time complexity of different SQL JOIN operations, and how does the database optimizer choose join algorithms?

JOIN Algorithm Complexities

Join Algorithm Types:

- **Nested Loop Join:** O(n × m) - simple but slow, used for small tables or when no indexes exist
- **Hash Join:** O(n + m) - builds hash table on smaller relation, probes with larger. Requires memory for hash table
- **Merge Join:** O(n log n + m log m) - requires sorted inputs, efficient for pre-sorted data or indexed columns

```
-- Force specific join algorithm (PostgreSQL)
SET enable_hashjoin = off;
SET enable_mergejoin = off;
-- Forces nested loop

EXPLAIN ANALYZE
SELECT * FROM orders o
```

```
JOIN customers c ON o.customer_id = c.id;
```

Optimizer Considerations: Table size, available indexes, memory, cardinality estimates, and join selectivity factor into cost-based decisions.

9. How do you implement a priority queue or heap structure for processing records by priority in SQL?

Priority Queue in SQL

A **priority queue** can be implemented using indexed priority columns with efficient extraction:

```
CREATE TABLE task_queue (
  id BIGSERIAL PRIMARY KEY,
  task_data JSONB,
  priority INT NOT NULL,
  created_at TIMESTAMP DEFAULT NOW(),
  INDEX idx_priority (priority DESC, created_at ASC)
);

-- ENQUEUE
INSERT INTO task_queue (task_data, priority)
VALUES ('data', 10);

-- DEQUEUE (highest priority)
DELETE FROM task_queue
WHERE id = (
  SELECT id FROM task_queue
  ORDER BY priority DESC, created_at ASC
  LIMIT 1 FOR UPDATE SKIP LOCKED
) RETURNING *;
```

Complexity: O(log n) for enqueue/dequeue with B-tree index. **SKIP LOCKED** ensures concurrent processing without contention.

10. Explain how bitmap indexes work and when they provide better performance than B-tree indexes.

Bitmap Index Structure

Bitmap indexes use bit arrays where each bit represents whether a row contains a specific value. Each distinct value has its own bitmap.

Structure Example: For column with values [A, B, A, C], bitmaps are:

- A: [1,0,1,0]
- B: [0,1,0,0]
- C: [0,0,0,1]

When to Use Bitmap Indexes:

- **Low cardinality columns** (few distinct values): gender, status, boolean flags
- **Complex WHERE clauses** with multiple conditions - bitmaps combine efficiently with AND/OR/NOT operations
- **Data warehouse/OLAP** workloads with read-heavy operations
- **Large tables** where space efficiency matters

Performance: O(1) for bitmap lookup, O(n/w) for bitmap operations where w is word size. NOT suitable for OLTP due to update overhead.

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. Discuss the database schema, API design, and how you'd handle high read/write traffic.

Architecture Overview

A URL shortener requires handling **high read traffic** (redirects) and moderate write traffic (URL creation). Key components include:

- **API Layer:** RESTful endpoints for creating and retrieving URLs
- **Database:** Store mappings between short codes and original URLs
- **Cache Layer:** Redis/Memcached for frequently accessed URLs
- **Load Balancer:** Distribute traffic across multiple application servers

Database Schema

```
CREATE TABLE urls (
  id BIGINT PRIMARY KEY AUTO_INCREMENT,
  short_code VARCHAR(10) UNIQUE NOT NULL,
  original_url TEXT NOT NULL,
  user_id BIGINT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  expires_at TIMESTAMP,
  click_count INT DEFAULT 0,
  INDEX idx_short_code (short_code)
);
```

Key Design Decisions

- **Short Code Generation:** Use base62 encoding (a-z, A-Z, 0-9) of auto-incrementing ID or MD5 hash with collision handling
- **Caching Strategy:** Cache hot URLs with TTL, cache-aside pattern for reads
- **Scalability:** Horizontal scaling with stateless application servers, read replicas for database
- **Database Choice:** SQL for ACID properties and relationships, or NoSQL (Cassandra) for massive scale

CAP Theorem Consideration

Prioritize **Availability and Partition Tolerance (AP)** over strict consistency. Eventual consistency is acceptable for analytics, but short code uniqueness requires strong consistency during creation.

2. How would you design a real-time chat application supporting millions of users? Discuss message storage, delivery guarantees, and presence management.

