Here are \*\*5 mock interviews\*\* structured around the topics you mentioned, using the previous experience as a template:

---

### \*\*Mock Interview 1: Python & Message Queues\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: Fix a Python function that processes messages from a queue but skips every 3rd element.

```python

def process\_queue(messages):

result = []

for i in range(0, len(messages)):

if i % 3 != 0:

result.append(messages[i])

return result

```

\*\*Issues\*\*: Syntax errors (`range` bracket, `messages[i])`), off-by-one logic (indices start at 0).

\*\*Fixed Code\*\*: Adjust loop to skip indices where `(i+1) % 3 == 0` to align with "every 3rd element" logic.

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: A message queue (e.g., Kafka/RabbitMQ) is experiencing high latency. Walk through debugging steps.

- Check consumer lag, broker health, network bottlenecks.

- Validate partitioning/load balancing.

- Monitor disk I/O for persistent queues.

\*\*3. Database vs. Message Queue\*\*

- \*\*Message Queue\*\*: Optimized for transient data, pub/sub, async processing.

- \*\*Database\*\*: Persistent storage, ACID transactions, query flexibility.

\*\*4. Security\*\*

- How to secure a message queue?

- Use TLS for encryption, SASL for authentication, ACLs for topic access.

\*\*5. Cron Jobs\*\*

- How to ensure cron jobs don’t overload a message queue?

- Throttle job frequency, use bulk processing, monitor queue depth.

---

### \*\*Mock Interview 2: Batch Processing & SQL\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: Write a Python script to aggregate daily sales data into a weekly batch. Handle missing dates.

\*\*Key Points\*\*: Use `pandas` for resampling, fill missing values with interpolation.

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: A batch job processing 1M records is slow. Debug steps:

- Check indexing on database tables.

- Optimize SQL queries (avoid `SELECT \*`, use `EXPLAIN`).

- Parallelize processing (e.g., multiprocessing).

\*\*3. SQL Query\*\*

- Write a query to find the 2nd highest salary in a table.

```sql

SELECT MAX(salary) FROM employees WHERE salary < (SELECT MAX(salary) FROM employees);

```

\*\*4. Batch vs. Stream Processing\*\*

- \*\*Batch\*\*: Scheduled, large datasets (e.g., ETL jobs).

- \*\*Stream\*\*: Real-time, low-latency (e.g., Kafka, Flink).

\*\*5. Database Index\*\*

- When to avoid indexes?

- On small tables, or columns with high write/low read usage.

---

### \*\*Mock Interview 3: Stream Processing & HTTP\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: Debug a Python generator for real-time sensor data that stops prematurely.

\*\*Issue\*\*: Missing `yield` statement or improper loop termination.

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: HTTP API for streaming data returns 503 errors.

- Check server load (CPU/memory), rate limiting, timeout configurations.

- Validate connection pooling and thread limits.

\*\*3. HTTP vs. HTTPS\*\*

- \*\*HTTPS\*\*: Encrypted (TLS), uses port 443, requires certificates.

- \*\*HTTP\*\*: Plaintext, port 80, no encryption.

\*\*4. HTTP Methods & Status Codes\*\*

- \*\*Methods\*\*: `GET` (read), `POST` (create), `PUT` (update), `DELETE`.

- \*\*Status Codes\*\*: 200 (OK), 201 (Created), 400 (Bad Request), 500 (Server Error).

\*\*5. DNS & Handshake\*\*

- Describe a DNS lookup flow.

- What happens in a TLS handshake?

- Cipher negotiation, certificate validation, key exchange.

---

### \*\*Mock Interview 4: Database Indexes & HTTPS\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: Simulate a database index in Python using a dictionary for fast lookups.

```python

index = {}

for idx, row in enumerate(data):

index[row['id']] = idx # O(1) lookup time

```

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: Database queries are slow despite indexing.

- Check for index fragmentation.

- Ensure queries use indexed columns (avoid functions on columns).

- Analyze query plans for full table scans.

\*\*3. Cookies & Headers\*\*

- How to set a cookie in an HTTP response?

- Header: `Set-Cookie: user\_id=123; Secure; HttpOnly`.

- What’s the `Cache-Control` header used for?

- Define caching rules (e.g., `max-age=3600`).

\*\*4. Database Index Types\*\*

- \*\*B-Tree\*\*: Default for most databases, good for range queries.

- \*\*Hash\*\*: Fast equality checks (no ranges).

- \*\*Composite\*\*: Index on multiple columns.

\*\*5. Cron Jobs\*\*

- How to log output of a cron job without filling disk space?

- Redirect output to syslog or a log rotation tool (e.g., `logrotate`).

