DynamoDB

Interview Questions and Answers

Core Concepts

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

1. Explain the difference between partition key and composite key in DynamoDB. When would you use each?

Partition Key (Simple Primary Key): A single attribute that uniquely identifies an item. DynamoDB uses this value's hash to determine the physical partition where data is stored.

- Use when items have a naturally unique identifier
- Example: userId as partition key for a users table

Composite Key (Partition Key + Sort Key): Combination of two attributes forming the primary key. Items with the same partition key are stored together, sorted by the sort key.

- Use when you need to store multiple related items together
- Enables range queries and hierarchical data modeling
- Example: userId (partition) + timestamp (sort) for user activity logs

Key Benefits of Composite Keys:

- Query multiple items with the same partition key efficiently
- Use sort key conditions (begins_with, between, <, >, etc.)
- Model one-to-many relationships within a single table

2. What are hot partitions in DynamoDB and how do you prevent them?

Hot Partitions occur when read/write traffic is concentrated on a small subset of partition key values, causing throttling even when total table capacity isn't exceeded. DynamoDB distributes capacity evenly across partitions, so uneven access patterns create bottlenecks.

Prevention Strategies:

- **Write Sharding:** Append random suffix to partition keys (e.g., userId + random(1-10)) to distribute writes
- **Time-based Sharding:** Include date components in partition key for time-series data
- **Composite Keys:** Design partition keys with high cardinality
- **Caching:** Use DAX or ElastiCache for read-heavy workloads
- **On-Demand Mode:** Automatically handles unpredictable traffic spikes

```
// Example: Write sharding
const shardId = Math.floor(Math.random() * 10);
const partitionKey = `USER#${userId}#${shardId}`;

// Read requires scatter-gather across shards
for (let i = 0; i < 10; i++) {
  const key = `USER#${userId}#${i}`;
  // Query each shard
}
```

3. Explain Global Secondary Indexes (GSI) vs Local Secondary Indexes (LSI). What are the trade-offs?

Local Secondary Index (LSI):

- Same partition key as base table, different sort key
- Must be created at table creation time
- Shares throughput capacity with base table
- Strongly consistent reads supported

- Limited to 10 GB per partition key value
- Maximum 5 LSIs per table

Global Secondary Index (GSI):

- Different partition and/or sort key from base table
- Can be added/removed anytime
- Independent throughput capacity
- Eventually consistent reads only
- No size limitations per partition
- Maximum 20 GSIs per table

Trade-offs:

Use LSI when: You need strong consistency and queries share the same partition key with different sort orders.
Use GSI when: You need to query by different attributes entirely, require flexibility to add indexes later, or need independent scaling.

Cost consideration: GSIs consume additional storage and write capacity units for every write to the base table.

4. How does DynamoDB's eventual consistency model work, and when should you use strongly consistent reads?

Eventual Consistency (Default): Read requests may not reflect results of recently completed writes. Data is replicated across multiple Availability Zones, and reads might return stale data from a replica that hasn't received the latest update (typically consistent within one second).

Strong Consistency: Returns the most up-to-date data, reflecting all successful writes prior to the read. DynamoDB routes the request to the primary replica.

Trade-offs:

- **Eventual:** Lower latency, higher throughput, costs 0.5 RCU per 4KB
- **Strong:** Higher latency, costs 1 RCU per 4KB (2x eventual)
- Strong consistency NOT available for GSIs or global tables

When to Use Strong Consistency:

- Read-after-write scenarios requiring immediate accuracy
- Financial transactions or inventory management
- User authentication where stale data could cause security issues

```
// Strongly consistent read
const params = {
  TableName: 'Users',
  Key: { userId: '123' },
  ConsistentRead: true
};
await dynamodb.get(params).promise();
```

5. Describe single-table design in DynamoDB. What are the benefits and challenges?

Single-Table Design stores multiple entity types in one table using composite keys and strategic attribute naming. Entities are distinguished by prefixed partition/sort keys (e.g., USER#123, ORDER#456).

Benefits:

- **Reduced costs:** Fewer tables mean lower provisioned capacity overhead
- **Better performance:** Related data co-located for efficient queries
- **Atomic operations:** Transactions across entity types in one table
- **Simplified access patterns:** One query retrieves related entities
- **Fewer network calls:** Batch operations more effective

Challenges:

- **Complex design:** Requires upfront access pattern analysis
- **Overloaded indexes:** GSIs must accommodate multiple entity types
- **Sparse attributes:** Different entities have different attributes
- **Testing complexity:** Harder to isolate entity types
- **Migration difficulty:** Schema changes affect entire table

```
// Example structure
PK: USER#123, SK: METADATA
PK: USER#123, SK: ORDER#2024-01-15
PK: ORDER#456, SK: METADATA
PK: ORDER#456, SK: ITEM#789
// Query user and their orders in one operation
```

6. How do DynamoDB Streams work, and what are common use cases?

DynamoDB Streams capture time-ordered sequences of item-level modifications (create, update, delete) in a table. Stream records are available for 24 hours and can trigger Lambda functions or be processed by Kinesis Client Library.

Stream View Types:

- **KEYS_ONLY:** Only partition and sort keys
- **NEW_IMAGE:** Entire item after modification
- **OLD_IMAGE:** Entire item before modification
- **NEW_AND_OLD_IMAGES:** Both before and after states

Common Use Cases:

- **Data replication:** Sync to other databases, data lakes, or search indexes (Elasticsearch)
- **Audit logging:** Track all changes for compliance
- **Real-time analytics:** Aggregate metrics and trigger alerts
- **Cross-region replication:** Basis for global tables
- **Materialized views:** Update aggregated data in other tables
- **Event-driven architectures:** Trigger workflows on data changes

```
// Lambda processing stream records
exports.handler = async (event) => {
  for (const record of event.Records) {
    if (record.eventName === 'INSERT') {
      const newItem = record.dynamodb.NewImage;
      // Process new item
    }
  }
};
```

7. Explain DynamoDB transactions. What are the limitations and performance considerations?

DynamoDB Transactions provide ACID guarantees across multiple items and tables, supporting up to 100 actions in a single all-or-nothing operation via TransactWriteItems and TransactGetItems.

Key Features:

- Atomicity: All operations succeed or all fail
- Consistency: Enforces constraints across items
- Isolation: Serializable isolation level
- Condition checks without consuming write capacity

Limitations:

- Maximum 100 items or 4 MB per transaction
- Cannot span AWS accounts or regions
- Costs 2x standard write/read capacity
- Higher latency than single-item operations
- No support for BatchWriteItem within transactions

Performance Considerations:

- Use sparingly for critical operations only
- Avoid transactions in hot partitions
- Consider idempotency tokens for retries

```
const params = {
  TransactItems: [{
    Update: {
      TableName: 'Accounts',
      Key: { id: 'account1' },
      UpdateExpression: 'SET balance = balance - :amt',
      ConditionExpression: 'balance >= :amt',
      ExpressionAttributeValues: { ':amt': 100 }
    }
  }]
};
```

8. What is the difference between provisioned and on-demand capacity modes? When would you choose each?

Provisioned Capacity Mode:

- Specify RCUs (Read Capacity Units) and WCUs (Write Capacity Units)
- Predictable, steady-state traffic
- Lower cost for consistent workloads
- Auto-scaling available but has lag time
- Burst capacity: Short spikes up to 300 seconds
- Reserved capacity available for additional savings

On-Demand Capacity Mode:

- Pay-per-request pricing
- Automatically scales up/down instantly
- No capacity planning required
- 2.5x more expensive per request than provisioned
- Handles unpredictable traffic patterns
- Instantly accommodates up to 2x previous peak

Choose Provisioned When:

- Traffic is predictable and consistent
- Cost optimization is priority
- Application has steady baseline load

Choose On-Demand When:

- New applications with unknown traffic
- Sporadic or unpredictable workloads
- Development/testing environments
- Traffic has extreme spikes

Note: You can switch modes twice per 24 hours.