System Components

- **WebSocket Gateway:** Persistent connections for real-time bidirectional communication
- **Message Queue:** Kafka/RabbitMQ for reliable message delivery
- **Presence Service:** Redis for tracking online/offline status
- **Message Storage:** Hybrid approach with hot/cold storage
- **Notification Service:** Push notifications for offline users

Database Schema

```
CREATE TABLE messages (
  id BIGINT PRIMARY KEY,
  conversation_id BIGINT NOT NULL,
```

```
  sender_id BIGINT NOT NULL,
  content TEXT,
  created_at TIMESTAMP,
  delivered_at TIMESTAMP,
  read_at TIMESTAMP,
  INDEX idx_conversation (conversation_id, created_at)
) PARTITION BY RANGE (UNIX_TIMESTAMP(created_at));
```

Architecture Decisions

- **Message Delivery:** At-least-once delivery using message acknowledgments and idempotency keys
- **Scalability:** Shard users across WebSocket servers using consistent hashing
- **Storage Strategy:** Recent messages in SQL/NoSQL, archive old messages to object storage (S3)
- **Presence:** Use Redis sorted sets with heartbeat mechanism, TTL-based expiration

CAP Theorem Trade-offs

Choose **AP (Availability + Partition Tolerance)**. Users can send messages even during network partitions, with eventual delivery. Use vector clocks or sequence numbers to handle message ordering conflicts.

3. Design a social media news feed system like Facebook or Twitter. How would you handle feed generation, ranking, and real-time updates at scale?

Feed Architecture Patterns

- **Fan-out on Write (Push Model):** Pre-compute feeds when posts are created, store in user's feed cache
- **Fan-out on Read (Pull Model):** Compute feed on-demand by querying followed users' posts
- **Hybrid Approach:** Push for users with few followers, pull for celebrities

Database Schema

```
CREATE TABLE posts (
  id BIGINT PRIMARY KEY,
  user_id BIGINT NOT NULL,
  content TEXT,
  created_at TIMESTAMP,
  INDEX idx_user_time (user_id, created_at)
);
CREATE TABLE feeds (
  user_id BIGINT,
  post_id BIGINT,
  score DECIMAL(10,2),
  created_at TIMESTAMP,
  PRIMARY KEY (user_id, post_id)
);
```

Feed Generation Strategy

- **Write Path:** When user posts, asynchronously fan-out to followers' feed tables/Redis lists
- **Read Path:** Fetch top N posts from pre-computed feed, merge with real-time posts
- **Ranking Algorithm:** Score based on recency, engagement (likes, comments), user affinity
- **Caching:** Redis sorted sets for hot feeds, pagination using cursor-based approach

Scalability Considerations

- **Database:** Shard by user_id, use Cassandra for write-heavy workloads
- **Real-time Updates:** WebSockets or Server-Sent Events for live feed updates
- **Load Balancing:** Consistent hashing to route users to same cache servers

4. Design a distributed rate limiting system to prevent API abuse. Discuss algorithms, storage mechanisms, and handling edge cases in a multi-datacenter setup.

Rate Limiting Algorithms

- **Token Bucket:** Refill tokens at fixed rate, consume tokens per request. Allows bursts.
- **Leaky Bucket:** Process requests at constant rate, queue overflow requests
- **Fixed Window:** Count requests per time window, simple but has boundary issues
- **Sliding Window Log:** Track timestamps of requests, accurate but memory-intensive
- **Sliding Window Counter:** Hybrid approach, weighted count from current and previous window

Implementation with Redis

```
-- Token Bucket in Redis
local tokens = redis.call('GET', KEYS[1])
if not tokens then
  redis.call('SET', KEYS[1], ARGV[1]-1, 'EX', ARGV[2])
  return 1
end
if tonumber(tokens) > 0 then
  redis.call('DECR', KEYS[1])
  return 1
end
return 0
```

Distributed Challenges

- **Race Conditions:** Use Redis Lua scripts for atomic operations
- **Clock Synchronization:** Use NTP or logical clocks to avoid skew issues
- **Multi-Datacenter:** Local rate limiting per DC with eventual global sync, or use distributed counters (Cassandra)
- **Storage Choice:** Redis for low latency, but consider replication lag

Advanced Considerations

- **User Tiers:** Different limits for free/premium users stored in configuration service
- **Graceful Degradation:** Allow requests if rate limiter is down (fail-open vs fail-closed)
- **HTTP Headers:** Return X-RateLimit-Remaining, X-RateLimit-Reset headers

5. Design a distributed job scheduler system like Airflow or Kubernetes CronJobs. How would you ensure jobs run exactly once and handle failures?