---

### \*\*Mock Interview 5: File System & Caching\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: Write a Python script to find duplicate files in a directory using MD5 hashes.

\*\*Key Points\*\*: Use `os.walk`, hash file contents, track duplicates in a dictionary.

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: A file system-based cache is causing high disk I/O.

- Switch to in-memory caching (e.g., Redis).

- Optimize file access patterns (batch reads/writes).

\*\*3. Cache Headers\*\*

- `ETag`: Validate cached content freshness.

- `Last-Modified`: Timestamp-based caching.

\*\*4. Database vs. File System Security\*\*

- \*\*File System\*\*: OS permissions (e.g., `chmod 600` for private files).

- \*\*Database\*\*: Role-based access (e.g., `GRANT SELECT ON table TO user`).

\*\*5. Stream Processing\*\*

- How does stream processing handle late data?

- Use watermarks (e.g., Apache Beam) or sliding windows.

---

### \*\*Key Takeaways\*\*

- \*\*Coding\*\*: Focus on edge cases (e.g., missing dates, duplicates).

- \*\*Debugging\*\*: Start with monitoring, then drill into code/config.

- \*\*Concepts\*\*: Contrast technologies (batch vs. stream, DB vs. file system).

- \*\*Security\*\*: Always consider encryption (HTTPS/TLS) and access controls.

Let me know if you’d like deeper dives into any topic! 🚀

---

### \*\*Mock Interview 1: Python Script Efficiency & Database Indexing\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: A Python script processes large CSV files but crashes due to high memory usage.

```python

def process\_csv(file\_path):

data = []

with open(file\_path, 'r') as f:

reader = csv.reader(f)

for row in reader:

data.append(row) # Loads entire file into memory

# Processing logic here

```

\*\*Task\*\*: Optimize the script to handle large files.

\*\*Solution\*\*: Use generators to process rows incrementally:

```python

def process\_csv(file\_path):

with open(file\_path, 'r') as f:

reader = csv.reader(f)

for row in reader:

# Process row immediately

process\_row(row)

```

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: A database query on an indexed column `user\_id` is slow.

\*\*Debugging Steps\*\*:

- Confirm the index exists (`SHOW INDEXES FROM table;`).

- Check if the query uses the index (`EXPLAIN SELECT ...`).

- Ensure the query doesn’t apply functions to `user\_id` (e.g., `WHERE LOWER(user\_id) = 'abc'`).

- Rebuild the index if fragmented.

\*\*3. Conceptual Questions\*\*

- \*\*When should you avoid database indexes?\*\*

- On small tables, frequently updated columns, or columns with low cardinality (e.g., boolean fields).

- \*\*What does HTTP 429 mean?\*\*

- "Too Many Requests": The client is rate-limited.

---

### \*\*Mock Interview 2: Service Downtime & HTTPS Issues\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: A Flask app returns 500 errors when fetching user data.

```python

@app.route('/users')

def get\_users():

users = db.execute('SELECT \* FROM users').fetchall() # No error handling

return jsonify(users)

```

\*\*Task\*\*: Add error handling for database disconnections.

\*\*Solution\*\*:

```python

try:

users = db.execute('SELECT \* FROM users').fetchall()

except DatabaseError as e:

return jsonify({"error": "Database unavailable"}), 503

```

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: Users report "SSL Handshake Failed" errors.

\*\*Debugging Steps\*\*:

- Check certificate expiry (`openssl x509 -dates -in cert.pem`).

- Verify TLS version compatibility (e.g., server only supports TLS 1.3).

- Test with `curl -v` to inspect handshake details.

\*\*3. Conceptual Questions\*\*

- \*\*Describe the TLS handshake process.\*\*

- Client hello → Server hello (with cert) → Key exchange → Encrypted session.

- \*\*What’s the purpose of a CNAME record?\*\*

- Maps a domain alias to another domain (e.g., `www.example.com` → `example.com`).

---

### \*\*Mock Interview 3: Message Queue Backlog & Stream Processing\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: A Kafka consumer misses messages during peak loads.

```python

consumer = KafkaConsumer('topic')

for message in consumer:

process(message) # No batching or parallelization

```

\*\*Task\*\*: Improve throughput using batch processing.

\*\*Solution\*\*:

```python

from concurrent.futures import ThreadPoolExecutor

def process\_batch(messages):

with ThreadPoolExecutor() as executor:

executor.map(process, messages)

batch = []

for message in consumer:

batch.append(message)

if len(batch) >= 100:

process\_batch(batch)

batch = []

```

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: Kafka consumer lag spikes.