9. How do you implement optimistic locking in DynamoDB to prevent lost updates?

Optimistic Locking prevents lost updates in concurrent environments by using a version attribute and conditional writes. The pattern ensures that updates only succeed if the item hasn't changed since it was read.

Implementation Strategy:

- Add a version number or timestamp attribute to items
- Read item and note current version
- Include condition expression checking version hasn't changed
- Increment version on successful update
- Handle ConditionalCheckFailedException and retry

```
// Read with current version
const item = await getItem({ userId: '123' });
```

```
const currentVersion = item.version;

// Update with condition
const params = {
  TableName: 'Users',
  Key: { userId: '123' },
  UpdateExpression: 'SET #data = :data, version = :newVer',
  ConditionExpression: 'version = :currentVer',
  ExpressionAttributeValues: {
    ':data': newData,
    ':currentVer': currentVersion,
    ':newVer': currentVersion + 1
  }
};
```

Best Practices:

- Use numeric version for clear incrementing
- Implement exponential backoff for retries
- Consider using transactions for multi-item consistency

10. Explain the concept of adaptive capacity in DynamoDB and how it helps with uneven access patterns.

Adaptive Capacity is DynamoDB's automatic mechanism to isolate frequently accessed items from throttling caused by uneven data access patterns. It responds within minutes to traffic imbalances across partitions.

How It Works:

- DynamoDB monitors partition-level traffic in real-time
- When a partition receives disproportionate traffic, adaptive capacity boosts its throughput
- Borrows unused capacity from less-active partitions
- Operates automatically without configuration
- Works for both provisioned and on-demand modes

Limitations:

- Not instantaneous—takes minutes to activate
- Cannot compensate for sustained hot partition issues
- Limited by the total table capacity
- Doesn't eliminate need for good key design

Complementary Strategies:

- **Burst capacity:** Handles short spikes (up to 300 seconds)
- **Proper key design:** Distribute traffic evenly from the start
- **On-demand mode:** Better for unpredictable spikes

Best Practice: Design for even distribution first; treat adaptive capacity as a safety net, not a primary solution.

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 DynamoDB query results to reduce read costs?

LRU Cache Implementation

An **LRU (Least Recently Used) Cache** evicts the least recently accessed item when capacity is reached. For DynamoDB, this optimizes expensive read operations.

```
class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return None
        self.cache.move_to_end(key)
        return self.cache[key]
```

Time Complexity: O(1) for both get and put operations using OrderedDict. **Space Complexity:** O(capacity).

- Use this to cache frequently accessed DynamoDB items by primary key
- Implement TTL to handle stale data
- Consider Redis or ElastiCache for distributed scenarios

2. Explain how you would use a Hash Set to deduplicate items during a DynamoDB batch write operation.

Deduplication with Hash Sets

A **Hash Set** provides O(1) lookup to prevent duplicate writes, which is critical since DynamoDB batch writes can fail with duplicates.

```
def batch_write_dedupe(items):
    seen = set()
    unique_items = []
    for item in items:
        key = item['id']
        if key not in seen:
            seen.add(key)
            unique_items.append(item)
    return unique_items
```

Key Benefits:

- Prevents wasted WCUs on duplicate writes
- O(1) average time complexity for membership testing
- Space complexity is O(n) where n is unique items
- Essential for idempotent batch operations

3. How would you implement a sliding window algorithm to calculate rolling metrics from DynamoDB time-series data?

Sliding Window for Time-Series

The **sliding window** technique efficiently processes sequential data by maintaining a window of

relevant items, perfect for DynamoDB time-series queries.

```
def sliding_window_avg(data, window_size):
    window_sum = sum(data[:window_size])
    results = [window_sum / window_size]
    for i in range(window_size, len(data)):
        window_sum += data[i] - data[i - window_size]
        results.append(window_sum / window_size)
    return results
```

Use Cases:

- Calculate moving averages for IoT sensor data
- Monitor API request rates over time windows
- Time Complexity: O(n) instead of O(n*k) naive approach
- Combine with DynamoDB Query with ScanIndexForward for chronological processing

4. Describe how to find pairs of items in a DynamoDB scan result that sum to a target value efficiently.

Two-Sum Problem Solution

The **two-pointer or hash map approach** solves pair-sum problems in O(n) time, useful when analyzing DynamoDB query results.

```
def find_pairs_sum(items, target):
    seen = {}
    pairs = []
    for item in items:
        complement = target - item['value']
        if complement in seen:
            pairs.append((seen[complement], item))
        seen[item['value']] = item
    return pairs
```

Applications:

- Match transactions that reconcile to zero
- Find complementary inventory items
- O(n) time and space complexity
- More efficient than nested loops $O(n^2)$

5. How would you implement a Min-Heap to process DynamoDB items by priority or timestamp?

Min-Heap for Priority Processing

A **Min-Heap** maintains the minimum element at the root, enabling efficient priority-based processing of DynamoDB items.

```
import heapq

def process_by_priority(items):
    heap = [(item['priority'], item) for item in items]
    heapq.heapify(heap)
    while heap:
        priority, item = heapq.heappop(heap)
        process_item(item)
```

Performance Characteristics:

- Insert: O(log n), Extract-Min: O(log n)
- Heapify: O(n) for initial construction
- Ideal for processing DynamoDB Streams events by timestamp
- Merge sorted results from multiple DynamoDB queries

6. Explain how a Trie data structure could optimize DynamoDB GSI key prefix searches.

Trie for Prefix Optimization

A **Trie (prefix tree)** can cache and optimize prefix-based searches, reducing expensive DynamoDB begins_with queries.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.keys = []

def build_trie(partition_keys):
    root = TrieNode()
    for key in partition_keys:
        node = root
        for char in key:
            node = node.children.setdefault(char, TrieNode())
        node.keys.append(key)
```

Advantages:

- O(m) search time where m is prefix length
- Cache common prefixes to avoid repeated DynamoDB queries
- Useful for autocomplete on cached GSI keys
- Space trade-off: O(ALPHABET_SIZE * N * M)

7. How would you use a Stack to implement undo functionality for DynamoDB item modifications?

Stack-Based Undo System

A **Stack (LIFO)** structure naturally implements undo/redo by tracking state changes in reverse chronological order.