Core Components

- **Scheduler Service:** Determines when jobs should run based on cron expressions or dependencies
- **Job Queue:** Distributed queue (RabbitMQ, Kafka) for job dispatch
- **Worker Pool:** Horizontally scalable workers that execute jobs
- **State Store:** Database tracking job status, execution history
- **Coordination Service:** Zookeeper or etcd for leader election

Database Schema

```
CREATE TABLE jobs (
  id BIGINT PRIMARY KEY,
  name VARCHAR(255) UNIQUE,
  schedule VARCHAR(100),
  last_run TIMESTAMP,
  next_run TIMESTAMP,
  status ENUM('pending','running','success','failed'),
  retry_count INT DEFAULT 0,
  max_retries INT DEFAULT 3
);
```

Exactly-Once Execution

- **Distributed Lock:** Use Redis SETNX or database row-level locks before job execution
- **Idempotency:** Assign unique execution ID, workers check if ID already processed
- **Pessimistic Locking:** UPDATE jobs SET status='running' WHERE id=? AND status='pending'
- **Lease-based Execution:** Worker acquires time-bound lease, heartbeat to maintain

Failure Handling

- **Dead Letter Queue:** Move failed jobs after max retries for manual inspection
- **Exponential Backoff:** Increase delay between retries
- **Circuit Breaker:** Pause job if downstream dependencies consistently fail
- **Monitoring:** Track SLA violations, execution duration, failure rates

6. Design a content delivery network (CDN) architecture. Discuss edge caching, cache invalidation strategies, and handling dynamic content.

CDN Architecture Layers

- **Edge Servers:** Geographically distributed cache nodes closest to users
- **Origin Servers:** Primary data source, application servers and databases
- **Mid-tier Cache:** Regional caches between edge and origin
- **DNS/Routing:** GeoDNS or Anycast to route users to nearest edge

Caching Strategy

- **Cache-Control Headers:** max-age, s-maxage, public/private directives from origin
- **Cache Keys:** URL + query parameters + headers (Accept-Language, Accept-Encoding)
- **Cache Hierarchy:** L1 (memory), L2 (SSD), L3 (origin fetch)
- **Eviction Policy:** LRU or LFU based on access patterns and storage capacity

Cache Invalidation Approaches

- **TTL-based:** Set expiration times, simple but may serve stale content
- **Purge API:** Explicit invalidation via API calls when content updates
- **Tag-based:** Associate content with tags, purge all content with specific tag
- **Versioned URLs:** Immutable content with version/hash in URL path

Dynamic Content Handling

- **Edge Computing:** Run application logic at edge (Cloudflare Workers, Lambda@Edge)
- **ESI (Edge Side Includes):** Cache page fragments, assemble at edge
- **Personalization:** Cache common parts, fetch personalized data via AJAX
- **Streaming:** Use HTTP/2 Server Push or chunked transfer encoding

7. Design an e-commerce inventory management system handling concurrent purchases. How would you prevent overselling and maintain consistency across distributed warehouses?

Core Challenges

- **Race Conditions:** Multiple users purchasing last item simultaneously
- **Distributed Inventory:** Stock spread across multiple warehouses/regions
- **Consistency vs Availability:** Balance between accurate stock counts and system availability

Database Schema

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

Concurrency Control Strategies

- **Pessimistic Locking:** SELECT FOR UPDATE during transaction, blocks other transactions
- **Optimistic Locking:** Version field, UPDATE WHERE version=old_version, retry on conflict
- **Reservation System:** Reserve inventory temporarily, confirm on payment, release on timeout
- **Atomic Operations:** UPDATE inventory SET quantity=quantity-1 WHERE product_id=? AND quantity>=1

Implementation Example

```
BEGIN TRANSACTION;
UPDATE inventory
SET reserved = reserved + ?, version = version + 1
WHERE product_id = ? AND warehouse_id = ?
  AND (quantity - reserved) >= ?
  AND version = ?;
IF affected_rows = 0 THEN ROLLBACK;
ELSE COMMIT;
END;
```

Distributed Warehouse Strategy

- **Routing Logic:** Check nearest warehouse first, fallback to others if out of stock
- **Eventual Consistency:** Use saga pattern for cross-warehouse transfers
- **CQRS:** Separate read model (cached aggregated stock) from write model (actual inventory)

8. Design a notification system supporting multiple channels (email, SMS, push, in-app). Discuss priority handling, retry logic, and preventing duplicate notifications.