\*\*Debugging Steps\*\*:

- Check consumer group status (`kafka-consumer-groups --describe`).

- Monitor broker network latency.

- Scale consumers or increase partitions.

\*\*3. Conceptual Questions\*\*

- \*\*Batch vs. Stream Processing\*\*:

- Batch: Daily sales reports. Stream: Real-time fraud detection.

- \*\*What is idempotency in message queues?\*\*

- Processing the same message multiple times has the same result.

---

### \*\*Mock Interview 4: File System vs. Database for User Uploads\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: A script stores user-uploaded images as BLOBs in a database, causing slow queries.

\*\*Task\*\*: Rewrite to store files in a file system and save paths in the database.

\*\*Solution\*\*:

```python

def save\_image(file):

path = f"/uploads/{file.filename}"

with open(path, 'wb') as f:

f.write(file.read())

db.execute('INSERT INTO images (path) VALUES (?)', (path,))

```

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: Users can’t access uploaded files due to permission errors.

\*\*Debugging Steps\*\*:

- Check file system permissions (`ls -l /uploads`).

- Ensure the web server (e.g., Nginx) has read access.

\*\*3. Conceptual Questions\*\*

- \*\*Why store files in a file system over a database?\*\*

- Faster I/O, cheaper scaling, and easier CDN integration.

- \*\*What’s SQL injection? How to prevent it?\*\*

- Injecting malicious SQL via inputs. Use parameterized queries.

---

### \*\*Mock Interview 5: Cron Job Failures & Caching\*\*

\*\*1. Coding Challenge\*\*

\*\*Problem\*\*: A Python cron job logs to a file but fails intermittently.

```python

def main():

# Logic here

print("Job completed") # Output to stdout, not captured

```

\*\*Task\*\*: Ensure logs are written to a file reliably.

\*\*Solution\*\*:

```python

import logging

logging.basicConfig(filename='/logs/job.log', level=logging.INFO)

def main():

try:

# Logic

logging.info("Job completed")

except Exception as e:

logging.error(str(e))

```

\*\*2. Troubleshooting\*\*

\*\*Scenario\*\*: A website serves stale content despite updated files.

\*\*Debugging Steps\*\*:

- Check `Cache-Control` headers (e.g., `max-age=86400`).

- Force CDN cache invalidation.

\*\*3. Conceptual Questions\*\*

- \*\*Best practices for cron jobs?\*\*

- Log outputs, set `MAILTO`, test in non-production.

- \*\*What does HTTP 304 mean?\*\*

- "Not Modified": The client’s cached version is still valid.

---

### \*\*Key Takeaways\*\*

- \*\*Foundational Focus\*\*: Prioritize practical debugging (e.g., memory leaks, error handling).

- \*\*Real-World Alignment\*\*: Simulate outages (TLS failures, cron job crashes) and scalability (message queues).

- \*\*Conceptual Clarity\*\*: Explain trade-offs (DB vs. file system) and standards (HTTP status codes).

Let me know if you’d like to drill deeper into any scenario! 🔧

I've extracted all the troubleshooting scenarios from the document. Let me analyze each one from a database and web expert perspective, including what clarifying questions you should ask before beginning your debugging process.

## Troubleshooting Scenario 1: Message Queue High Latency

\*\*Scenario\*\*: A message queue (Kafka/RabbitMQ) experiencing high latency.

\*\*Debugging Steps\*\*:

- Check consumer lag, broker health, network bottlenecks

- Validate partitioning/load balancing

- Monitor disk I/O for persistent queues

\*\*Analysis of Debugging Logic\*\*:

This approach follows a "narrow down the bottleneck" methodology. The steps move from application layer (consumer lag) to infrastructure (broker health, network) to hardware (disk I/O). This aligns with the data flow path in message queues where messages flow from producers through brokers to consumers.

\*\*Clarifying Questions to Ask\*\*:

1. "When did the latency begin? Was there a deployment or configuration change beforehand?"

2. "What's the current message throughput compared to normal operations?"

3. "Are all topics/queues affected or only specific ones?"

4. "What metrics do we have available? Do we have monitoring for consumer lag and broker health?"

5. "What's the message retention policy and current disk usage on the brokers?"

## Troubleshooting Scenario 2: Slow Batch Job

\*\*Scenario\*\*: A batch job processing 1M records is slow.

\*\*Debugging Steps\*\*:

- Check indexing on database tables

- Optimize SQL queries (avoid `SELECT \*`, use `EXPLAIN`)

- Parallelize processing (multiprocessing)

\*\*Analysis of Debugging Logic\*\*:

This follows a "database first, application second" approach. It assumes the bottleneck is likely in data retrieval before considering application-side optimizations. This makes sense since database operations typically dominate performance in batch processing.

