Couchbase

Interview Questions and Answers

Core Concepts

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

1. Explain Couchbase's bucket architecture and the differences between Couchbase, Ephemeral, and Memcached bucket types.

Bucket Architecture

Couchbase organizes data into **buckets**, which are logical containers for documents. Each bucket has independent RAM quotas, replication settings, and access controls.

Bucket Types

- **Couchbase Buckets:** Full-featured buckets with persistence to disk, replication, and all indexing capabilities. Data is stored in memory and asynchronously persisted. Supports all query services (N1QL, Views, FTS).
- **Ephemeral Buckets:** Memory-only buckets without disk persistence. Faster writes since no disk I/O occurs. Data is lost on cluster restart. Ideal for session storage or caching with replication for availability. Supports ejection policies: NRU (Not Recently Used) or no ejection.
- **Memcached Buckets:** Legacy bucket type providing Memcached protocol compatibility. No persistence, replication, or indexing. Simple key-value operations only. Generally deprecated in favor of Ephemeral buckets.

Key Considerations

Couchbase buckets are preferred for production applications requiring durability. Ephemeral buckets offer **20-30% better write performance** when persistence isn't needed. Bucket selection impacts memory management, as ejection policies differ: Couchbase uses disk-backed ejection, while Ephemeral uses NRU or fails writes when memory is full.

2. How does Couchbase implement the vBucket mechanism for data distribution, and why is it superior to consistent hashing?

vBucket Architecture

Couchbase uses **vBuckets (virtual buckets)** as an indirection layer for data distribution. By default, each bucket has **1024 vBuckets** that are mapped to physical nodes. When a document is stored, its key is hashed to determine the vBucket number (0-1023), and a cluster map shows which node owns that vBucket.

Advantages Over Consistent Hashing

- **Rebalancing Efficiency:** When nodes are added/removed, only vBucket ownership changes—no rehashing of keys required. With consistent hashing, adding a node can require remapping a significant portion of the keyspace.
- **Controlled Data Movement:** Administrators can precisely control which vBuckets move during rebalance, enabling predictable migration patterns and minimal disruption.
- **Replica Management:** Each vBucket has explicit active and replica copies. The cluster map tracks exactly where each replica lives, simplifying failover.
- **Even Distribution:** 1024 vBuckets provide fine-grained distribution even with small clusters, avoiding hotspots common in naive consistent hashing implementations.

Practical Example

Key "user:12345" → Hash → vBucket 742
Cluster Map: vBucket 742 → Node 3 (active), Node 1 (replica)
Client directly contacts Node 3 for operations

This architecture enables **zero-downtime rebalancing** and automatic failover without client

reconfiguration.

3. Describe Couchbase's memory management and the difference between working set and data set. How do you size a cluster appropriately?

Memory Architecture

Couchbase is a **memory-first database** where all active data resides in RAM. Understanding memory management is critical for performance:

- **Data Set:** Total size of all documents and metadata across the cluster
- **Working Set:** Subset of data actively accessed by applications (typically 20-30% of data set in most applications)
- **Resident Ratio:** Percentage of working set in memory (target: >90% for optimal performance)

Memory Components

Each node's RAM quota is divided among:

- **Data Service:** Document storage and metadata
- **Index Service:** GSI index structures
- **Query Service:** Query execution buffers
- **Search Service:** Full-text index storage

Sizing Formula

RAM per node = (Working Set Size / # data nodes) × 1.3
+ Metadata overhead
+ Index memory
+ 20% headroom

Metadata ≈ (# documents × 56 bytes) / # nodes

Best Practices

- Monitor **ep_bg_fetched** metric—high values indicate insufficient RAM (data being read from disk)
- Keep resident ratio above 90% by ensuring working set fits in memory
- Use ephemeral buckets for pure caching scenarios to avoid disk I/O entirely
- Plan for growth: size for 18-24 months of data expansion

4. Explain the CAP theorem implications in Couchbase and how it achieves high availability during network partitions.

CAP Theorem Position

Couchbase is a **CP system** (Consistency and Partition Tolerance) by default, but offers tunable consistency for AP behavior. During network partitions, it prioritizes data consistency while maintaining availability through replica promotion.

Consistency Model

- **Strong Consistency:** All mutations are first committed to the active vBucket. Replicas are updated asynchronously but clients can request **ReplicaRead** or **ObserveSeqno** for consistency guarantees.
- **Read-Your-Own-Writes:** Guaranteed by default when reading from the same connection that performed the write
- **Eventual Consistency:** Replica vBuckets eventually sync with active vBuckets (typically <10ms in healthy clusters)

Partition Handling

When a network partition occurs:

1. Cluster manager detects node failure (default: 3-second timeout)
2. If auto-failover is enabled, replica vBuckets are promoted to active status
3. Clients receive updated cluster map and redirect requests to new active nodes
4. Write operations continue on the majority partition; minority partition rejects writes to prevent

split-brain

Tunable Durability

```
// Require persistence before acknowledging
insert(key, doc,
  {durability: DurabilityLevel.MAJORITY})

// Balance availability vs consistency
// NONE: async replication (AP)
// MAJORITY: wait for majority replicas (CP)
```

The **durability requirements** (introduced in Couchbase 6.5) allow developers to choose per-operation consistency levels, trading latency for stronger guarantees.

5. What are the different index types in Couchbase (GSI, View, FTS) and when would you use each?

Index Types Comparison

Couchbase provides three distinct indexing mechanisms, each optimized for different query patterns:

1. Global Secondary Indexes (GSI)

- **Use Case:** Standard N1QL queries with WHERE clauses, JOINs, and aggregations
- **Architecture:** Separate Index Service nodes using ForestDB or Plasma storage engines
- **Performance:** Low-latency lookups, optimized for OLTP workloads
- **Partitioning:** Supports partitioned indexes for horizontal scaling

```
CREATE INDEX idx_user_email
ON users(email)
WHERE type = 'active'
WITH {"num_replica": 1};
```

2. View Indexes (MapReduce)

- **Use Case:** Complex aggregations, pre-computed analytics, legacy applications
- **Architecture:** Distributed across Data Service nodes, incrementally updated
- **Performance:** Higher latency but excellent for aggregations and grouping
- **Status:** Deprecated in favor of GSI for most use cases

3. Full-Text Search (FTS)

- **Use Case:** Text search, fuzzy matching, linguistic analysis, geospatial queries
- **Architecture:** Separate Search Service using Bleve engine (similar to Elasticsearch)
- **Features:** Analyzers, faceting, highlighting, relevance scoring

```
// FTS query for fuzzy search
SELECT META().id, description
FROM products
WHERE SEARCH(description, "laptop",
  {"fuzziness": 2})
```

Selection Criteria

- **GSI:** Default choice for 90% of queries (equality, range, JOIN)
- **FTS:** When you need text search, typo tolerance, or geospatial
- **Views:** Only for legacy compatibility or specific MapReduce requirements

6. How does Cross Datacenter Replication (XDCR) work in Couchbase, and what are the conflict resolution strategies?

XDCR Architecture

Cross Datacenter Replication (XDCR) enables asynchronous replication between geographically distributed Couchbase clusters for disaster recovery, data locality, and active-active deployments.

