Distributed Database Systems: Interview Questions & Answers

Technical Interview Preparation Notes

July 19, 2025

Contents

1 Write-Heavy Database Optimization

Q1

What happens when we have a write-heavy database and master-slave replication isn't optimized? What are the alternatives?

Answer

Problems with Master-Slave in Write-Heavy Scenarios:

- **Single Write Bottleneck:** All writes funnel through one master node
- **Replication Lag:** Slaves fall behind under heavy write load
- **Master Overload:** CPU/IO saturation on the single write node
- **Scalability Limit:** Cannot horizontally scale writes

Better Solutions:

1. **Master-Master (Multi-Master):** Distribute writes across multiple nodes
2. **Database Sharding:** Partition data horizontally across multiple databases
3. **Write-Optimized Databases:** Use systems designed for high write throughput
4. **Queue-Based Writes:** Buffer writes through message queues

Key Insight: Master-slave is read-optimized, not write-optimized. For write-heavy workloads, distribute the write load across multiple nodes.

2 Consistency Challenges in Multi-Master Systems

Q2

Don't you think multi-master creates consistency problems between masters?

Answer

Yes, absolutely! Multi-master introduces significant consistency challenges:
The CAP Theorem Trade-off:

- Must choose between **Consistency** and **Availability** during network partitions
- Cannot achieve both strong consistency and high availability simultaneously

Specific Multi-Master Problems:

1. **Write Conflicts:** Same data updated simultaneously on different masters
2. **Split-Brain Scenarios:** Network partition creates conflicting states
3. **Eventual Consistency Delays:** Temporary inconsistencies during synchronization

Solutions Based on Requirements:
If Strong Consistency is Critical:

- Consensus protocols (Raft, Paxos) - reduces availability
- Two-phase commit - high latency, blocking
- Single leader per partition - hybrid approach

If Eventual Consistency is Acceptable:

- Conflict-free replicated data types (CRDTs)
- Last-write-wins with timestamps
- Application-level conflict resolution

3 Consensus Protocol Latency Trade-offs

Q3

Don't you think consensus protocols will add latency to the system?

Answer

Absolutely correct! Consensus protocols definitely add latency:
Why Consensus Adds Latency:

- **Multiple Round Trips:** Need majority agreement before commit
- **Network Communication Overhead:** Must wait for responses from multiple nodes
- **Blocking Operations:** Write operations wait until consensus is reached

Classic Trade-off:

$$\text{Strong Consistency} \Rightarrow \text{Higher Latency} \tag{1}$$
$$\text{Low Latency} \Rightarrow \text{Weaker Consistency Guarantees} \tag{2}$$

Real-world Approach - Hybrid Architecture:

1. **Analyze Use Case First:**
   - Financial transactions → Accept higher latency for consistency
   - Social media feeds → Prioritize low latency, eventual consistency
2. **Component-Based Architecture:**
   - Critical data (accounts, payments) → Consensus-based
   - Non-critical data (logs, analytics) → Eventual consistency
3. **Optimization Techniques:**
   - Read replicas for read-heavy operations
   - Async replication where possible
   - Batching writes to amortize consensus overhead

4 Real-World Example: E-commerce Platform

Q4

Can you provide a concrete example of how you would architect a write-heavy system?

Answer

E-commerce Platform with Write-Heavy Components:

```
Write Load Distribution:
- User activity logs: 10,000 writes/sec
- Inventory updates: 1,000 writes/sec
- Order processing: 500 writes/sec
- Payment transactions: 100 writes/sec
```

Architecture Strategy by Component:

1. Payment Transactions (Strong Consistency Required):

- Use consensus-based system (Raft/Paxos)
- Accept 50-100ms latency
- Justification: Financial data cannot be inconsistent

2. Inventory Updates (Semi-Critical):

- Sharded by `product_id`
- Single master per shard
- Async replication to read replicas
- Target: 10-20ms latency

3. User Activity Logs (Eventual Consistency OK):

- Multi-master with conflict-free writes
- Or message queue $\rightarrow$ batch processing
- Target: $<$ 5ms latency
- Justification: User behavior analytics can tolerate delays

4. Order Processing (Hybrid):

- Order creation: Strong consistency (payment involved)
- Order status updates: Eventual consistency

Key Architecture Principle: Different components have different consistency requirements - optimize each based on business impact, not one-size-fits-all.

5 Synchronization Complexity in Distributed Systems

Q5

When using consistent hashing with multiple replicas, doesn't synchronization between replicas lead to O(n^2) complexity?

Answer

Excellent observation! You're absolutely right about the O(n^2) problem:
The O(n^2) Problem:

$$\text{If every replica compares with every other replica:} \quad (3)$$
$$n \text{ replicas} \rightarrow \frac{n(n-1)}{2} \text{ comparisons} = O(n^2) \quad (4)$$

This becomes catastrophic at scale!
Efficient Synchronization Approaches:
1. Vector Clocks:

- Time Complexity: $O(n)$ per operation
- Each node maintains vector $[v_1, v_2, v_3, \ldots]$
- Compare only version vectors, not full datasets
- Sync only conflicting/newer records

2. Merkle Trees (Cassandra/DynamoDB approach):

- Time Complexity: $O(\log n)$ for comparison
- Tree-based checksums hierarchy
- Compare hashes level by level
- Sync only differing branches

3. Consistent Hashing with Anti-Entropy:

- Time Complexity: $O(k)$ where k = changed records
- Each node syncs only with immediate neighbors
- Background process handles drift correction

Production Example - Cassandra's Approach:

1. Gossip protocol - $O(\log n)$ nodes contacted
2. Merkle tree comparison - $O(\log \text{records})$
3. Delta sync - $O(\text{changed\_records})$

Optimal Solution: Use Merkle trees for efficient comparison ($O(\log n)$) and vector clocks for conflict resolution, reducing sync complexity to $O(\log n + k)$ where k is changed records.