System Architecture

- **Notification Service:** API for creating notification requests
- **Channel Handlers:** Separate workers for email, SMS, push notifications
- **Message Queue:** Kafka/SQS with priority queues for urgent notifications
- **Template Engine:** Store and render notification templates
- **Delivery Tracker:** Track delivery status and user preferences

Database Schema

```
CREATE TABLE notifications (
  id BIGINT PRIMARY KEY,
  user_id BIGINT NOT NULL,
  type VARCHAR(50),
  channels JSON,
  priority ENUM('low','medium','high','urgent'),
  idempotency_key VARCHAR(255) UNIQUE,
  status VARCHAR(20),
  created_at TIMESTAMP,
  INDEX idx_user_status (user_id, status)
);
```

Priority and Routing

- **Priority Queues:** Separate Kafka topics/SQS queues per priority level
- **User Preferences:** Check opt-in/opt-out settings before sending
- **Channel Selection:** Try primary channel first, fallback to alternatives on failure
- **Rate Limiting:** Prevent notification spam per user per time window

Reliability Mechanisms

- **Idempotency:** Use idempotency_key to prevent duplicate sends
- **Retry Logic:** Exponential backoff with max retry limit, dead letter queue
- **Circuit Breaker:** Pause channel if provider consistently fails
- **Delivery Confirmation:** Webhooks from providers (Twilio, SendGrid) to update status

Scalability Considerations

- **Batching:** Batch similar notifications to reduce API calls
- **Template Caching:** Cache rendered templates in Redis
- **Async Processing:** All channel handlers work asynchronously

9. Design a search and autocomplete system like Google Search. Discuss indexing strategies, query processing, ranking algorithms, and handling typos.

System Components

- **Indexing Pipeline:** Crawl/ingest data, tokenize, build inverted index

- **Search Service:** Process queries, retrieve and rank results
- **Autocomplete Service:** Suggest completions as user types
- **Storage:** Elasticsearch/Solr for full-text search, Trie for autocomplete

Indexing Strategy

- **Inverted Index:** Map terms to document IDs with positional information
- **Tokenization:** Lowercase, remove stopwords, stemming/lemmatization
- **N-grams:** Generate character n-grams for fuzzy matching
- **Document Boosting:** Assign weights based on authority, freshness, popularity

Autocomplete Implementation

```
CREATE TABLE autocomplete (
  prefix VARCHAR(100),
  suggestion VARCHAR(255),
  score INT,
  PRIMARY KEY (prefix, suggestion)
);
-- Or use Trie data structure
-- Redis sorted sets for popularity-based ranking
ZADD autocomplete:prefix score suggestion
```

Query Processing

- **Query Parsing:** Identify operators (AND, OR, NOT), phrases, filters
- **Query Expansion:** Add synonyms, handle stemming variations
- **Fuzzy Matching:** Levenshtein distance, phonetic algorithms (Soundex, Metaphone)
- **Spell Correction:** Suggest corrections using edit distance and frequency

Ranking Algorithm

- **TF-IDF:** Term frequency × inverse document frequency for relevance
- **BM25:** Improved probabilistic ranking function
- **PageRank:** Link-based authority for web documents
- **Machine Learning:** LambdaMART or neural ranking models trained on click data

10. Design a video streaming platform like YouTube or Netflix. Discuss video encoding, adaptive bitrate streaming, CDN integration, and handling live streams.

System Architecture

- **Upload Service:** Handle video uploads, validation, metadata extraction
- **Transcoding Pipeline:** Convert videos to multiple formats and resolutions
- **Storage:** Object storage (S3) for video files, database for metadata
- **CDN:** Distribute content globally for low-latency delivery
- **Streaming Server:** Serve video chunks using HLS or DASH protocols

Database Schema

```
CREATE TABLE videos (
  id BIGINT PRIMARY KEY,
  title VARCHAR(255),
  user_id BIGINT,
  duration INT,
  status ENUM('processing','ready','failed'),
  views BIGINT DEFAULT 0,
  created_at TIMESTAMP
);
CREATE TABLE video_files (
  video_id BIGINT,
  resolution VARCHAR(10),
  bitrate INT,
  file_path VARCHAR(500),
  PRIMARY KEY (video_id, resolution)
);
```

Video Processing Pipeline

- **Transcoding:** FFmpeg to generate multiple bitrates (360p, 720p, 1080p, 4K)
- **Segmentation:** Split video into small chunks (2-10 seconds) for streaming
- **Thumbnail Generation:** Extract keyframes for preview thumbnails
- **Queue-based:** Use message queue for async processing, scale workers horizontally

Adaptive Bitrate Streaming

- **HLS/DASH:** Client requests manifest file, selects quality based on bandwidth
- **Quality Ladder:** Encode at multiple bitrates with appropriate resolutions
- **CDN Caching:** Cache video segments at edge, manifest files have short TTL

Live Streaming

- **Ingestion:** RTMP protocol from encoder to streaming server
- **Low Latency:** Use CMAF, LL-HLS, or WebRTC for sub-second latency
- **Scaling:** Distribute viewers across multiple edge servers

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 Employee table.