Replication Mechanism

- **Topology:** Unidirectional or bidirectional replication between clusters
- **Granularity:** Bucket-level replication with optional filtering
- **Protocol:** Uses DCP (Database Change Protocol) to stream mutations
- **Network:** Compressed, encrypted replication streams over WAN
- **Performance:** Parallel replication pipelines per vBucket for high throughput

Conflict Resolution Strategies

In active-active scenarios, concurrent updates to the same document create conflicts. Couchbase provides two resolution methods:

1. Revision ID (Default)

- Each mutation increments a revision counter
- Conflict: document with **higher revision ID wins**
- Deterministic but may not preserve business logic

2. Timestamp-Based (LWW)

- Uses Last-Write-Wins based on **hybrid logical clock (HLC)**
- Combines physical timestamp with logical counter
- More intuitive but requires synchronized clocks (NTP)

```
// Configure XDCR with custom conflict resolution
xdcrSetup.conflictResolutionType = "lww";
xdcrSetup.filterExpression = "type = 'user'";
xdcrSetup.compressionType = "Snappy";
```

Best Practices

- Use **document design** to avoid conflicts (append-only, CRDTs)
- Implement **application-level versioning** for critical data
- Monitor **XDCR lag** and bandwidth utilization
- Enable **filtering** to replicate only necessary documents
- Use **priority replication** for critical buckets

7. Explain N1QL query optimization techniques and how the cost-based optimizer works in Couchbase.

Cost-Based Optimizer (CBO)

Introduced in Couchbase 7.0, the **cost-based optimizer** analyzes query structure, available indexes, and statistics to generate optimal execution plans. It replaces the earlier rule-based optimizer.

Optimization Process

1. **Query Parsing:** Parse N1QL into abstract syntax tree
2. **Statistics Gathering:** Analyze index and bucket statistics (cardinality, distribution)
3. **Plan Generation:** Generate multiple candidate execution plans
4. **Cost Estimation:** Estimate CPU, I/O, and network costs for each plan
5. **Plan Selection:** Choose lowest-cost plan

Key Optimization Techniques

- **Index Selection:** Choose covering indexes to avoid document fetches
- **Predicate Pushdown:** Apply WHERE filters as early as possible
- **Join Optimization:** Reorder joins based on selectivity
- **Unnest Optimization:** Efficiently process array operations

Practical Optimization Examples

```
-- Use covering index
CREATE INDEX idx_cover ON users(email, name);
SELECT email, name FROM users
```

```
WHERE email LIKE 'a%';

-- Explain to see plan
EXPLAIN SELECT ...;

-- Force index hint
SELECT /*+ INDEX(users idx_cover) */ ...
```

Performance Best Practices

- **Update Statistics:** Run UPDATE STATISTICS regularly for accurate cost estimation
- **Use Covering Indexes:** Include all queried fields to avoid document fetches
- **Partition Large Indexes:** Distribute index across nodes for parallel scans
- **Monitor Query Performance:** Use system:completed_requests to identify slow queries
- **Avoid SELECT *:** Request only needed fields to reduce network transfer
- **Leverage Prepared Statements:** Reuse query plans for repeated queries

8. What is the role of the Data Change Protocol (DCP) in Couchbase, and how is it used for building real-time applications?

Data Change Protocol (DCP)

DCP is Couchbase's internal streaming protocol that provides a persistent, ordered stream of all mutations (creates, updates, deletes) in a bucket. It's the foundation for replication, indexing, and real-time data pipelines.

DCP Characteristics

- **Ordering:** Guarantees per-vBucket sequential ordering of mutations
- **Reliability:** Supports resumption from specific sequence numbers (no data loss)
- **Completeness:** Streams all mutations including metadata and expiration events
- **Performance:** Low-latency streaming with backpressure handling
- **Filtering:** Can filter by collection, scope, or custom criteria

Internal Uses

- **XDCR:** Replicates data between clusters
- **Indexing:** GSI and FTS services consume DCP to update indexes
- **Views:** MapReduce indexes built from DCP streams
- **Backup:** cbbackupmgr uses DCP for incremental backups

External Use Cases

```
// Using Couchbase Kafka Connector (DCP-based)
{
  "name": "couchbase-source",
  "connector.class": "CouchbaseSourceConnector",
  "couchbase.bucket": "users",
  "couchbase.stream.from": "BEGINNING",
  "topic.name": "user-events"
}
```

Building Real-Time Applications

- **Change Data Capture (CDC):** Stream changes to Kafka, Kinesis, or custom consumers
- **Cache Invalidation:** Invalidate application caches when documents change
- **Event Sourcing:** Build event logs from mutation streams
- **Real-Time Analytics:** Feed changes to stream processing engines (Flink, Spark Streaming)
- **Audit Logging:** Track all data modifications for compliance

DCP Client Libraries

Available for Java, Go, and C. Provides low-level control for custom streaming applications with checkpoint management and failover handling.

9. How do you implement sub-document operations in Couchbase, and what are the performance benefits?

Sub-Document API

The **Sub-Document API** allows atomic operations on specific paths within JSON documents without retrieving or replacing the entire document. This provides significant performance and concurrency benefits.

Supported Operations

- **mutateIn:** Modify specific fields (upsert, insert, replace, remove, arrayAppend, arrayPrepend, arrayInsert, counter)
- **lookupIn:** Retrieve specific fields without fetching entire document
- **Atomic:** All operations are atomic at the document level
- **Efficient:** Only modified paths are transmitted and processed

Code Examples

```
// Update nested field atomically
collection.mutateIn("user:123",
  Arrays.asList(
    MutateInSpec.upsert("address.city", "NYC"),
    MutateInSpec.increment("loginCount", 1),
    MutateInSpec.arrayAppend("tags", "premium")
  ));

// Retrieve only needed fields
collection.lookupIn("user:123",
  Arrays.asList(
    LookupInSpec.get("email"),
    LookupInSpec.exists("preferences")
  ));
```

Performance Benefits

- **Reduced Network Transfer:** Only path and value transmitted (not entire document)
- **Lower CPU Usage:** Server only parses/modifies specific paths
- **Better Concurrency:** Reduces contention since full document lock not held as long
- **Optimistic Locking:** Can use CAS with sub-doc operations

Use Cases

- **Counters:** Increment view counts, likes without reading document
- **Arrays:** Add items to shopping carts, append log entries
- **Nested Updates:** Update deeply nested fields in complex documents
- **Partial Reads:** Fetch only required fields for API responses

Benchmark Results

Sub-document operations can be **3-5x faster** than full document operations for large documents (>100KB) and provide **2x higher throughput** under concurrent access patterns.