6 Git-like Version Control in Databases

Q6

Is this synchronization approach similar to how Git VCS works?

Answer

Excellent analogy! Git is indeed a perfect model for distributed database synchronization:
Git's Approach = Distributed DB Sync:
1. Merkle Trees (Git's Core Architecture):

```
Git commit structure:
          Root hash (represents entire repository state)
          Tree objects (directory hashes)
          Blob objects (file content hashes)

Database equivalent:
          Root hash (shard state snapshot)
          Partition hashes (table/collection level)
          Record hashes (individual data items)
```

2. Version Tracking:

$$\text{Git:} \quad \text{commit\_hash} + \text{parent\_commits} \tag{5}$$

$$\text{Database:} \quad [node_1\_version, node_2\_version, node_3\_version] \tag{6}$$

3. Conflict Resolution Strategies:

- **Git:** Merge conflicts → Manual resolution
- **Database:** Write conflicts → Last-write-wins/CRDTs

4. Efficient Synchronization Process:

```
Git pull/push workflow:
1. Compare commit histories - O(log n)
2. Find divergence point
3. Transfer only delta changes

Database anti-entropy:
1. Compare Merkle tree hashes - O(log n)
2. Identify differing partitions
3. Sync only changed records
```

Key Insight: Git solved distributed consistency at the file system level; distributed databases solve it at the data level using identical principles: content-addressed storage, cryptographic hashing, and delta synchronization.
Real-world Implementation: Some databases (IPFS-based systems) literally adopt Git's model for data versioning.

7 IPFS as Database Storage Layer

Q7

What about IPFS as a database? Is it trending or more of an older, unused technology?

Answer

Clarification: IPFS isn't exactly a database - it's a distributed file system that can serve as a storage layer for databases.

Current Status (2025): "Promising but Niche"

IPFS Adoption Level:

- ✓ Active development (Protocol Labs)
- ✓ Growing developer community
- ✓ Some production use cases
- × Not mainstream enterprise adoption
- × Limited proven real-world scalability

Where IPFS is Actually Used:

1. **NFT Storage:** OpenSea, various NFT platforms
2. **Decentralized Applications:** DApps, Web3 ecosystem
3. **Academic/Research:** Data sharing and archival
4. **Blockchain Integration:** Filecoin storage network
5. **Content Archiving:** Internet Archive experiments

IPFS-Based Database Examples:

```
1. OrbitDB: Distributed database using IPFS
   const db = await orbitdb.log('my-log')
   await db.add('record1') // Stored as IPFS hash

2. Ceramic Protocol: Document-based database on IPFS
3. ThreadsDB: MongoDB-like API over IPFS
```

Why Not More Popular:

- **Performance Issues:** Slow content discovery, network latency
- **Query Limitations:** Complex queries difficult to implement
- **Operational Complexity:** Steep learning curve, infrastructure overhead
- **Mutable Data Challenges:** Designed for immutable content

Enterprise Reality: Most companies still prefer:

- Traditional databases (PostgreSQL, MySQL)
- Cloud storage (S3, Azure Blob)
- Proven distributed databases (Cassandra, MongoDB)

Interview Perspective: IPFS is interesting for specific use cases like immutable content storage and decentralized applications, but traditional databases remain more practical for typical enterprise applications due to performance, tooling maturity, and operational simplicity.

Verdict: Experimental/emerging technology rather than mainstream trending.

8 Additional Advanced Topics

8.1 Bonus Question: Database Partitioning Strategies

Q8

What are the different database partitioning strategies and their trade-offs?

Answer

Database Partitioning Strategies:
1. Horizontal Partitioning (Sharding):

- **Range-based:** Partition by value ranges (e.g., A-M, N-Z)
- **Hash-based:** Use hash function for even distribution
- **Directory-based:** Lookup service maps keys to partitions

2. Vertical Partitioning:

- Split tables by columns
- Separate frequently vs. rarely accessed data
- Useful for wide tables with different access patterns

Trade-offs Analysis:

$$\text{Benefits: } \text{Improved performance, scalability, parallel processing} \tag{7}$$
$$\text{Challenges: } \text{Cross-partition queries, rebalancing complexity} \tag{8}$$

8.2 Bonus Question: Event Sourcing vs. Traditional CRUD

Q9

When would you choose Event Sourcing over traditional CRUD operations in a distributed system?

Answer

Event Sourcing Advantages:

- **Audit Trail:** Complete history of all changes
- **Temporal Queries:** Query system state at any point in time
- **Conflict Resolution:** Events are naturally ordered and immutable
- **Debugging:** Replay events to reproduce issues

Use Cases for Event Sourcing:

1. Financial systems (transaction history critical)
2. Collaborative applications (Google Docs-style editing)
3. Analytics systems (need historical data analysis)
4. Microservices (event-driven architecture)

Trade-offs:

+ Strong consistency guarantees

+ Natural fit for distributed systems

− Storage overhead (all events stored)

− Query complexity (need to replay events)

− Learning curve for developers

9 Key Takeaways for Interviews

Interview Success Tips

1. **Always Ask About Requirements:** Consistency vs. Availability vs. Performance
2. **Discuss Trade-offs Explicitly:** No solution is perfect - show you understand compromises
3. **Use Concrete Examples:** Real-world scenarios demonstrate practical understanding
4. **Know the Complexity:** Time/space complexity analysis shows deep technical knowledge
5. **Connect Concepts:** Relating to known systems (Git, web architectures) shows broader understanding
6. **Business Context Matters:** Technical decisions should align with business requirements