Solution

Use a subquery with **LIMIT and OFFSET** or the **MAX function with a WHERE clause**:

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

Alternative using LIMIT/OFFSET:

```
SELECT DISTINCT salary AS second_highest
FROM Employee
ORDER BY salary DESC
LIMIT 1 OFFSET 1;
```

Key considerations:

- Handle NULL values when no second highest exists
- Use DISTINCT to avoid duplicate salary values
- Consider performance with proper indexing on salary column

2. How would you debug a slow-running SQL query in production?

Debugging Approach

Step-by-step debugging process:

- **EXPLAIN/EXPLAIN ANALYZE:** Examine the query execution plan to identify full table scans, missing indexes, or inefficient joins
- **Check indexes:** Verify that appropriate indexes exist on WHERE, JOIN, and ORDER BY columns
- **Query profiling:** Use database-specific tools (MySQL slow query log, PostgreSQL pg_stat_statements, SQL Server Profiler)
- **Analyze statistics:** Ensure table statistics are up-to-date for the query optimizer
- **Monitor locks:** Check for blocking queries or deadlocks using system views
- **Review query structure:** Look for N+1 queries, unnecessary subqueries, or missing WHERE clauses

```
EXPLAIN ANALYZE
SELECT * FROM orders o
JOIN customers c ON o.customer_id = c.id
WHERE o.created_at > '2024-01-01';
```

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

Finding Duplicates

Use **GROUP BY with HAVING** to identify duplicates:

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

To retrieve full duplicate records:

```
SELECT u.*
```

```
FROM users u
INNER JOIN (
  SELECT email FROM users
  GROUP BY email HAVING COUNT(*) > 1
) dup ON u.email = dup.email
ORDER BY u.email;
```

Advanced technique: Use window functions to identify and remove duplicates while keeping one record:

```
DELETE FROM users
WHERE id IN (
  SELECT id FROM (
    SELECT id, ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) as rn
    FROM users
  ) t WHERE rn > 1
);
```

4. Explain the difference between WHERE and HAVING clauses. When would you use each?

WHERE vs HAVING

WHERE clause:

- Filters rows **before** grouping and aggregation
- Cannot use aggregate functions (SUM, COUNT, AVG)
- Applied to individual rows
- More efficient as it reduces data early

HAVING clause:

- Filters groups **after** GROUP BY aggregation
- Can use aggregate functions
- Applied to grouped results
- Used specifically with GROUP BY

```
SELECT department, AVG(salary) as avg_sal
FROM employees
WHERE hire_date > '2020-01-01'
GROUP BY department
HAVING AVG(salary) > 50000;
```

In this example, WHERE filters individual employees before grouping, while HAVING filters departments after calculating averages.

5. How do you handle deadlocks in SQL databases?

Deadlock Prevention and Handling

Detection and monitoring:

- Enable deadlock logging and monitoring
- Use system views: sys.dm_tran_locks (SQL Server), pg_locks (PostgreSQL)
- Analyze deadlock graphs to identify conflicting queries

Prevention strategies:

- **Consistent lock order:** Always access tables in the same order across transactions
- **Keep transactions short:** Minimize transaction duration to reduce lock holding time
- **Use appropriate isolation levels:** READ COMMITTED instead of SERIALIZABLE when possible
- **Proper indexing:** Reduce lock escalation with appropriate indexes
- **Retry logic:** Implement exponential backoff for deadlock victims

```
BEGIN TRANSACTION;
SET LOCK_TIMEOUT 5000;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

6. Write a query to calculate a running total or cumulative sum.

Running Total with Window Functions

Use **window functions** for efficient running totals:

```
SELECT
  order_date,
  amount,
  SUM(amount) OVER (
    ORDER BY order_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS running_total
FROM orders
ORDER BY order_date;
```

Partitioned running total (by category):

```
SELECT
  category,
  order_date,
  amount,
  SUM(amount) OVER (
    PARTITION BY category
    ORDER BY order_date
  ) AS category_running_total
FROM orders;
```

Advantages: Window functions are more efficient than correlated subqueries and provide cleaner syntax for analytical queries.

7. How do you optimize a query that involves multiple JOIN operations?

JOIN Optimization Techniques

Key optimization strategies:

- **Index foreign keys:** Ensure all JOIN columns have appropriate indexes
- **JOIN order matters:** Start with smallest result set; modern optimizers handle this but explicit ordering helps
- **Use INNER JOIN when possible:** More efficient than OUTER JOINs
- **Filter early:** Apply WHERE conditions before JOINs when possible
- **Avoid SELECT *:** Specify only needed columns to reduce data transfer
- **Consider denormalization:** For frequently joined tables, strategic denormalization can improve performance