10. Describe Couchbase's transaction support (ACID) and the two-phase commit protocol implementation.

Distributed ACID Transactions

Couchbase introduced **multi-document ACID transactions** in version 6.5, supporting operations across multiple documents, collections, and scopes within a single cluster.

Transaction Properties

- **Atomicity:** All operations commit or rollback together
- **Consistency:** Transactions maintain data integrity constraints
- **Isolation:** Read Committed isolation level (configurable)
- **Durability:** Configurable persistence and replication requirements

Implementation Architecture

Couchbase uses an **optimistic concurrency control** approach with two-phase commit:

1. **Active Transaction Record (ATR):** Metadata document tracking transaction state
2. **Staging:** Writes are staged with transaction metadata
3. **Commit:** ATR updated to committed state
4. **Cleanup:** Staged mutations become visible, metadata removed

Code Example

```
cluster.transactions().run((ctx) -> {
  // Read documents
  TransactionGetResult user =
    ctx.get(collection, "user:123");
  TransactionGetResult account =
    ctx.get(collection, "account:456");

  // Modify and commit atomically
  ctx.replace(user,
    updatedUserDoc);
  ctx.replace(account,
    updatedAccountDoc);
});
```

Performance Considerations

- **Overhead:** Transactions add ~2-3ms latency per operation due to staging
- **Conflicts:** Optimistic locking means retries on concurrent modifications
- **Scope:** Keep transactions small (5-10 documents) for best performance
- **Durability:** Configure durability requirements per transaction

Best Practices

- Use transactions only when atomicity across documents is required
- Design documents to minimize cross-document transactions
- Implement retry logic for transaction conflicts
- Monitor transaction metrics (commit rate, rollback rate, conflicts)
- Avoid long-running transactions (timeout default: 15 seconds)

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. How would you implement an LRU Cache for Couchbase query results with O(1) operations?

LRU Cache Implementation

An **LRU (Least Recently Used) Cache** requires O(1) time complexity for both get and put operations. Use a combination of a **HashMap** and **Doubly Linked List**.

```
class LRUCache:
    def __init__(self, capacity):
        self.cache = {}
        self.capacity = capacity
        self.head = Node(0, 0)
        self.tail = Node(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head
```

- **HashMap**: Stores key-value pairs for O(1) lookup
- **Doubly Linked List**: Maintains access order, most recent at head
- **Get Operation**: Move accessed node to head
- **Put Operation**: Add to head, evict from tail if capacity exceeded

2. Explain how you would use a Trie data structure to optimize N1QL autocomplete queries in Couchbase.

Trie for Autocomplete Optimization

A **Trie (Prefix Tree)** is ideal for autocomplete as it provides efficient prefix-based searches with O(m) complexity where m is the length of the prefix.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False
        self.frequency = 0

class Trie:
    def insert(self, word):
        node = self.root
        for char in word:
            node = node.children.setdefault(char, TrieNode())
```

- **Space Efficiency**: Shares common prefixes among words
- **Search Time**: O(m) for prefix lookup
- **Integration**: Cache frequent search terms from Couchbase FTS
- **Ranking**: Store frequency counts at nodes for popularity-based suggestions

3. What is the time complexity of finding all pairs in a Couchbase document array that sum to a target value?

Two Sum Problem - Pair Finding

Finding pairs that sum to a target can be solved in **O(n)** time using a HashSet approach, which is optimal for processing Couchbase query results.

```
def find_pairs(arr, target):
```

```
    seen = set()
    pairs = []
    for num in arr:
        complement = target - num
        if complement in seen:
            pairs.append((complement, num))
        seen.add(num)
    return pairs
```

- **Time Complexity**: O(n) - single pass through array
- **Space Complexity**: O(n) - HashSet storage
- **Alternative**: Sorting approach is O(n log n)
- **Use Case**: Processing aggregated Couchbase query results efficiently

4. How would you implement a sliding window algorithm to analyze time-series data from Couchbase?

Sliding Window for Time-Series Analysis

The **sliding window technique** is essential for analyzing streaming or time-series data with O(n) complexity, perfect for Couchbase event data processing.

```
def max_sum_window(data, k):
    window_sum = sum(data[:k])
    max_sum = window_sum
    for i in range(k, len(data)):
        window_sum += data[i] - data[i-k]
        max_sum = max(max_sum, window_sum)
    return max_sum
```

- **Fixed Window**: Constant size, O(n) time
- **Dynamic Window**: Expands/contracts based on conditions
- **Applications**: Moving averages, anomaly detection, rate limiting
- **Couchbase Integration**: Process event streams from Eventing Service

5. Describe how to implement a Min Heap for priority-based document processing in Couchbase.

Min Heap Implementation

A **Min Heap** provides O(log n) insertion and O(1) minimum access, ideal for priority queues in document processing workflows.

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, priority, doc_id):
        heapq.heappush(self.heap, (priority, doc_id))

    def pop(self):
        return heapq.heappop(self.heap)[1]
```

- **Insert**: O(log n) - maintains heap property
- **Extract Min**: O(log n) - removes root
- **Peek Min**: O(1) - access root without removal
- **Use Cases**: Task scheduling, document replication prioritization, batch processing

6. What algorithm would you use to detect cycles in a Couchbase document reference graph?

Cycle Detection in Document Graphs

Depth-First Search (DFS) with a color-coding scheme is the most efficient approach for cycle detection in O(V + E) time.

```
def has_cycle(graph):
```

```
    WHITE, GRAY, BLACK = 0, 1, 2
    color = {node: WHITE for node in graph}

    def dfs(node):
        color[node] = GRAY
        for neighbor in graph.get(node, []):
            if color[neighbor] == GRAY: return True
            if color[neighbor] == WHITE and dfs(neighbor): return True
        color[node] = BLACK
        return False
```

- **WHITE**: Unvisited node
- **GRAY**: Currently in DFS stack (back edge indicates cycle)
- **BLACK**: Fully processed
- **Application**: Validate document references, prevent circular dependencies

7. How do you implement a consistent hashing algorithm for Couchbase vBucket distribution?

Consistent Hashing Implementation

Consistent hashing minimizes key redistribution when nodes are added/removed, essential for Couchbase's vBucket architecture.

```
import hashlib

class ConsistentHash:
    def __init__(self, nodes, replicas=3):
        self.ring = {}
        for node in nodes:
            for i in range(replicas):
                key = hashlib.md5(f"{node}:{i}".encode()).hexdigest()
                self.ring[int(key, 16)] = node
        self.sorted_keys = sorted(self.ring.keys())
```

- **Virtual Nodes**: Multiple hash positions per physical node for balance
- **Time Complexity**: O(log n) lookup using binary search
- **Minimal Disruption**: Only K/n keys move when adding/removing nodes
- **Couchbase**: Uses 1024 vBuckets distributed via consistent hashing