\*\*Clarifying Questions to Ask\*\*:

1. "Has this job run successfully before? If so, what changed?"

2. "What's the execution plan for the queries? Can we see the EXPLAIN output?"

3. "How are we currently retrieving data - in a single transaction or chunked?"

4. "What indexes currently exist on the tables being queried?"

5. "What's the server load during job execution? Are we CPU, memory, or I/O bound?"

6. "Are there any locks or blocking queries during execution?"

## Troubleshooting Scenario 3: HTTP API Returning 503 Errors

\*\*Scenario\*\*: HTTP API for streaming data returns 503 errors.

\*\*Debugging Steps\*\*:

- Check server load (CPU/memory), rate limiting, timeout configurations

- Validate connection pooling and thread limits

\*\*Analysis of Debugging Logic\*\*:

This approach focuses on resource exhaustion, which is appropriate since 503 means "Service Unavailable" - typically caused by server overload or maintenance. The debugging path examines both hardware resources and software configuration limits.

\*\*Clarifying Questions to Ask\*\*:

1. "When did the 503 errors start occurring? Is there a pattern to when they happen?"

2. "What's the current traffic volume compared to normal?"

3. "Are all endpoints affected or just the streaming endpoints?"

4. "What's our current connection pool configuration? Are we seeing connection timeouts?"

5. "Do we have monitoring data for CPU, memory, and thread usage?"

6. "Are there any scheduled maintenance tasks or backup jobs running during the error periods?"

## Troubleshooting Scenario 4: Slow Database Queries Despite Indexing

\*\*Scenario\*\*: Database queries are slow despite indexing.

\*\*Debugging Steps\*\*:

- Check for index fragmentation

- Ensure queries use indexed columns (avoid functions on columns)

- Analyze query plans for full table scans

\*\*Analysis of Debugging Logic\*\*:

This follows a "verify index effectiveness" approach. It acknowledges that simply having indexes isn't enough - they need to be well-maintained and actually used by the query optimizer. The steps progress from index health to query structure to execution plan analysis.

\*\*Clarifying Questions to Ask\*\*:

1. "When was the last time indexes were rebuilt or reorganized?"

2. "Can you show me the exact queries that are running slowly?"

3. "What's the data distribution in the indexed columns? Do we have skewed data?"

4. "Are there any functions or expressions applied to the indexed columns in the WHERE clauses?"

5. "What does the execution plan show? Are we seeing index seeks or scans?"

6. "How large are the tables, and what's the current database server load?"

## Troubleshooting Scenario 5: File System Cache Causing High Disk I/O

\*\*Scenario\*\*: A file system-based cache is causing high disk I/O.

\*\*Debugging Steps\*\*:

- Switch to in-memory caching (Redis)

- Optimize file access patterns (batch reads/writes)

\*\*Analysis of Debugging Logic\*\*:

This approach suggests both architectural changes (switch to Redis) and optimization of the current approach. It recognizes that file-based caching inherently involves I/O operations, which can become a bottleneck at scale.

\*\*Clarifying Questions to Ask\*\*:

1. "What's the current cache hit/miss ratio?"

2. "How large are the cached objects and how frequently do they change?"

3. "What's our current eviction policy and cache size limit?"

4. "Are we using any kind of buffering or batch operations currently?"

5. "What's the system's I/O capacity and current utilization?"

6. "Do we have the infrastructure available to implement Redis or another in-memory solution?"

## Troubleshooting Scenario 6: Database Query on Indexed Column is Slow

\*\*Scenario\*\*: A database query on an indexed column `user\_id` is slow.

\*\*Debugging Steps\*\*:

- Confirm the index exists (`SHOW INDEXES FROM table;`)

- Check if the query uses the index (`EXPLAIN SELECT...`)

- Ensure the query doesn't apply functions to `user\_id`

- Rebuild the index if fragmented

\*\*Analysis of Debugging Logic\*\*:

This follows a methodical "verify, validate, optimize" approach focused specifically on index usage problems. It starts with basic existence verification before diving deeper into how the index is being used in queries.

\*\*Clarifying Questions to Ask\*\*:

1. "What's the exact query that's running slowly?"

2. "What's the data type of the user\_id column?"

3. "How many rows are in the table, and what's the cardinality of the user\_id column?"

4. "Are there any implicit conversions happening in the query?"

5. "Is this a sudden performance degradation or has it always been slow?"

6. "What's the database server's current load and memory allocation for query cache?"