```
SELECT o.id, c.name, p.product_name
FROM orders o
INNER JOIN customers c ON o.customer_id = c.id
INNER JOIN order_items oi ON o.id = oi.order_id
INNER JOIN products p ON oi.product_id = p.id
WHERE o.created_at > '2024-01-01'
AND c.status = 'active';
```

Use EXPLAIN to verify the optimizer chooses efficient join algorithms (hash join, merge join, nested loop).

8. Explain the difference between UNION and UNION ALL. When would you use each?

UNION vs UNION ALL

UNION:

- Combines results and **removes duplicates**
- Performs implicit DISTINCT operation
- Slower due to duplicate elimination
- Requires sorting/hashing to identify duplicates

UNION ALL:

- Combines results and **keeps all duplicates**
- No duplicate checking
- Faster and more efficient
- Preferred when duplicates are impossible or acceptable

```
SELECT customer_id FROM orders_2023
UNION ALL
SELECT customer_id FROM orders_2024;
```

Best practice: Use UNION ALL by default unless you specifically need duplicate removal. If you know the datasets are mutually exclusive, UNION ALL avoids unnecessary overhead.

Requirements: Both queries must have the same number of columns with compatible data types.

9. How would you implement pagination efficiently for large datasets?

Efficient Pagination Strategies

Method 1: Offset-based (simple but slow for large offsets):

```
SELECT * FROM products
ORDER BY id
LIMIT 20 OFFSET 1000;
```

Problem: Database must scan and skip first 1000 rows, inefficient for large offsets.

Method 2: Keyset/Cursor-based (recommended for large datasets):

```
SELECT * FROM products
WHERE id > 1000
ORDER BY id
LIMIT 20;
```

Advantages of keyset pagination:

- Constant performance regardless of page depth
- Uses index efficiently with WHERE clause
- No need to scan skipped rows
- Handles real-time data changes better

Implementation tip: Return the last ID with each page to use as cursor for next request. Combine with created_at for time-based ordering.

10. Write a query to pivot rows into columns or unpivot columns into rows.

Pivot and Unpivot Operations

Pivot (rows to columns) using CASE:

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

Unpivot (columns to rows) using UNION ALL:

```
SELECT product_id, 'Jan' AS month, jan_sales AS sales FROM sales
UNION ALL
SELECT product_id, 'Feb', feb_sales FROM sales
UNION ALL
SELECT product_id, 'Mar', mar_sales FROM sales;
```

Modern approach: PostgreSQL supports CROSSTAB, SQL Server has PIVOT/UNPIVOT operators. For dynamic pivoting, consider generating SQL dynamically or using reporting tools.

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 poorly performing SQL query. What was your approach?

Situation: Our e-commerce platform's product search page was timing out during peak hours, with queries taking over 30 seconds to return results.

Task: I was assigned to identify and resolve the performance bottleneck affecting customer experience and causing cart abandonment.

Action: I analyzed the query execution plan and discovered missing indexes on frequently joined columns. I added composite indexes on (category_id, price, created_at) and rewrote the query to eliminate a correlated subquery, replacing it with a JOIN. I also implemented query result caching for common searches.

```
-- Before: Correlated subquery
SELECT p.* FROM products p
WHERE price < (SELECT AVG(price) FROM products WHERE category_id = p.category_id);