8. Explain the time complexity of merging K sorted Couchbase query results using a heap.

Merge K Sorted Lists

Merging K sorted query results efficiently uses a **Min Heap** approach with overall time complexity of **O(N log K)** where N is total elements.

```
import heapq

def merge_k_sorted(lists):
    heap = []
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(heap, (lst[0], i, 0))
    result = []
    while heap:
        val, list_idx, elem_idx = heapq.heappop(heap)
        result.append(val)
```

- **Time Complexity**: O(N log K) - N elements, K lists
- **Space Complexity**: O(K) - heap size
- **Heap Operations**: Each insert/extract is O(log K)
- **Use Case**: Merging results from multiple Couchbase nodes or indexes

9. How would you implement a Bloom Filter to optimize Couchbase document existence checks?

Bloom Filter for Existence Checks

A **Bloom Filter** is a space-efficient probabilistic data structure that provides O(1) membership testing with possible false positives but no false negatives.

```
import hashlib

class BloomFilter:
    def __init__(self, size, hash_count):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = [0] * size

    def add(self, item):
        for i in range(self.hash_count):
            idx = self._hash(item, i) % self.size
            self.bit_array[idx] = 1
```

- **Space Efficiency**: Uses bits instead of storing actual keys
- **Time Complexity**: O(k) where k is number of hash functions
- **False Positive Rate**: Configurable via size and hash count
- **Couchbase Use**: Cache layer to avoid unnecessary database lookups

10. What is the optimal algorithm for finding the Kth largest document by score in Couchbase query results?

Kth Largest Element - QuickSelect

The **QuickSelect algorithm** provides average O(n) time complexity for finding the Kth largest element, more efficient than full sorting for large datasets.

```
import random

def quickselect(arr, k):
    if len(arr) == 1: return arr[0]
    pivot = random.choice(arr)
    lows = [x for x in arr if x < pivot]
    highs = [x for x in arr if x > pivot]
    pivots = [x for x in arr if x == pivot]
    if k < len(highs): return quickselect(highs, k)
    elif k < len(highs) + len(pivots): return pivot
    else: return quickselect(lows, k - len(highs) - len(pivots))
```

- **Average Time**: O(n) - better than O(n log n) sorting
- **Worst Case**: $O(n^2)$ - mitigated by random pivot selection
- **Space**: O(1) in-place version available
- **Alternative**: Min Heap of size K is O(n log k), better for streaming data

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service using Couchbase. How would you handle high read/write throughput and ensure low latency?

Architecture Overview

A URL shortener with Couchbase requires careful consideration of data modeling, partitioning, and caching strategies.

Key Components

- **Data Model:** Store mappings with short code as document key for O(1) lookups
- **Counter Service:** Use Couchbase atomic counters for generating unique IDs
- **Bucket Design:** Separate buckets for URLs (high read) and analytics (high write)
- **Indexes:** Create secondary index on original URL for reverse lookups

Sample Document Structure

```
{
  "type": "url_mapping",
  "shortCode": "abc123",
  "originalUrl": "https://example.com/very/long/url",
  "createdAt": 1234567890,
  "expiresAt": 1234567890,
  "clicks": 0
}
```

Scalability Strategies

- **Horizontal Scaling:** Couchbase automatically shards data across nodes using vBuckets
- **Read Replicas:** Configure replica count to 2-3 for high availability
- **Caching Layer:** Use built-in memory-first architecture; hot keys stay in RAM
- **Write Performance:** Async click tracking using N1QL UPDATE with CAS for optimistic locking

Handling Collisions

Use base62 encoding with counter-based ID generation to avoid collisions. Implement retry logic for rare edge cases.

2. How would you design a real-time social media feed system with Couchbase that supports millions of users with different follower counts?

System Architecture

Design must handle the **fan-out problem** where celebrity users have millions of followers while maintaining real-time delivery.

Hybrid Fan-Out Approach

- **Fan-out on Write:** For users with <1000 followers, write to each follower's feed
- **Fan-out on Read:** For celebrity users (>100k followers), fetch posts on demand
- **Hybrid:** Users between 1k-100k get partial fan-out with caching

Data Model

```
// User Timeline Document
```

```
{
  "type": "timeline",
  "userId": "user123",
  "posts": [
    {"postId": "p1", "authorId": "a1", "ts": 1234567890},
    {"postId": "p2", "authorId": "a2", "ts": 1234567880}
  ]
}
```

Couchbase Features Used

- **Sub-document API:** Append to timeline arrays without fetching entire document
- **N1QL Queries:** For fan-out on read, query posts from followed users with time-based filtering
- **Cross Data Center Replication (XDCR):** Geo-distributed deployment for global users
- **Eventing Service:** Async processing of post creation to trigger fan-out workers

Optimization Techniques

- Maintain separate hot/cold storage with TTL-based archival
- Use materialized views for trending content
- Implement pagination with range scans on timestamp fields

3. Design a distributed session management system using Couchbase. How would you handle session consistency, failover, and security?

Session Storage Design

Couchbase is ideal for session management due to its **low latency**, **automatic failover**, and **TTL support**.

Document Structure

```
{
  "type": "session",
  "sessionId": "sess_abc123",
  "userId": "user456",
  "data": {"cart": [], "prefs": {}},
  "createdAt": 1234567890,
  "lastAccessed": 1234567900,
  "ipAddress": "192.168.1.1"
}
```

Key Implementation Details

- **Document Key:** Use sessionId as key for O(1) access
- **TTL Management:** Set document expiration (e.g., 30 minutes) with touch-on-access pattern
- **Atomic Updates:** Use CAS (Compare-And-Swap) to prevent race conditions
- **Security:** Encrypt sensitive session data before storing; use TLS for transport

High Availability Strategy

- **Replication:** Configure 1-2 replicas for automatic failover
- **Consistency:** Use strong consistency reads when critical (e.g., payment flow)
- **Multi-Region:** XDCR for disaster recovery with conflict resolution policies

Session Invalidation

Implement logout via immediate document deletion. For security events (password change), use a blacklist bucket with shorter TTL checked on each request.

Scalability Considerations

Sessions distribute automatically across cluster nodes. For very high throughput, use connection pooling and async operations in SDK.

4. How would you design a real-time analytics dashboard using Couchbase that aggregates metrics from millions of events per second?

Architecture Components

Real-time analytics requires a **lambda architecture** combining batch and stream processing with Couchbase as the serving layer.

Data Pipeline

- **Ingestion:** Events written to Couchbase with high write throughput
- **Stream Processing:** Couchbase Eventing Service for real-time aggregation
- **Batch Processing:** Scheduled N1QL queries for historical rollups
- **Serving Layer:** Pre-aggregated metrics stored in separate bucket

Event Document

```
{
  "type": "event",
  "eventId": "evt_123",
  "userId": "u456",
  "action": "click",
  "timestamp": 1234567890,
  "metadata": {"page": "/home"}
}
```

Aggregation Strategy

- **Time-Window Buckets:** Store metrics by minute/hour/day as separate documents
- **Counter Documents:** Use atomic increment operations for simple counts
- **Eventing Functions:** Trigger on event write to update real-time counters
- **Materialized Views:** Pre-compute complex aggregations using N1QL

Query Optimization

- Create covering indexes on timestamp and userId fields
- Use index partitioning for time-series data
- Implement result caching for frequently accessed dashboards

Scalability Patterns

Partition events by time ranges across multiple buckets. Use XDCR for analytics cluster separation from production workload.