```
class UndoManager:
    def __init__(self):
        self.undo_stack = []

    def execute(self, action, item_key, old_val, new_val):
        self.undo_stack.append((item_key, old_val))
        # Perform DynamoDB update

    def undo(self):
        if self.undo_stack:
            return self.undo_stack.pop()
```

Key Points:

- O(1) push and pop operations
- Store previous DynamoDB item versions before updates
- Combine with DynamoDB Transactions for atomicity
- Consider max stack size to prevent memory issues

8. Describe implementing a Circular Buffer for batching DynamoDB write requests efficiently.

Circular Buffer for Batch Writes

A **Circular Buffer (Ring Buffer)** provides fixed-size buffering with O(1) operations, ideal for batching DynamoDB writes.

```
class WriteBuffer:
    def __init__(self, size=25):
        self.buffer = [None] * size
        self.head = 0
        self.count = 0

    def add(self, item):
        self.buffer[self.head] = item
        self.head = (self.head + 1) % len(self.buffer)
        self.count = min(self.count + 1, len(self.buffer))
```

Benefits:

- Fixed memory footprint regardless of throughput
- Batch writes when buffer reaches 25 items (DynamoDB limit)
- O(1) insert and remove operations
- Prevents memory growth in high-throughput scenarios

9. How would you implement a Bloom Filter to reduce unnecessary DynamoDB read operations?

Bloom Filter for Read Optimization

A **Bloom Filter** is a probabilistic data structure that tests set membership with no false negatives, reducing wasteful DynamoDB reads.

```
from bitarray import bitarray
import mmh3

class BloomFilter:
    def __init__(self, size, hash_count):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = bitarray(size)

    def add(self, item):
        for i in range(self.hash_count):
            self.bit_array[mmh3.hash(item, i) % self.size] = 1
```

Use Cases:

- Check if key might exist before querying DynamoDB
- Space-efficient: uses bits instead of storing actual keys
- False positive rate controllable via size and hash functions
- Saves RCUs when checking non-existent items

10. Explain how to use a Binary Search Tree to maintain sorted secondary indexes in memory for DynamoDB results.

BST for In-Memory Indexing

A **Binary Search Tree** maintains sorted order with O(log n) operations, useful for secondary sorting of DynamoDB results.

```
class BSTNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None

def insert(root, key, value):
    if not root:
        return BSTNode(key, value)
    if key < root.key:
        root.left = insert(root.left, key, value)
```

Applications:

- Sort DynamoDB query results by non-indexed attributes
- O(log n) insert, search, delete for balanced trees
- In-order traversal yields sorted sequence
- Consider self-balancing trees (AVL, Red-Black) for guaranteed O(log n)
- Alternative to creating additional GSIs

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 DynamoDB. What would be your data model, partition key strategy, and how would you handle high read/write throughput?

Architecture Overview

A URL shortener requires **low-latency reads**, **high write throughput**, and **global distribution**. DynamoDB is ideal due to its single-digit millisecond latency and horizontal scalability.

Data Model

- **Table:** UrlMappings
- **Partition Key:** shortCode (e.g., 'abc123')
- **Attributes:** longUrl, createdAt, expiresAt, userId, clickCount

Key Design Decisions

- **Partition Key Strategy:** Use the short code as PK for uniform distribution and direct lookup. Short codes should be generated using base62 encoding or a distributed ID generator to ensure uniqueness.
- **GSI for User Queries:** Create a GSI with userId as PK and createdAt as SK to query all URLs created by a user.
- **TTL:** Enable DynamoDB TTL on expiresAt attribute to automatically delete expired URLs.
- **Read Optimization:** Use DynamoDB Accelerator (DAX) for caching frequently accessed URLs, reducing read latency to microseconds.
- **Write Throughput:** Use on-demand billing mode for unpredictable traffic or provisioned mode with auto-scaling for cost optimization.

Sample Item Structure

```
{
  "shortCode": "abc123",
  "longUrl": "https://example.com/very/long/url",
  "userId": "user_456",
  "createdAt": 1640000000,
  "expiresAt": 1672000000,
  "clickCount": 0
}
```

Handling Analytics

For click tracking, use **DynamoDB Streams** to trigger Lambda functions that aggregate click data into a separate analytics table or send to Kinesis for real-time processing.

2. How would you design a social media feed system using DynamoDB that supports following relationships, timeline generation, and efficient pagination?

System Components

- **Users Table:** Store user profiles
- **Relationships Table:** Store follower/following connections
- **Posts Table:** Store individual posts
- **Timeline Table:** Store pre-computed user timelines (fan-out on write)

Relationships Table Design

- **PK:** userId
- **SK:** FOLLOWER#followerId or FOLLOWING#followingId
- This allows querying both followers and following lists efficiently

Posts Table Design

- **PK:** userId
- **SK:** timestamp#postId
- **Attributes:** content, mediaUrls, likes, comments
- Enables efficient query of all posts by a user in chronological order

Timeline Strategy: Fan-out on Write

When a user creates a post, use **DynamoDB Streams + Lambda** to:

- Query the Relationships table to get all followers
- Write the post to each follower's timeline table using BatchWriteItem
- This trades write cost for read performance

Timeline Table Design

- **PK:** userId (the viewer)
- **SK:** timestamp#postId
- **Attributes:** authorId, content, mediaUrls

Pagination Implementation

```
const params = {
  TableName: 'Timeline',
  KeyConditionExpression: 'userId = :uid',
  ExpressionAttributeValues: { ':uid': 'user123' },
  ScanIndexForward: false,
  Limit: 20,
  ExclusiveStartKey: lastEvaluatedKey
};
```

Optimization for Celebrities

For users with millions of followers, use **hybrid approach**: fan-out for regular users, pull-based for celebrities with caching via ElastiCache.

3. Design a real-time chat application using DynamoDB. How would you model conversations, ensure message ordering, and handle read receipts efficiently?

Core Requirements

- Support 1-on-1 and group conversations
- Maintain message ordering
- Track read receipts and online status
- Enable message search and pagination

Conversations Table

- **PK:** conversationId
- **SK:** MESSAGE#timestamp#messageId
- **Attributes:** senderId, content, messageType, attachments
- Sort key ensures chronological ordering

Message Ordering Strategy

Use **microsecond-precision timestamps** combined with unique messageId to prevent collisions:

```
const timestamp = Date.now() * 1000 + counter;
const SK = `MESSAGE#${timestamp}#${uuidv4()}`;
```

User Conversations Index

- **GSI:** userId-lastMessageTime-index

- **PK:** userId
- **SK:** lastMessageTime
- Enables querying all conversations for a user sorted by recent activity

Read Receipts Design

Separate table for scalability:

- **Table:** ReadReceipts
- **PK:** conversationId
- **SK:** userId
- **Attributes:** lastReadMessageId, lastReadTimestamp

Real-time Updates

- Use **DynamoDB Streams** to trigger Lambda functions
- Lambda publishes to **AWS IoT Core or AppSync** for WebSocket connections
- Clients subscribe to their conversation channels

Unread Count Optimization

Maintain a separate **UnreadCounts table** updated via Streams to avoid scanning messages.

CAP Theorem Consideration

DynamoDB provides **eventual consistency** by default. For critical operations like payments or message delivery confirmation, use **strongly consistent reads**.

4. How would you design a distributed session store using DynamoDB that handles millions of concurrent users with automatic expiration and cross-region replication?

Table Design

- **Table Name:** UserSessions
- **PK:** sessionId (UUID or secure random token)
- **Attributes:** userId, userData, createdAt, expiresAt, ipAddress, userAgent
- **TTL Attribute:** expiresAt (Unix timestamp)

Automatic Expiration

Enable **DynamoDB TTL** on the expiresAt attribute. DynamoDB automatically deletes expired items within 48 hours, reducing storage costs.