-- After: JOIN with indexed columns
SELECT p.* FROM products p
JOIN (SELECT category_id, AVG(price) as avg_price FROM products GROUP BY category_id) avg_p
ON p.category_id = avg_p.category_id WHERE p.price < avg_p.avg_price;
```

Result: Query execution time dropped from 30+ seconds to under 500ms, page load times improved by 85%, and we saw a 12% increase in conversion rates during peak traffic periods.

2. Describe a situation where you had to handle a database outage or critical data issue in production.

Situation: At 2 AM, our monitoring system alerted that the primary database server went down due to disk space exhaustion, causing complete service unavailability for our SaaS application.

Task: As the on-call database engineer, I needed to restore service immediately while preserving data integrity and identifying the root cause.

Action: I immediately failed over to our read replica, promoting it to primary to restore read access within 5 minutes. I identified that log files from an abandoned backup process were consuming disk space. I cleared the logs, expanded disk capacity, and implemented automated log rotation. I then conducted a full data integrity check and updated our monitoring to alert on disk usage at 70% threshold.

Result: Service was restored within 15 minutes with zero data loss. I documented the incident in a post-mortem, implemented preventive monitoring alerts, and established a runbook for similar scenarios. We haven't experienced a similar outage in the 18 months since.

3. Give an example of how you've handled database schema changes in a production environment with zero downtime.

Situation: We needed to split a monolithic 'users' table into separate 'users' and 'user_profiles' tables to improve performance, but our application served 50,000+ concurrent users with strict uptime SLAs.

Task: I was responsible for designing and executing a migration strategy that would allow the schema change without any service interruption or data inconsistency.

Action: I implemented a multi-phase approach:

- Phase 1: Created the new user_profiles table and added dual-write logic in the application layer
- Phase 2: Backfilled existing data using batched migrations during low-traffic hours
- Phase 3: Updated application code to read from both tables with fallback logic
- Phase 4: Switched reads to the new table after validation
- Phase 5: Removed old columns after a 2-week monitoring period

Result: The migration completed successfully over 3 weeks with zero downtime, no data loss, and imperceptible impact on users. Query performance improved by 40% and the modular schema enabled faster feature development.

4. Tell me about a time when you had to make a difficult trade-off decision regarding database design or architecture.

Situation: While designing a real-time analytics dashboard, I faced a conflict between maintaining strict data consistency (ACID compliance) and achieving the sub-second response times required by stakeholders.

Task: I needed to decide between implementing a traditional relational approach with complex aggregations or adopting an eventually consistent denormalized design with materialized views.

Action: I conducted a thorough analysis presenting both options to stakeholders with concrete metrics. I recommended a hybrid approach: maintaining transactional integrity for critical financial data in PostgreSQL while using a read-optimized materialized view refreshed every 30 seconds for dashboard queries. I implemented triggers to update aggregates and added clear UI indicators showing data freshness timestamps.

```
CREATE MATERIALIZED VIEW analytics_summary AS
SELECT date, product_id, SUM(revenue) as total_revenue,
  COUNT(*) as order_count
FROM orders WHERE status = 'completed'
GROUP BY date, product_id;

CREATE INDEX idx_analytics_date ON analytics_summary(date);
```

Result: Dashboard load times decreased from 8 seconds to 300ms, user satisfaction scores increased by 35%, and we maintained data accuracy within acceptable business tolerances while preserving transactional integrity where it mattered most.

5. Describe a situation where you identified and resolved a data integrity issue.

Situation: During a routine audit, I discovered that our order processing system had created approximately 3,000 orphaned payment records over six months due to a race condition in our distributed transaction handling.

Task: I needed to identify the root cause, fix the bug, reconcile the inconsistent data, and prevent future occurrences without disrupting ongoing operations.

Action: I first wrote diagnostic queries to identify all orphaned records and their patterns. I discovered the issue occurred when payment confirmations arrived before order creation completed. I implemented database constraints and application-level distributed locks using Redis. For remediation, I created a reconciliation script that matched orphaned payments to orders using transaction IDs and timestamps, processing them in batches with manual review for edge cases.

```
-- Diagnostic query
SELECT p.* FROM payments p
LEFT JOIN orders o ON p.order_id = o.id
WHERE o.id IS NULL AND p.created_at > '2023-01-01';

-- Added constraint
ALTER TABLE payments ADD CONSTRAINT fk_order
FOREIGN KEY (order_id) REFERENCES orders(id);
```

Result: Successfully reconciled 2,847 records automatically and manually reviewed 153 edge cases. Implemented foreign key constraints and distributed locking that prevented any recurrence. Recovered $47,000 in previously unprocessed payments.

6. Tell me about a time when you had to mentor or guide junior developers on database

best practices.

Situation: A junior developer on my team repeatedly wrote queries that caused table locks during business hours, resulting in application timeouts and customer complaints.

Task: I needed to educate the developer on database performance and locking mechanisms while building their confidence and maintaining a positive learning environment.

Action: I scheduled a series of pair-programming sessions where we reviewed their queries together. I explained transaction isolation levels, the importance of indexes, and how to read execution plans. I created a checklist document covering query best practices and set up a code review process where I provided constructive feedback. I also introduced them to tools like EXPLAIN ANALYZE and set up a staging environment where they could safely test queries under load.