5. Design a distributed caching layer using Couchbase for an e-commerce platform. How would you handle cache invalidation, consistency, and the thundering herd problem?

Caching Strategy

Couchbase serves as both **primary cache** and **persistent store**, eliminating traditional cache-aside complexity.

Cache Hierarchy

- **L1 Cache:** Couchbase in-memory tier (working set in RAM)
- **L2 Cache:** Couchbase disk persistence for cold data
- **Source of Truth:** Relational DB or other systems synced via CDC

Product Cache Document

```
{
  "type": "product",
  "productId": "prod_123",
  "name": "Widget",
  "price": 29.99,
  "inventory": 100,
  "version": 5,
  "cachedAt": 1234567890
}
```

Cache Invalidation Strategies

- **TTL-Based:** Set expiration on documents (e.g., 1 hour for product data)
- **Event-Driven:** Use Kafka/CDC to push updates to Couchbase on source changes
- **Versioning:** Include version field; clients check staleness
- **Active Expiration:** Eventing functions to proactively refresh before expiry

Thundering Herd Mitigation

- **Probabilistic Early Expiration:** Refresh cache before TTL with jitter
- **Locking Pattern:** Use document with short TTL as distributed lock during refresh
- **Request Coalescing:** Application-level deduplication of concurrent refresh requests

Consistency Models

Use **eventual consistency** for reads with **strong consistency** for inventory updates via CAS operations to prevent overselling.

6. How would you architect a multi-tenant SaaS application using Couchbase? Discuss data isolation, performance isolation, and scaling strategies.

Multi-Tenancy Approaches

Choose between **shared**, **siloed**, or **hybrid** models based on tenant size and isolation requirements.

Data Isolation Strategies

- **Bucket-per-Tenant:** Complete isolation but limited scalability (max ~30 buckets)
- **Scope-per-Tenant:** Logical isolation within bucket (up to 1000 scopes)
- **Collection-per-Tenant:** Fine-grained isolation (up to 1000 collections per scope)
- **Document-level:** Shared collections with tenantId field and filtered queries

Recommended Hybrid Model

```
// Large tenants: dedicated scope
// Small tenants: shared collection
{
  "type": "user",
  "tenantId": "tenant_123",
  "userId": "user_456",
  "email": "user@example.com",
  "plan": "enterprise"
}
```

Performance Isolation

- **Resource Quotas:** Set memory/disk limits per bucket or scope
- **Query Limits:** Implement rate limiting using N1QL query timeouts
- **Separate Clusters:** VIP tenants on dedicated clusters with XDCR sync
- **Index Strategy:** Create filtered indexes per tenant for large customers

Query Patterns

Always include tenantId in WHERE clause. Use prepared statements with parameterized tenantId to leverage query plan caching.

Scaling Strategies

- Start with document-level multi-tenancy for small tenants
- Graduate large tenants to dedicated scopes/collections
- Use analytics service for cross-tenant reporting without impacting OLTP

7. Design a distributed job queue system using Couchbase. How would you ensure exactly-once processing, handle failures, and prioritize jobs?

Job Queue Architecture

Leverage Couchbase's **atomic operations** and **TTL features** to build a reliable distributed queue without external dependencies.

Job Document Structure

```
{
  "type": "job",
  "jobId": "job_123",
  "status": "pending",
  "priority": 1,
  "payload": {"task": "send_email"},
  "attempts": 0,
  "maxAttempts": 3,
  "createdAt": 1234567890,
  "lockedBy": null,
  "lockedUntil": null
}
```

Exactly-Once Processing

- **Optimistic Locking:** Use CAS (Compare-And-Swap) when claiming jobs
- **Lease-Based Locking:** Set lockedUntil timestamp; workers must renew or release
- **Idempotency Keys:** Include unique jobId in payload for idempotent operations
- **Status Transitions:** Atomic updates: pending → processing → completed/failed

Worker Claim Pattern

Workers query for pending jobs with N1QL, then attempt atomic update with CAS. Failed CAS means another worker claimed it.

Failure Handling

- **Visibility Timeout:** Jobs with expired locks become visible again
- **Retry Logic:** Increment attempts counter; move to DLQ after maxAttempts
- **Dead Letter Queue:** Separate collection for failed jobs requiring manual intervention

Priority Queue Implementation

Use N1QL query ordered by priority and createdAt. Create composite index on (status, priority, createdAt) for efficient polling.

Scalability

Multiple workers poll independently. Partition jobs by type into separate collections for parallel processing.

8. How would you design a content delivery and personalization system using Couchbase that serves personalized content to millions of users with sub-100ms latency?

System Design Overview

Combine Couchbase's **memory-first architecture** with **Full Text Search** and **machine learning features** for real-time personalization.

Data Models

```
// User Profile
{
  "type": "profile",
  "userId": "u123",
  "preferences": ["tech", "sports"],
  "segments": ["premium", "engaged"],
  "history": ["c1", "c2", "c3"]
}
// Content Item
{
  "type": "content",
```

```
  "contentId": "c456",
  "tags": ["tech", "ai"],
  "score": 0.95
}
```

Architecture Components

- **Profile Service:** User profiles cached in Couchbase with sub-document API for fast updates
- **Content Catalog:** All content items indexed by tags, categories, and metadata
- **Recommendation Engine:** Pre-computed recommendations stored per user segment
- **Real-Time Scoring:** Eventing functions to update content scores based on engagement

Personalization Strategy

- **Segment-Based:** Pre-compute recommendations for user segments (e.g., premium users)
- **Individual:** Store top-N personalized items per user for instant retrieval
- **Hybrid:** Combine segment recommendations with user-specific boosting

Query Optimization

- Use Full Text Search for content discovery with user preference boosting
- Create covering indexes on frequently queried fields
- Implement result caching with 5-10 minute TTL

Latency Optimization

Keep hot user profiles and popular content in memory. Use SDK connection pooling and async operations. Deploy geo-distributed clusters with XDCR.

9. Design a fraud detection system using Couchbase that processes transactions in real-time and maintains a dynamic risk scoring model. How would you handle high throughput and evolving fraud patterns?

Real-Time Fraud Detection Architecture

Build a **streaming analytics pipeline** with Couchbase as the operational data store for risk profiles and transaction history.