```
const params = {
  TableName: 'UserSessions',
  Item: {
    sessionId: token,
    userId: 'user123',
    userData: {...},
    expiresAt: Math.floor(Date.now()/1000) + 3600
  }
};
```

Access Pattern Optimization

- **Primary Access:** GetItem by sessionId (single-digit ms latency)
- **Secondary Access:** GSI with userId as PK to query all sessions for a user (for logout-all functionality)

Cross-Region Replication

Use **DynamoDB Global Tables** for multi-region active-active replication:

- Automatic replication typically completes in under 1 second
- Provides disaster recovery and low-latency access globally
- Last-writer-wins conflict resolution

Session Refresh Strategy

Implement sliding expiration by updating expiresAt on each request using **UpdateItem with conditional expression** to prevent race conditions:

```
const params = {
  TableName: 'UserSessions',
  Key: { sessionId: token },
  UpdateExpression: 'SET expiresAt = :newExp',
  ConditionExpression: 'attribute_exists(sessionId)',
  ExpressionAttributeValues: {
    ':newExp': newExpiration
  }
};
```

Security Considerations

- Encrypt sensitive session data using **DynamoDB encryption at rest**
- Use **VPC endpoints** to keep traffic private
- Implement rate limiting using API Gateway or application-level throttling

5. Design an e-commerce inventory management system using DynamoDB that handles high-concurrency stock updates, prevents overselling, and supports real-time inventory tracking across multiple warehouses.

Core Challenges

- Prevent overselling (race conditions)
- Support atomic stock decrements
- Track inventory across multiple locations
- Handle high read/write concurrency

Inventory Table Design

- **PK:** productId
- **SK:** warehouseId
- **Attributes:** availableStock, reservedStock, totalStock, lastUpdated

Preventing Overselling with Optimistic Locking

Use **conditional writes** to ensure atomic stock updates:

```
const params = {
  TableName: 'Inventory',
  Key: { productId: 'prod123', warehouseId: 'wh1' },
  UpdateExpression: 'SET availableStock = availableStock - :qty',
  ConditionExpression: 'availableStock >= :qty',
  ExpressionAttributeValues: { ':qty': 5 },
  ReturnValues: 'ALL_NEW'
};
```

If the condition fails, DynamoDB throws **ConditionalCheckFailedException**, preventing overselling.

Two-Phase Reservation System

- **Phase 1 - Reserve:** Move stock from availableStock to reservedStock
- **Phase 2 - Commit:** Decrease reservedStock after payment confirmation
- **Rollback:** Move back to availableStock if payment fails or timeout occurs

Aggregated Inventory View

For product listing pages showing total available stock:

- Use **DynamoDB Streams + Lambda** to maintain aggregated counts in a separate table
- Or query all warehouse locations and sum in application layer with caching

GSI for Low-Stock Alerts

- **GSI:** stockLevel-index

- **PK:** stockLevel (e.g., 'LOW', 'MEDIUM', 'HIGH')
- **SK:** productId
- Update stockLevel attribute via Streams when thresholds are crossed

High Concurrency Strategy

- Use **exponential backoff with jitter** for retry logic on conditional check failures
- Implement **on-demand billing** to handle traffic spikes during flash sales
- Consider **DynamoDB Transactions** for multi-item updates (e.g., bundle purchases)

6. How would you design a serverless job queue system using DynamoDB that supports priority queues, delayed execution, dead-letter queues, and exactly-once processing semantics?

Architecture Components

- **Jobs Table:** Store job metadata and state
- **DynamoDB Streams:** Trigger Lambda workers
- **Priority Queue:** Implemented via GSI with priority sorting
- **DLQ Table:** Store failed jobs for manual inspection

Jobs Table Design

- **PK:** jobId (UUID)
- **SK:** status#timestamp
- **Attributes:** priority, payload, status, retryCount, executeAfter, createdAt
- **Status values:** PENDING, PROCESSING, COMPLETED, FAILED

Priority Queue Implementation

- **GSI:** status-priority-index
- **PK:** status
- **SK:** priority#executeAfter#jobId
- Workers query this index to fetch highest priority jobs ready for execution

```
const params = {
  TableName: 'Jobs',
  IndexName: 'status-priority-index',
  KeyConditionExpression: 'status = :pending AND SK <= :now',
  ExpressionAttributeValues: {
    ':pending': 'PENDING',
    ':now': `${priority}#${Date.now()}`
  },
  Limit: 10
};
```

Exactly-Once Processing

Use **optimistic locking** to claim jobs atomically:

```
const params = {
  TableName: 'Jobs',
  Key: { jobId: id },
  UpdateExpression: 'SET status = :proc, workerId = :wid',
  ConditionExpression: 'status = :pending',
  ExpressionAttributeValues: {
    ':proc': 'PROCESSING',
    ':pending': 'PENDING',
    ':wid': workerId
  }
};
```

Delayed Execution

Set **executeAfter** timestamp. Workers only process jobs where executeAfter <= current time. Use DynamoDB TTL to clean up old completed jobs.

Dead Letter Queue

- Track retryCount in job item
- After max retries, move job to DLQ table using **TransactWriteItems**
- DLQ table has same structure for analysis and manual retry

Handling Worker Failures

Implement **heartbeat mechanism**: workers update lastHeartbeat attribute. A separate Lambda scans for stale jobs and resets them to PENDING.

7. Design a multi-tenant SaaS application using DynamoDB where you need to ensure data isolation, implement per-tenant rate limiting, and optimize costs with different tenant tiers (free, premium, enterprise).

Data Isolation Strategies

Three approaches with tradeoffs:

- **Pool Model (Shared Table):** tenantId as part of composite key - most cost-effective
- **Bridge Model:** Separate tables per tier - balance of isolation and cost
- **Silo Model:** Separate tables per tenant - maximum isolation, highest cost

Recommended: Pool Model with Partition Strategy

- **PK:** tenantId#entityType
- **SK:** entityId#timestamp
- Ensures all tenant data is co-located for efficient queries
- Use IAM policies and attribute-based access control (ABAC) for isolation