```
-- Taught the difference
-- Bad: Locks entire table
UPDATE orders SET status = 'processed' WHERE user_id = 123;

-- Good: Uses index, minimal locking
UPDATE orders SET status = 'processed'
WHERE id IN (SELECT id FROM orders WHERE user_id = 123 LIMIT 100);
```

Result: Within a month, the developer was writing efficient queries independently and even identified optimization opportunities in legacy code. Production incidents related to database locks decreased by 90%, and the developer later became a resource for other team members on database topics.

7. Describe a time when you had to work with a legacy database system and modernize it.

Situation: I inherited a 10-year-old MySQL 5.5 database with no documentation, hundreds of stored procedures with embedded business logic, and no proper version control, supporting a critical billing system.

Task: I was tasked with modernizing the system to improve maintainability, upgrade to MySQL 8.0, and migrate business logic to the application layer without disrupting monthly billing cycles.

Action: I started by documenting the existing schema and mapping all dependencies using automated tools. I created a phased migration plan: first upgrading to MySQL 5.7 in a test environment, identifying compatibility issues, then refactoring the most critical stored procedures into application services. I implemented comprehensive integration tests to ensure behavioral consistency. I used feature flags to gradually shift traffic from stored procedures to application logic, monitoring closely for discrepancies.

Result: Successfully upgraded to MySQL 8.0 over 6 months with zero billing errors. Reduced stored procedure count from 300+ to 40 essential ones. Deployment time decreased from 4 hours to 20 minutes, and new developers could onboard 60% faster with logic now in version-controlled application code. Database CPU usage decreased by 25%.

8. Tell me about a challenging database performance problem you diagnosed and solved.

Situation: Our reporting system experienced severe degradation every Monday morning, with queries timing out and causing cascading failures across dependent services. The issue was intermittent and difficult to reproduce.

Task: I was assigned to identify the root cause and implement a permanent solution while maintaining system availability during peak business hours.

Action: I enabled query logging and performance monitoring to capture Monday morning traffic patterns. I discovered that a weekly batch job was running without proper indexes, causing full table scans on a 500GB table during peak hours. I analyzed the query patterns and created a covering index. I also rescheduled the batch job to run during off-peak hours and implemented query timeout limits to prevent resource exhaustion. Additionally, I set up read replicas to offload reporting queries from the primary database.

```
-- Created covering index
CREATE INDEX idx_weekly_report ON transactions
(created_at, status, user_id)
```

```
INCLUDE (amount, category)
WHERE created_at >= CURRENT_DATE - INTERVAL '7 days';
```

Result: Monday morning query times dropped from 45+ seconds to under 2 seconds. System stability improved with zero timeout incidents in subsequent months. The read replica strategy reduced primary database load by 40%, and I documented the findings in a capacity planning guide.

9. Give an example of how you've implemented database security measures or handled a security concern.

Situation: During a security audit, our team discovered that multiple application services were connecting to the production database using a shared root account with full privileges, violating compliance requirements and creating significant risk.

Task: I was responsible for implementing proper access controls, principle of least privilege, and ensuring compliance with SOC 2 requirements without breaking existing application functionality.

Action: I conducted an audit of all database operations by service and created individual service accounts with minimal required permissions. I implemented role-based access control (RBAC), created separate read-only accounts for reporting tools, and enforced SSL/TLS for all connections. I rotated all credentials and stored them in a secrets management system (HashiCorp Vault). I also enabled audit logging for all DDL operations and privileged access.

```
-- Created restricted service accounts
CREATE USER 'order_service'@'%' IDENTIFIED BY 'secure_pass';
GRANT SELECT, INSERT, UPDATE ON shop.orders TO 'order_service'@'%';
GRANT SELECT ON shop.products TO 'order_service'@'%';

CREATE USER 'reporting_readonly'@'%' IDENTIFIED BY 'secure_pass';
GRANT SELECT ON shop.* TO 'reporting_readonly'@'%';
```

Result: Successfully implemented least-privilege access across 15 services with zero production incidents. Passed SOC 2 audit requirements, reduced attack surface significantly, and established a security baseline that became standard for all new services. Audit logs enabled us to detect and prevent an unauthorized access attempt three months later.

10. Describe a situation where you had to balance technical debt against new feature development in database architecture.

Situation: Our startup's database had accumulated significant technical debt with denormalized tables, missing foreign keys, and inconsistent naming conventions. Meanwhile, the product team was pushing for rapid feature development to meet investor milestones.

Task: As the senior database engineer, I needed to advocate for addressing technical debt while supporting business objectives and maintaining team velocity.

Action: I quantified the impact of technical debt by measuring deployment time, bug frequency, and developer productivity metrics. I presented data to leadership showing that technical debt was costing 30% of development time. I proposed a balanced approach: allocating 20% of each sprint to incremental refactoring while delivering new features. I created a prioritized technical debt backlog, focusing first on high-impact items like adding foreign key constraints and normalizing the most problematic tables. I also implemented database migration patterns and documentation standards for new development.

Result: Over 6 months, we reduced critical technical debt by 60% while delivering all planned features. Deployment-related bugs decreased by 45%, and new developer onboarding time reduced from 3 weeks to 1 week. The structured approach gained leadership buy-in for ongoing maintenance, and we established technical debt as a permanent backlog item with dedicated capacity.