Transaction Document

```
{
  "type": "transaction",
  "txnId": "txn_789",
  "userId": "u123",
  "amount": 500.00,
  "timestamp": 1234567890,
  "riskScore": 0.75,
  "features": {"velocity": 3, "location": "US"},
  "status": "flagged"
}
```

Risk Scoring Components

- **User Risk Profile:** Document per user with historical patterns and current risk level
- **Feature Store:** Real-time features like transaction velocity, location changes, device fingerprints
- **Rule Engine:** Eventing functions to evaluate rules on each transaction
- **ML Model Scores:** Pre-computed model scores updated periodically

Real-Time Processing

- **Eventing Service:** Trigger on transaction write to compute risk score immediately
- **Sub-document API:** Update user profile features without full document read/write
- **Atomic Counters:** Track transaction velocity per user/card in rolling time windows
- **N1QL Queries:** Analyze patterns across recent transactions for anomaly detection

Handling Evolving Patterns

- Store feature vectors for offline model retraining
- Use Analytics Service for batch analysis of fraud trends
- Implement A/B testing framework for rule changes
- Maintain audit trail with document versioning

High Throughput Design

Partition transactions by time ranges. Use async writes with durability levels based on risk. Scale cluster horizontally as volume grows.

10. How would you design a distributed configuration management system using Couchbase that supports versioning, rollback, and real-time updates across thousands of microservices?

Configuration Management Architecture

Build a **centralized config store** with Couchbase providing high availability, versioning, and push-based updates to services.

Configuration Document Model

```
{
  "type": "config",
  "service": "payment-service",
  "environment": "production",
  "version": 5,
  "config": {"timeout": 30, "retries": 3},
  "createdAt": 1234567890,
  "createdBy": "admin@example.com",
  "active": true
}
```

Key Features

- **Versioning:** Store each config change as new document with incremented version
- **Active Config:** Use active flag or separate current pointer document
- **History:** Query all versions using N1QL for audit trail
- **Rollback:** Set older version as active via atomic update

Real-Time Update Distribution

- **SDK Watch API:** Services subscribe to document changes for push notifications
- **Eventing Functions:** Trigger webhooks or pub/sub messages on config updates
- **Polling Fallback:** Services poll with If-Modified-Since pattern for reliability
- **Cache Invalidation:** Include version in ETag for HTTP cache busting

Multi-Environment Strategy

Use collections per environment (dev/staging/prod) or include environment in document key. Implement promotion workflow with approval gates.

Advanced Features

- Feature flags with percentage rollouts stored as config
- A/B test configurations with user segment targeting
- Schema validation using N1QL CHECK constraints
- Encryption for sensitive config values

High Availability

Configure 2-3 replicas for config bucket. Use XDCR for multi-region deployments. Implement circuit breakers in services for Couchbase unavailability.

Coding and Debugging

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

1. How do you perform a basic UPSERT operation in Couchbase using the Node.js SDK?

Basic UPSERT Operation

An **upsert** operation inserts a document if it doesn't exist or updates it if it does. Here's how to implement it:

```
const couchbase = require('couchbase');
const cluster = await couchbase.connect('couchbase://localhost', {
  username: 'Administrator',
  password: 'password'
});
const bucket = cluster.bucket('travel-sample');
const collection = bucket.defaultCollection();

await collection.upsert('user::001', {
  name: 'John Doe',
  email: 'john@example.com'
});
```

Key Points:

- Upsert is atomic and thread-safe
- Returns a MutationResult with CAS value
- Use for both insert and update scenarios

2. Write a function to perform a N1QL query with parameterized inputs and handle pagination in Couchbase.

Parameterized N1QL Query with Pagination

Parameterized queries prevent **injection attacks** and enable query plan caching:

```
async function getUsersByCity(city, limit = 10, offset = 0) {
  const query = `SELECT name, email FROM \`users\`
    WHERE city = $1 LIMIT $2 OFFSET $3`;

  const result = await cluster.query(query, {
    parameters: [city, limit, offset]
  });

  return result.rows;
}
```

Best Practices:

- Always use positional ($1, $2) or named parameters
- Implement appropriate indexes for WHERE clauses
- Use LIMIT/OFFSET for pagination
- Consider using meta().id for cursor-based pagination

3. How would you debug a CAS mismatch error in Couchbase? Provide a code example with retry logic.

Handling CAS Mismatch with Retry Logic

A **CAS (Compare-And-Swap)** mismatch occurs during concurrent modifications. Implement exponential backoff retry:

```
async function updateWithRetry(key, updateFn, maxRetries = 3) {
  for (let i = 0; i < maxRetries; i++) {
    try {
      const result = await collection.get(key);
      const updated = updateFn(result.content);
      await collection.replace(key, updated, { cas: result.cas });
      return updated;
    } catch (err) {
      if (err.name === 'CasMismatchError' && i < maxRetries - 1) {
        await new Promise(r => setTimeout(r, 100 * Math.pow(2, i)));
      } else throw err;
    }
  }
}
```

Debugging Tips:

- Log CAS values to track concurrent modifications
- Use Couchbase SDK logging to trace operations
- Monitor for high contention on specific documents

4. Implement a function to perform a bulk get operation efficiently in Couchbase.

Efficient Bulk Get Operation

For retrieving multiple documents, use the **getMulti** or Promise.all approach:

```
async function bulkGet(keys) {
  const promises = keys.map(key =>
    collection.get(key)
      .then(result => ({ key, content: result.content, success: true }))
      .catch(err => ({ key, error: err.message, success: false }))
  );

  return await Promise.all(promises);
}
```

Performance Considerations:

- Promise.all executes requests in parallel
- Handle individual failures gracefully
- Consider batching for very large key sets (1000+ keys)
- Use connection pooling for optimal throughput

5. How do you implement a subdocument mutation to update a nested field without retrieving the entire document?

Subdocument Mutation

Subdocument operations allow efficient updates to nested fields without full document retrieval:

```
async function updateUserAddress(userId, newAddress) {
  await collection.mutateIn(userId, [
    couchbase.MutateInSpec.upsert('address', newAddress),
    couchbase.MutateInSpec.increment('version', 1)
  ]);
}

// Usage
await updateUserAddress('user::001', {
  street: '123 Main St',
  city: 'Boston'
});
```

Advantages:

- Reduced network overhead (only sends changed fields)
- Atomic operations on nested paths
- Better performance for large documents

- Supports array operations (arrayAppend, arrayPrepend)

6. Write a function to handle connection pooling and graceful shutdown in a Couchbase application.

Connection Management

Proper **connection lifecycle management** prevents resource leaks:

```
class CouchbaseManager {
  constructor() {
    this.cluster = null;
  }

  async connect(connStr, username, password) {
    this.cluster = await couchbase.connect(connStr, {
      username, password,
      timeouts: { kvTimeout: 5000 }
    });
    return this.cluster;
  }

  async shutdown() {
    if (this.cluster) await this.cluster.close();
  }
}
```

Best Practices:

- Reuse cluster instances across the application
- Set appropriate timeout values
- Implement graceful shutdown on SIGTERM/SIGINT
- Use health checks to monitor connection status