```
const params = {
  TableName: 'SaasData',
  KeyConditionExpression: 'PK = :tenant',
  FilterExpression: 'entityType = :type',
  ExpressionAttributeValues: {
    ':tenant': 'tenant123#users',
    ':type': 'USER'
  }
};
```

Per-Tenant Rate Limiting

- Store tenant quotas in **TenantsConfig table**
- Implement application-level rate limiting using **token bucket algorithm**
- Track usage in **ElastiCache** for fast access
- Use **API Gateway usage plans** for API-level throttling per tenant

Tenant Metadata Table

- **PK:** tenantId
- **Attributes:** tier, maxUsers, maxStorage, requestsPerSecond, billingStatus
- Cache this in application memory or ElastiCache

Cost Optimization by Tier

- **Free Tier:** Shared on-demand table with low quotas
- **Premium:** Shared provisioned capacity with auto-scaling
- **Enterprise:** Dedicated tables with reserved capacity, Global Tables for multi-region

Monitoring and Compliance

- Use **DynamoDB Streams** to audit all data access
- Tag resources with tenantId for cost allocation
- Implement **CloudWatch alarms** for per-tenant throttling events

Data Migration Strategy

For tenants upgrading from shared to dedicated tables, use **DynamoDB export to S3** then import to new table, minimizing downtime.

8. Design a leaderboard system for a gaming application using DynamoDB that supports real-time score updates, top-N queries, percentile rankings, and historical leaderboards with millions of players.

Challenge Analysis

- DynamoDB doesn't natively support sorting by non-key attributes
- Need to handle high write throughput for score updates
- Top-N queries require creative indexing
- Global leaderboards vs regional/friend leaderboards

Hybrid Architecture Approach

Combine DynamoDB with **ElastiCache (Redis Sorted Sets)** for optimal performance:

- **DynamoDB:** Source of truth, stores player scores persistently
- **Redis:** Real-time leaderboard queries using ZADD and ZRANGE
- **DynamoDB Streams:** Sync updates to Redis

Scores Table Design

- **PK:** playerId
- **SK:** gameId#timestamp
- **Attributes:** score, rank, percentile, metadata

Pure DynamoDB Approach: Score Buckets

For cost-sensitive implementations without Redis:

- Create **score buckets** (e.g., 0-1000, 1001-2000)
- **GSI:** scoreBucket-score-index
- **PK:** scoreBucket
- **SK:** score#playerId
- Query top buckets first to get top players

```
const params = {
  TableName: 'Scores',
  IndexName: 'scoreBucket-score-index',
  KeyConditionExpression: 'scoreBucket = :bucket',
  ExpressionAttributeValues: { ':bucket': 'bucket_9000' },
  ScanIndexForward: false,
  Limit: 100
};
```

Time-Based Leaderboards

- Create separate tables or partitions for daily/weekly/monthly leaderboards
- Use **TTL** to automatically archive old leaderboards to S3
- Partition key: period#gameId (e.g., '2024-W01#game123')

Percentile Calculation

Maintain aggregate statistics in a separate table updated via Streams:

- Store score distribution buckets
- Calculate percentiles using approximate algorithms (T-Digest)
- Update player percentile attribute asynchronously

Friend Leaderboards

- Query player's friends list from social graph
- BatchGetItem to fetch their scores
- Sort in application layer (limited dataset)
- Cache results in ElastiCache with short TTL

9. How would you design a document versioning system using DynamoDB that supports concurrent edits, conflict resolution, branching, and efficient retrieval of any historical version?

System Requirements

- Store multiple versions of documents
- Support concurrent editing with conflict detection
- Enable branching and merging (similar to Git)
- Efficient retrieval of latest and historical versions
- Minimize storage costs for large documents

Documents Table Design

- **PK:** documentId
- **SK:** version#timestamp
- **Attributes:** content, authorId, parentVersion, conflictStatus, branch, checksum
- Latest version pointer stored separately for quick access

Optimistic Concurrency Control

Use **conditional writes** with version checking:

```
const params = {
  TableName: 'Documents',
  Item: newVersion,
  ConditionExpression: 'attribute_not_exists(SK) AND currentVersion = :expected',
  ExpressionAttributeValues: {
    ':expected': expectedVersion
  }
};
```

If condition fails, a concurrent edit occurred - trigger conflict resolution workflow.

Latest Version Optimization

- Maintain separate item with SK = 'LATEST' pointing to current version
- Use **TransactWriteItems** to atomically update both version and LATEST pointer
- Enables single-item read for most common access pattern

Delta Storage for Cost Optimization

Instead of storing full content for each version:

- Store full content for major versions (every 10th version)
- Store deltas (diffs) for intermediate versions
- Reconstruct versions by applying deltas from last full version
- Trade compute for storage cost

Branching Strategy

- Add **branch** attribute (e.g., 'main', 'feature-x')
- **GSI:** documentId-branch-version
- **PK:** documentId
- **SK:** branch#version
- Enables querying all versions in a specific branch

Conflict Resolution

- **Three-way merge:** Store parentVersion to identify common ancestor
- When conflict detected, create both versions with conflictStatus = 'CONFLICT'
- Use Lambda to attempt automatic merge or flag for manual resolution

Metadata and Search

Separate table for document metadata and search:

- Use **DynamoDB Streams + Lambda** to index content in OpenSearch
- Enables full-text search across all versions

10. Design a distributed rate limiting system using DynamoDB that supports multiple rate limit tiers (per-second, per-minute, per-hour), handles burst traffic, and works

across multiple application servers without coordination.

Rate Limiting Algorithms

- **Token Bucket:** Allows bursts, smooth rate limiting
- **Sliding Window:** More accurate but complex
- **Fixed Window:** Simple but has boundary issues

Recommended: Token Bucket with DynamoDB

- **Table:** RateLimits
- **PK:** userId#resource (e.g., 'user123#api')
- **SK:** window#granularity (e.g., '2024-01-15-10#minute')
- **Attributes:** tokens, capacity, lastRefill, expiresAt

Atomic Token Consumption

```
const params = {
  TableName: 'RateLimits',
  Key: { PK: key, SK: window },
  UpdateExpression: 'SET tokens = tokens - :cost',
  ConditionExpression: 'tokens >= :cost',
  ExpressionAttributeValues: {
    ':cost': 1
  },
  ReturnValues: 'ALL_NEW'
};
```

If ConditionExpression fails, rate limit exceeded.

Multi-Tier Rate Limiting

Create separate items for each time window:

- SK = 'second#2024-01-15-10-30-45'
- SK = 'minute#2024-01-15-10-30'
- SK = 'hour#2024-01-15-10'
- Check all applicable tiers before allowing request

Token Refill Strategy

Calculate refill on-demand rather than background process:

```
const elapsed = now - lastRefill;
const refillAmount = Math.floor(elapsed * refillRate);
const newTokens = Math.min(capacity, tokens + refillAmount);
```

Handling Distributed Servers

- DynamoDB's conditional writes provide **distributed coordination**
- No need for Redis or centralized rate limiter
- Each server independently checks and updates counters
- Strong consistency ensures accuracy

Performance Optimization

- Use **DAX** for caching rate limit checks (microsecond latency)
- Implement **local in-memory cache** with short TTL (100-500ms) to reduce DynamoDB calls
- Accept slight inaccuracy for better performance

TTL for Automatic Cleanup

- Set expiresAt to window end + grace period
- DynamoDB automatically deletes old window records

Burst Allowance

Set capacity higher than sustained rate to allow temporary bursts while maintaining average rate

limit over time.

Coding and Debugging

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

1. Write a Python function to efficiently query DynamoDB items with pagination and handle rate limiting.

Efficient DynamoDB Query with Pagination

This solution uses exponential backoff for rate limiting and properly handles pagination:

```
import boto3
import time
from botocore.exceptions import ClientError

def query_with_pagination(table_name, key_condition):
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)
    items, retries = [], 0
    params = {'KeyConditionExpression': key_condition}
    while True:
        try:
            response = table.query(**params)
            items.extend(response['Items'])
            if 'LastEvaluatedKey' not in response:
                break
            params['ExclusiveStartKey'] = response['LastEvaluatedKey']
        except ClientError as e:
            if e.response['Error']['Code'] == 'ProvisionedThroughputExceededException':
                time.sleep(2 ** retries)
                retries += 1
            else:
                raise
    return items
```

Key points:

- Uses **ExclusiveStartKey** for pagination to retrieve all results
- Implements exponential backoff for throttling exceptions
- Accumulates items across multiple pages
- Handles ProvisionedThroughputExceededException gracefully

2. How would you debug a scenario where DynamoDB BatchWriteItem is silently failing for some items?

Debugging BatchWriteItem Silent Failures

BatchWriteItem doesn't throw exceptions for individual item failures. Here's how to debug:

```
def batch_write_with_retry(table_name, items):
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)
    with table.batch_writer() as batch:
        for item in items:
            batch.put_item(Item=item)

# Manual approach with error checking:
response = dynamodb.batch_write_item(
    RequestItems={'MyTable': [{'PutRequest': {'Item': item}}]}
)
unprocessed = response.get('UnprocessedItems', {})
if unprocessed:
```

```
    print(f"Failed items: {unprocessed}")
```

Common debugging steps:

- Check **UnprocessedItems** in the response - contains failed writes
- Enable CloudWatch Logs for DynamoDB to see throttling metrics
- Verify item size doesn't exceed 400KB limit
- Check for duplicate primary keys in the batch
- Use **batch_writer()** context manager which auto-retries unprocessed items
- Monitor **ConsumedCapacity** to detect throughput issues

3. Write code to implement optimistic locking in DynamoDB to prevent concurrent update conflicts.

Optimistic Locking with Version Numbers

Prevent race conditions using conditional updates with version tracking:

```
def update_with_optimistic_lock(table_name, key, updates):
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)
    try:
        response = table.get_item(Key=key)
        current_version = response['Item'].get('version', 0)
        table.update_item(
            Key=key,
            UpdateExpression='SET #data = :val, version = :new_ver',
            ConditionExpression='version = :curr_ver',
            ExpressionAttributeNames={'#data': 'data'},
            ExpressionAttributeValues={
                ':val': updates, ':curr_ver': current_version,
                ':new_ver': current_version + 1
            }
        )
        return True
    except ClientError as e:
        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            return False
        raise
```

Key concepts:

- Store a **version** attribute that increments with each update
- Use **ConditionExpression** to ensure version hasn't changed
- Catch **ConditionalCheckFailedException** to detect conflicts
- Retry logic should re-fetch and merge changes

4. How do you profile and optimize a slow DynamoDB scan operation consuming excessive read capacity?

Profiling and Optimizing Scan Operations

Diagnosis techniques:

- Enable **ReturnConsumedCapacity='TOTAL'** to measure RCU usage
- Use CloudWatch metrics: **ConsumedReadCapacityUnits** and **ThrottledRequests**
- Check scan duration with Python's time module or X-Ray tracing
- Analyze **ScannedCount** vs **Count** to detect inefficient filtering

Optimization strategies:

```
# Use parallel scans for large tables
def parallel_scan(table_name, segments=4):
    from concurrent.futures import ThreadPoolExecutor
    def scan_segment(segment):
        table = boto3.resource('dynamodb').Table(table_name)
        return table.scan(
            Segment=segment, TotalSegments=segments,
            ProjectionExpression='id,name',
```

```
            FilterExpression='attribute_exists(active)'
        )['Items']
    with ThreadPoolExecutor(max_workers=segments) as executor:
        results = executor.map(scan_segment, range(segments))
    return [item for sublist in results for item in sublist]
```

- Use **ProjectionExpression** to fetch only needed attributes
- Implement parallel scans with **TotalSegments**
- Consider switching to Query with GSI/LSI if access pattern allows
- Use **FilterExpression** server-side instead of client filtering

5. Debug this DynamoDB query that returns empty results despite data existing in the table.

Debugging Empty Query Results

Common causes and solutions:

```
# Incorrect - using Scan with KeyConditionExpression
response = table.scan(
    KeyConditionExpression=Key('userId').eq('123')
)

# Correct - use Query for key-based lookups
response = table.query(
    KeyConditionExpression=Key('userId').eq('123')
)

# Check data types match
response = table.query(
    KeyConditionExpression=Key('userId').eq(123)  # Number not String
)
```

Debugging checklist:

- **Scan vs Query:** KeyConditionExpression only works with query(), not scan()
- **Data type mismatch:** Ensure key values match table schema (String vs Number)
- **Case sensitivity:** DynamoDB is case-sensitive for attribute names and values
- **Index usage:** Verify querying correct index with IndexName parameter
- **Partition key required:** Query must include partition key in KeyConditionExpression
- **FilterExpression timing:** Applied after read, doesn't reduce RCU consumption
- Use **DescribeTable** to verify key schema and attribute types

6. Implement a DynamoDB transaction that atomically updates multiple items with rollback on failure.

DynamoDB Transactions with Rollback

Transactions ensure ACID properties across multiple items:

```
def transfer_balance(from_id, to_id, amount):
    dynamodb = boto3.client('dynamodb')
    try:
        response = dynamodb.transact_write_items(
            TransactItems=[
                {'Update': {
                    'TableName': 'Accounts',
                    'Key': {'userId': {'S': from_id}},
                    'UpdateExpression': 'SET balance = balance - :amt',
                    'ConditionExpression': 'balance >= :amt',
                    'ExpressionAttributeValues': {':amt': {'N': str(amount)}}
                }},
                {'Update': {
                    'TableName': 'Accounts',
                    'Key': {'userId': {'S': to_id}},
                    'UpdateExpression': 'SET balance = balance + :amt',
                    'ExpressionAttributeValues': {':amt': {'N': str(amount)}}
                }}
            ]
```

```
        )
        return True
    except ClientError as e:
        if e.response['Error']['Code'] == 'TransactionCanceledException':
            reasons = e.response['Error']['CancellationReasons']
            print(f"Transaction failed: {reasons}")
        return False
```

Key features:

- All operations succeed or all fail (atomic)
- Supports up to 25 items per transaction
- Conditions validated before commit
- Automatic rollback on any failure

7. How would you debug and fix hot partition issues causing throttling in DynamoDB?

Debugging and Fixing Hot Partitions

Detection methods:

- CloudWatch metric: **UserErrors** for ProvisionedThroughputExceededException
- Enable **CloudWatch Contributor Insights** to identify hot keys
- Use **X-Ray** to trace throttled requests
- Check **SystemErrors** for internal throttling

Solutions:

```
# Add random suffix to distribute writes
import random
def write_with_sharding(user_id, data):
    shard = random.randint(0, 9)
    partition_key = f"{user_id}#{shard}"
    table.put_item(Item={'pk': partition_key, 'data': data})

# Read from all shards
def read_with_sharding(user_id):
    items = []
    for shard in range(10):
        pk = f"{user_id}#{shard}"
        items.extend(table.query(
            KeyConditionExpression=Key('pk').eq(pk)
        )['Items'])
    return items
```

- **Write sharding:** Distribute hot keys across multiple partition key values
- **Caching:** Use DAX or ElastiCache for read-heavy workloads
- **On-demand mode:** Switch from provisioned to on-demand capacity
- **Burst capacity:** Understand DynamoDB saves unused capacity for bursts

8. Write code to handle DynamoDB Streams for real-time data replication with error handling.

DynamoDB Streams Processing

Process change data capture events with proper error handling:

```
def lambda_handler(event, context):
    for record in event['Records']:
        try:
            if record['eventName'] == 'INSERT':
                new_item = record['dynamodb']['NewImage']
                replicate_to_target(new_item)
            elif record['eventName'] == 'MODIFY':
                old_item = record['dynamodb']['OldImage']
                new_item = record['dynamodb']['NewImage']
                update_target(old_item, new_item)
            elif record['eventName'] == 'REMOVE':
                old_item = record['dynamodb']['OldImage']
```

```
            delete_from_target(old_item)
        except Exception as e:
            print(f"Error processing {record['eventID']}: {e}")
            # Send to DLQ for retry
            raise
    return {'statusCode': 200}
```

Best practices:

- Use **NewImage** and **OldImage** to access item data
- Enable **StreamViewType** as NEW_AND_OLD_IMAGES
- Configure DLQ for failed records
- Set appropriate batch size (default 100)
- Use **bisect on error** for partial batch failures
- Monitor **IteratorAge** metric for processing lag

9. Debug memory issues when querying large result sets from DynamoDB in a Lambda function.

Debugging Lambda Memory Issues with DynamoDB

Problem diagnosis:

- Monitor CloudWatch Logs for "Runtime exited with error: signal: killed"
- Check Lambda metrics: **MaxMemoryUsed** vs configured memory
- Large result sets load entirely into memory before processing

Solution - Use generators for streaming:

```
def query_items_generator(table_name, key_condition):
    table = boto3.resource('dynamodb').Table(table_name)
    params = {'KeyConditionExpression': key_condition}
    while True:
        response = table.query(**params)
        for item in response['Items']:
            yield item  # Process one item at a time
        if 'LastEvaluatedKey' not in response:
            break
        params['ExclusiveStartKey'] = response['LastEvaluatedKey']

# Usage
for item in query_items_generator('MyTable', Key('pk').eq('123')):
    process_item(item)  # Memory efficient
```

Additional optimizations:

- Use **ProjectionExpression** to reduce payload size
- Increase Lambda memory (also increases CPU)
- Process in batches with **Limit** parameter
- Consider Step Functions for large datasets

10. Implement a retry mechanism with exponential backoff and jitter for DynamoDB operations.

Advanced Retry with Exponential Backoff and Jitter

Prevent thundering herd problem with randomized delays:

```
import time
import random
from botocore.exceptions import ClientError

def retry_with_backoff(func, max_retries=5, base_delay=0.1):
    for attempt in range(max_retries):
        try:
            return func()
        except ClientError as e:
            if e.response['Error']['Code'] not in [
                'ProvisionedThroughputExceededException',
```

```
            'ThrottlingException'
        ]:
            raise
        if attempt == max_retries - 1:
            raise
        delay = min(base_delay * (2 ** attempt), 20)
        jitter = random.uniform(0, delay * 0.1)
        time.sleep(delay + jitter)