## Troubleshooting Scenario 7: Users Report "SSL Handshake Failed" Errors

\*\*Scenario\*\*: Users report "SSL Handshake Failed" errors.

\*\*Debugging Steps\*\*:

- Check certificate expiry (`openssl x509 -dates -in cert.pem`)

- Verify TLS version compatibility

- Test with `curl -v` to inspect handshake details

\*\*Analysis of Debugging Logic\*\*:

This approach follows the TLS handshake process, examining certificate issues first (the most common cause) before checking protocol compatibility. The final step provides detailed diagnosis information.

\*\*Clarifying Questions to Ask\*\*:

1. "Are all users affected or only users on certain browsers/devices?"

2. "When did these errors start occurring? Was there a recent certificate change?"

3. "What TLS versions do we support on our server, and what's the minimum our clients support?"

4. "Are we using any TLS features that might not be universally supported?"

5. "Can you provide the exact error message the users are seeing?"

6. "Have we recently changed our cipher suite preferences?"

## Troubleshooting Scenario 8: Kafka Consumer Lag Spikes

\*\*Scenario\*\*: Kafka consumer lag spikes.

\*\*Debugging Steps\*\*:

- Check consumer group status (`kafka-consumer-groups --describe`)

- Monitor broker network latency

- Scale consumers or increase partitions

\*\*Analysis of Debugging Logic\*\*:

This follows a "verify current state, check infrastructure, then scale" approach. It first diagnoses the extent of the problem before looking at network issues, and finally suggests scaling solutions.

\*\*Clarifying Questions to Ask\*\*:

1. "Which topics and partitions are experiencing lag?"

2. "What's the message production rate versus consumption rate?"

3. "Are consumers rebalancing frequently? Any consumer group membership changes?"

4. "What's the size of the messages being processed?"

5. "Are there any errors in the consumer logs?"

6. "What's the current consumer processing logic? Is it CPU or I/O bound?"

## Troubleshooting Scenario 9: Permission Errors for Uploaded Files

\*\*Scenario\*\*: Users can't access uploaded files due to permission errors.

\*\*Debugging Steps\*\*:

- Check file system permissions (`ls -l /uploads`)

- Ensure the web server (Nginx) has read access

\*\*Analysis of Debugging Logic\*\*:

This is a straightforward permission verification approach. It checks the actual file permissions first, then verifies the service account permissions.

\*\*Clarifying Questions to Ask\*\*:

1. "Which user account does the web server run as?"

2. "What's the current ownership and permission mode of the upload directory?"

3. "Are we using any access control lists (ACLs) beyond basic permissions?"

4. "Has the server been recently updated or reconfigured?"

5. "Are all uploads affected or only certain file types/directories?"

6. "Are we using any chroot environments or container isolation?"

## Troubleshooting Scenario 10: Website Serving Stale Content

\*\*Scenario\*\*: A website serves stale content despite updated files.

\*\*Debugging Steps\*\*:

- Check `Cache-Control` headers (e.g., `max-age=86400`)

- Force CDN cache invalidation

\*\*Analysis of Debugging Logic\*\*:

This approach correctly identifies caching as the likely culprit, starting with HTTP header examination and then escalating to CDN-level cache management.

\*\*Clarifying Questions to Ask\*\*:

1. "Are we using a CDN? If so, which one and what's our caching configuration?"

2. "What are our current Cache-Control and ETag header values?"

3. "Have we implemented cache busting strategies (file versioning, query parameters)?"

4. "Is this affecting all users or only some geographic regions?"

5. "Are all resources stale or only specific file types?"

6. "When was the content last updated? Do our TTL settings exceed this timeframe?"

## Core Principles Across All Scenarios:

As a database and web expert, I notice these debugging approaches follow several key principles:

1. \*\*Layer-by-layer examination\*\*: Start at the application layer, then move to middleware, database, and infrastructure

2. \*\*Data-driven diagnosis\*\*: Collect metrics and logs before making changes

3. \*\*Configuration validation\*\*: Verify settings before assuming code issues

4. \*\*Performance tradeoff awareness\*\*: Recognize the inherent tradeoffs (like indexed vs. non-indexed, in-memory vs. disk-based)

Before starting any debugging process, always ask:

1. "When did the problem start? What changed around that time?"

2. "Do we have monitoring data showing the normal vs. current state?"

3. "Is this affecting all users/requests or just a subset?"

4. "What's the impact on the business? How urgently do we need a solution vs. a proper fix?"

5. "What resources (server access, logs, metrics) are available for debugging?"

Would you like me to dive deeper into any particular scenario or explain more about the underlying technologies involved in any of these troubleshooting approaches?