7. How would you debug slow N1QL queries? Provide a code example to extract and analyze query execution plans.

Query Performance Debugging

Use **EXPLAIN** to analyze query execution plans and identify bottlenecks:

```
async function analyzeQuery(queryStr) {
  const explainQuery = `EXPLAIN ${queryStr}`;
  const result = await cluster.query(explainQuery);

  const plan = result.rows[0];
  console.log('Operator:', plan.operator);
  console.log('Index Used:', plan['~children']?.[0]?.index);

  return plan;
}
```

Debugging Checklist:

- Check if appropriate indexes exist using EXPLAIN
- Look for full bucket scans (IntersectScan vs PrimaryScan)
- Enable query logging with metrics: true option
- Monitor query execution time and document scanned count
- Use covering indexes to avoid document fetches

8. Implement error handling for common Couchbase exceptions including timeout, document not found, and network errors.

Comprehensive Error Handling

Proper **exception handling** improves application resilience:

```
async function safeGet(key) {
  try {
```

```
    const result = await collection.get(key);
    return { success: true, data: result.content };
  } catch (err) {
    if (err.name === 'DocumentNotFoundError') {
      return { success: false, error: 'NOT_FOUND' };
    } else if (err.name === 'TimeoutError') {
      return { success: false, error: 'TIMEOUT', retry: true };
    } else if (err.name === 'NetworkError') {
      return { success: false, error: 'NETWORK', retry: true };
    }
    throw err;
  }
}
```

Error Categories:

- DocumentNotFoundError: Document doesn't exist
- TimeoutError: Operation exceeded timeout threshold
- CasMismatchError: Concurrent modification conflict
- NetworkError: Connection or network issues

9. How do you implement a full-text search query with facets and highlighting in Couchbase?

Full-Text Search with Facets

FTS (Full-Text Search) provides advanced search capabilities:

```
async function searchProducts(term, category) {
  const result = await cluster.searchQuery('products-index',
    couchbase.SearchQuery.match(term), {
      fields: ['name', 'description'],
      facets: {
        category: couchbase.SearchFacet.term('category', 5)
      },
      highlight: { style: 'html', fields: ['name'] },
      limit: 20
    });

  return result;
}
```

FTS Features:

- Facets enable aggregation and filtering
- Highlighting shows matching terms in context
- Supports fuzzy matching and wildcards
- Configure analyzers for language-specific processing

10. Write a function to implement optimistic locking with version fields for conflict resolution in Couchbase.

Optimistic Locking Pattern

Version-based locking provides application-level conflict detection:

```
async function updateWithVersion(key, updateFn) {
  const doc = await collection.get(key);
  const currentVersion = doc.content.version || 0;

  const updated = updateFn(doc.content);
  updated.version = currentVersion + 1;

  await collection.replace(key, updated, { cas: doc.cas });
  return updated;
}

// Usage with conflict handling
await updateWithVersion('order::123', (order) => {
```

```
  order.status = 'shipped';
  return order;
});
```

Implementation Notes:

- Combine version field with CAS for dual protection
- Increment version on each successful update
- Handle conflicts at application level
- Useful for distributed systems and eventual consistency scenarios

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 had to optimize a slow-performing Couchbase query in production.

Situation: In my previous role, our e-commerce application experienced severe performance degradation during peak hours, with product search queries taking over 5 seconds to complete. The queries were hitting a Couchbase bucket with 50 million documents.

Task: I was tasked with reducing query response times to under 500ms while maintaining accuracy and not disrupting the production environment.

Action: I analyzed the N1QL queries using the Query Workbench and identified missing indexes. I created covering indexes that included all fields referenced in the WHERE and SELECT clauses. I also implemented query result caching for frequently accessed data and optimized the document structure to reduce JOIN operations by denormalizing related data.

Result: Query response times dropped from 5+ seconds to an average of 300ms, representing a 94% improvement. This resulted in a 25% increase in conversion rates during peak traffic periods and significantly improved customer satisfaction scores.

2. Describe a situation where you had to handle a Couchbase cluster failure or data loss scenario.

Situation: During a routine data center maintenance window, we experienced an unexpected power failure that caused two nodes in our three-node Couchbase cluster to go offline simultaneously, risking data loss and service disruption for our real-time analytics platform.

Task: I needed to restore cluster functionality, ensure data integrity, and implement measures to prevent similar incidents while minimizing downtime for our 24/7 service.

Action: I immediately initiated our disaster recovery protocol. I verified that the remaining node maintained quorum and began bringing nodes back online one at a time, allowing full replication to complete before adding the next node. I validated data consistency using document counts and checksum verification. I then worked with infrastructure to implement UPS backup systems and configured Cross Data Center Replication (XDCR) to a secondary cluster for high availability.

Result: We restored full cluster functionality within 45 minutes with zero data loss. The XDCR implementation reduced our RTO from 45 minutes to under 5 minutes for future incidents, and we achieved 99.99% uptime over the following year.

3. Give an example of how you mentored a junior developer on Couchbase best practices.

Situation: A junior developer on my team was struggling with designing an efficient data model for a new microservice that would store user session data in Couchbase. They were creating excessive documents and performing inefficient queries that would not scale.

Task: I needed to mentor them on Couchbase data modeling principles, document design, and query optimization while ensuring the project stayed on schedule.

Action: I scheduled pair programming sessions where we reviewed their current implementation together. I explained key concepts like document-oriented thinking versus relational thinking, the importance of document keys for direct lookups, and when to use embedded documents versus separate documents. We refactored their code together, implementing a more efficient key structure and using sub-documents for nested data. I also created internal documentation with examples and code snippets for the team to reference.

Result: The junior developer successfully delivered a well-architected solution that handled 10,000 requests per second with sub-millisecond latency. They became our team's go-to person for Couchbase questions and later presented their learnings at our engineering all-hands meeting,

helping other teams avoid similar pitfalls.

4. Tell me about a time when you had to migrate data from a relational database to Couchbase.

Situation: Our company decided to migrate a legacy order management system from PostgreSQL to Couchbase to improve scalability and reduce query latency. The PostgreSQL database contained 15 tables with complex relationships and 200GB of historical order data.

Task: I was the technical lead responsible for designing the migration strategy, transforming the relational schema into a document-oriented model, and executing the migration with zero data loss and minimal downtime.

Action: I analyzed the existing schema and access patterns to design an optimal document structure, denormalizing related data where appropriate. I created a migration pipeline using Apache Kafka to stream changes in real-time. I implemented dual-write logic to write to both databases during the transition period and built validation scripts to ensure data consistency. I conducted load testing on the new Couchbase implementation and created rollback procedures.

Result: We successfully migrated all data over a three-week period with zero downtime. The new system reduced average query response times from 800ms to 50ms and could handle 5x more concurrent requests. The improved performance enabled new real-time features that increased order processing efficiency by 40%.

5. Describe a challenging situation where you had to debug a complex issue in a distributed Couchbase environment.

Situation: Our production Couchbase cluster serving a mobile application started experiencing intermittent timeout errors affecting approximately 5% of requests. The errors were random, making them difficult to reproduce, and traditional monitoring showed no obvious resource constraints or network issues.

Task: I needed to identify the root cause of these intermittent timeouts and resolve them before they impacted our SLA agreements and user experience.

Action: I enabled detailed logging and collected metrics across all cluster nodes. I analyzed slow query logs and discovered that certain N1QL queries were occasionally taking 10x longer than normal. Using the Query Monitoring API, I identified that these queries coincided with index building operations. I discovered that our auto-compaction settings were triggering during peak hours. I adjusted the compaction schedule to off-peak hours, implemented query timeout settings with appropriate retry logic in the application, and set up better alerting for index build operations.

Result: The timeout errors dropped from 5% to less than 0.1%, well within our SLA targets. The improved monitoring and alerting helped us proactively identify and resolve three other potential issues before they impacted users. This experience led me to create a troubleshooting runbook that reduced our mean time to resolution by 60%.

6. Share an experience where you had to make a critical architectural decision involving Couchbase.

Situation: While designing a new real-time notification system expected to handle 100,000 concurrent users, I had to choose between using Couchbase with its built-in full-text search capabilities or integrating a separate Elasticsearch cluster alongside Couchbase for search functionality.

Task: I needed to evaluate both options considering performance, operational complexity, cost, and long-term maintainability, then present a recommendation to the architecture review board.

Action: I conducted a thorough analysis by creating proof-of-concept implementations for both approaches. I benchmarked query performance, analyzed resource requirements, and calculated total cost of ownership. I evaluated the maturity of Couchbase FTS versus Elasticsearch for our specific use cases. I documented trade-offs including operational overhead of maintaining two systems versus feature limitations of Couchbase FTS. I presented my findings with concrete metrics and recommended using Couchbase FTS for our use case to reduce complexity.

Result: The architecture board approved the Couchbase-only approach. This decision reduced our infrastructure costs by 30% and simplified our deployment pipeline. The system successfully launched handling 150,000 concurrent users with search query latencies under 100ms, exceeding our performance targets while requiring fewer engineering resources to maintain.

7. Tell me about a time when you had to balance technical debt with new feature development in a Couchbase-based system.

Situation: Our team was under pressure to deliver new features for a customer-facing application built on Couchbase, but we had accumulated significant technical debt including outdated SDK versions, inefficient document structures, and missing monitoring that occasionally caused production issues.

Task: I needed to advocate for addressing technical debt while maintaining feature delivery velocity and demonstrate the business value of the refactoring work.

Action: I quantified the impact of technical debt by tracking incident frequency, mean time to recovery, and developer velocity metrics. I created a proposal showing how technical debt was costing us approximately 30% of our development capacity. I negotiated with product management to allocate 20% of each sprint to technical improvements. I prioritized debt items by risk and impact, starting with upgrading the Couchbase SDK to gain performance improvements and bug fixes, then refactoring our most problematic document structures. I ensured each improvement was measurable and communicated wins to stakeholders.

Result: Over six months, we reduced production incidents by 65% and improved deployment frequency by 40%. The SDK upgrade alone improved query performance by 20%. The improved stability and developer productivity actually enabled us to deliver features faster, and product management became advocates for continued technical investment.

8. Describe a situation where you had to implement security best practices for Couchbase in a compliance-driven environment.

Situation: Our healthcare application needed to achieve HIPAA compliance, which required implementing comprehensive security controls for our Couchbase cluster storing protected health information (PHI). Our existing implementation lacked encryption, proper access controls, and audit logging.

Task: I was responsible for implementing security enhancements to meet HIPAA requirements while maintaining system performance and availability, with a deadline tied to a major client contract.

Action: I conducted a security audit of our current implementation and created a remediation plan. I implemented encryption at rest using Couchbase's built-in encryption features and encryption in transit using TLS for all client and node-to-node communication. I configured role-based access control (RBAC) with least-privilege principles, creating specific roles for different application services and administrative functions. I enabled audit logging to track all data access and administrative actions. I implemented field-level encryption for the most sensitive PHI fields using the Couchbase Field Level Encryption library. I documented all procedures and trained the team on security protocols.

Result: We successfully passed the HIPAA compliance audit on the first attempt with zero critical findings. The security implementation added less than 5ms to average query latency, well within acceptable limits. We secured the major client contract worth $2M annually, and the security framework became a template for three other projects in the organization.

9. Give an example of how you handled a disagreement with a team member about Couchbase implementation approach.

Situation: During the design phase of a new feature, a senior developer on my team insisted on using N1QL queries with complex JOINs for retrieving user profile data, while I advocated for a denormalized document approach with embedded data to optimize for read performance.

Task: I needed to resolve this technical disagreement constructively while maintaining team cohesion and ensuring we chose the best approach for our use case.

Action: I proposed that we both create proof-of-concept implementations and benchmark them against our expected load patterns. I scheduled a technical discussion session where we each presented our approaches, including code examples, performance metrics, and trade-offs. I remained open to their perspective and asked questions to understand their reasoning. During testing, my approach showed 3x better read performance, but their approach was more flexible for ad-hoc queries. We discovered that we could use a hybrid approach: denormalized documents for the main API endpoints and a separate aggregated view for admin dashboards that could tolerate higher latency.

Result: We implemented the hybrid solution that satisfied both performance requirements and

flexibility needs. The collaborative approach strengthened our working relationship, and the senior developer appreciated the data-driven decision process. This experience established a team practice of using proof-of-concepts to resolve technical debates, which improved our overall decision-making quality.

10. Tell me about a time when you had to learn and implement a new Couchbase feature under tight deadlines.

Situation: Our product team committed to delivering a new real-time analytics dashboard feature in three weeks, which required implementing Couchbase Analytics (formerly CBAS) that none of our team had experience with. The feature was critical for a major client demo.

Task: I needed to quickly learn Couchbase Analytics, evaluate its suitability for our use case, implement the solution, and deliver it on time while ensuring it met performance and reliability requirements.

Action: I immediately dove into the official documentation and completed the Couchbase Analytics tutorials. I joined the Couchbase community forums and reached out to Couchbase support for architectural guidance. I created a learning plan that included hands-on experimentation with sample datasets. Within three days, I had a working prototype demonstrating key functionality. I identified potential pitfalls early, such as the need for careful data modeling in Analytics collections and understanding the eventual consistency model. I documented my learnings and conducted a knowledge-sharing session with the team. I implemented the solution with proper error handling and monitoring.

Result: We delivered the analytics dashboard feature on schedule with excellent performance, handling complex analytical queries across 50 million documents in under 2 seconds. The client demo was successful, leading to a contract renewal. My documentation and knowledge transfer enabled two other team members to work confidently with Analytics, and we've since built five additional features using this capability. This experience reinforced my ability to rapidly learn and apply new technologies under pressure.