# Usage
result = retry_with_backoff(
    lambda: table.put_item(Item={'id': '123', 'data': 'value'})
)
```

Key features:

- **Exponential backoff:** Delay doubles with each retry
- **Jitter:** Random component prevents synchronized retries
- **Max delay cap:** Prevents excessive wait times
- **Selective retry:** Only retry throttling errors
- boto3 has built-in retries (default 5), this adds application-level control

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 DynamoDB table that was experiencing performance issues.

Situation: Our e-commerce application was experiencing slow query response times during peak hours, with DynamoDB read latency exceeding 500ms for product catalog queries.

Task: I was tasked with identifying the bottleneck and reducing query latency to under 100ms while staying within budget constraints.

Action: I analyzed CloudWatch metrics and discovered hot partition issues due to poor partition key design. I redesigned the schema using a composite key with product category as the partition key and product ID as the sort key. I also implemented a Global Secondary Index (GSI) for frequently queried attributes and enabled DynamoDB Accelerator (DAX) for caching frequently accessed items.

Result: Query latency dropped to an average of 45ms, hot partitions were eliminated, and we reduced read capacity units by 40%, saving approximately $2,000 monthly while improving user experience significantly.

2. Describe a situation where you had to choose between DynamoDB and a relational database for a project.

Situation: We were architecting a real-time analytics platform for tracking user events across multiple applications, expecting millions of writes per day with unpredictable traffic patterns.

Task: I needed to evaluate and recommend the most suitable database solution that could handle high write throughput, scale automatically, and maintain low latency.

Action: I created a comparison matrix evaluating DynamoDB, Aurora, and PostgreSQL RDS based on scalability, cost, latency, and operational overhead. I conducted proof-of-concept tests simulating peak loads. DynamoDB's auto-scaling, single-digit millisecond latency, and serverless nature made it ideal for our write-heavy, unpredictable workload. I presented findings showing DynamoDB could handle 50,000 writes/second with minimal operational overhead.

Result: We selected DynamoDB, which successfully handled peak loads of 75,000 writes/second during product launches. The solution required zero database administration and scaled seamlessly, reducing operational costs by 60% compared to managing RDS instances.

3. Share an example of when you had to handle a DynamoDB capacity planning challenge.

Situation: Our mobile gaming application was preparing for a global launch, and initial estimates suggested we'd need substantial DynamoDB capacity, but actual usage patterns were uncertain.

Task: I was responsible for designing a capacity strategy that could handle launch day traffic spikes without over-provisioning and wasting budget.

Action: I implemented a hybrid approach using on-demand mode for the first two weeks to capture actual usage patterns without throttling risks. I set up comprehensive CloudWatch dashboards and alarms to monitor consumed capacity. After collecting baseline metrics, I analyzed access patterns and calculated that provisioned capacity with auto-scaling would be 55% more cost-effective. I created Lambda functions to automatically adjust capacity based on time-of-day patterns.

Result: We successfully handled 2 million concurrent users during launch without any throttling incidents. After transitioning to provisioned capacity with auto-scaling, we reduced monthly DynamoDB costs from $12,000 to $5,400 while maintaining performance SLAs.

4. Tell me about a time when you had to migrate data from one DynamoDB table design to another without downtime.

Situation: Our user profile system had a poorly designed schema that couldn't support new query patterns for our recommendation engine, requiring a complete table redesign with different partition and sort keys.

Task: I needed to migrate 50 million user records to the new table structure without any service interruption or data loss.

Action: I implemented a dual-write strategy where the application wrote to both old and new tables simultaneously. I created an EMR job using Apache Spark to backfill historical data from the old table to the new one, processing data in batches to avoid throttling. I implemented feature flags to gradually route read traffic to the new table, starting with 5% of users and monitoring for issues. I set up DynamoDB Streams on the old table to capture any updates during migration.

Result: The migration completed successfully over 10 days with zero downtime. Data consistency was maintained at 100%, and the new schema enabled 3 additional query patterns that improved recommendation engine performance by 70%. We decommissioned the old table after two weeks of validation.

5. Describe a situation where you had to debug and resolve DynamoDB throttling issues in production.

Situation: Our order processing system started experiencing intermittent failures during business hours, with error logs showing ProvisionedThroughputExceededException errors from DynamoDB.

Task: I was assigned to identify the root cause and implement a solution quickly, as order failures were impacting revenue.

Action: I analyzed CloudWatch metrics and discovered that while average capacity utilization was only 60%, specific partition keys were being accessed disproportionately, causing hot partitions. I identified that orders were partitioned by date, causing all concurrent orders to hit the same partition. I redesigned the partition key to include a hash suffix derived from order ID, distributing writes across multiple partitions. I also implemented exponential backoff with jitter in the application code and enabled auto-scaling with more aggressive scaling policies.

Result: Throttling errors dropped from 1,200 per hour to zero. Order processing throughput increased by 3x, and we could handle Black Friday traffic (10x normal load) without issues. The solution was documented and became our standard pattern for high-throughput tables.

6. Tell me about a complex DynamoDB access pattern you had to implement using Global Secondary Indexes.

Situation: Our project management application needed to support multiple query patterns: tasks by project, tasks by assignee, tasks by due date, and tasks by status, but our initial design only supported queries by project.

Task: I needed to enable these additional access patterns efficiently without scanning the entire table or creating duplicate data stores.

Action: I designed three sparse Global Secondary Indexes (GSIs) to support the different query patterns. For tasks by assignee, I created a GSI with assignee ID as partition key and due date as sort key. For overdue tasks, I used a sparse index that only projected items with a status attribute set to 'active'. I carefully selected projected attributes to minimize storage costs while maintaining query performance. I also implemented composite sort keys to enable range queries on multiple attributes.

Result: All four query patterns achieved sub-50ms latency. The GSI storage overhead was only 30% of base table size due to sparse indexing and selective projection. The solution eliminated the need for Scan operations, reducing read costs by 85% and enabling the application to scale to 500,000 tasks across 10,000 projects.

7. Share an experience where you had to implement a DynamoDB solution that required strong consistency guarantees.

Situation: We were building a financial ledger system where account balance accuracy was critical, and eventual consistency could lead to overdrafts or incorrect balance displays.

Task: I needed to design a DynamoDB-based solution that guaranteed strong consistency for balance reads while maintaining acceptable performance.

Action: I implemented strongly consistent reads for all balance query operations by setting the

ConsistentRead parameter to true. To handle concurrent updates safely, I used conditional writes with expected attribute values to implement optimistic locking. I designed the partition key using account ID to ensure all operations for a single account hit the same partition, enabling transactional operations. I also implemented DynamoDB transactions for operations that needed to update multiple accounts atomically (transfers between accounts).

Result: The system processed 100,000 transactions daily with zero consistency issues or balance discrepancies. Strongly consistent reads added only 15ms average latency compared to eventually consistent reads. The solution passed financial audits and supported atomic multi-account transfers, enabling us to launch new features like instant peer-to-peer payments.

8. Describe a time when you had to implement cost optimization strategies for DynamoDB without sacrificing performance.

Situation: Our DynamoDB costs had grown to $18,000 monthly, and management requested a 40% cost reduction while maintaining current performance levels.

Task: I was assigned to analyze usage patterns and implement cost optimization strategies across our 15 DynamoDB tables.

Action: I conducted a comprehensive audit using AWS Cost Explorer and CloudWatch Contributor Insights. I identified that 3 tables with infrequent access were good candidates for DynamoDB Standard-IA storage class, saving 60% on storage costs. I implemented TTL (Time To Live) on 2 tables to automatically delete expired data, reducing storage by 40%. For tables with predictable traffic, I switched from on-demand to provisioned capacity with auto-scaling. I also optimized GSIs by removing unnecessary projections and deleting 2 unused indexes. I implemented DAX caching for read-heavy tables, reducing read capacity needs by 70%.

Result: Monthly costs decreased from $18,000 to $9,800 (46% reduction), exceeding the target. Query latency improved by 25% due to DAX implementation. Storage usage decreased by 35%, and the optimizations were documented as best practices for future projects.

9. Tell me about a time when you had to design a DynamoDB schema for a multi-tenant application.

Situation: We were building a SaaS platform for project management where each customer (tenant) needed isolated data, but we wanted to avoid creating separate tables per tenant due to AWS service limits and operational complexity.

Task: I needed to design a single-table schema that supported data isolation, efficient querying per tenant, and the ability to scale to 10,000+ tenants.

Action: I implemented a single-table design using a composite partition key pattern with tenant ID as a prefix (e.g., 'TENANT#12345#PROJECT#67890'). I used generic attribute names (PK, SK, GSI1PK, GSI1SK) to support multiple entity types in one table. I implemented fine-grained IAM policies and application-level access controls to ensure tenant isolation. I designed GSIs to support cross-entity queries within a tenant. I also implemented request throttling per tenant at the application layer to prevent noisy neighbor issues.

Result: The single-table design supported 5,000 tenants with complete data isolation. Query performance remained consistent at 30ms average latency regardless of tenant size. We avoided the 256 table limit per region and reduced operational overhead by 90% compared to managing multiple tables. The design scaled seamlessly to 15,000 tenants over two years.

10. Share an example of when you had to implement disaster recovery and backup strategies for critical DynamoDB data.

Situation: Our healthcare application stored sensitive patient data in DynamoDB, and regulatory requirements mandated point-in-time recovery capabilities and cross-region disaster recovery with RPO under 1 hour and RTO under 4 hours.

Task: I was responsible for designing and implementing a comprehensive backup and disaster recovery strategy that met compliance requirements.

Action: I enabled Point-in-Time Recovery (PITR) on all production tables to support recovery to any point within the last 35 days. I configured AWS Backup to create daily snapshots with 90-day retention and automated cross-region snapshot copying to a secondary region. I implemented DynamoDB Global Tables for critical tables requiring active-active replication across two regions. I created runbooks and automated recovery scripts using AWS Lambda and Step Functions. I

conducted quarterly disaster recovery drills to validate RTO/RPO targets.

Result: The solution met all compliance requirements and was validated during audits. During a real incident where a developer accidentally deleted 10,000 records, we recovered data within 20 minutes using PITR with zero data loss. The Global Tables setup provided automatic failover capabilities, and our DR drills consistently achieved RTO of 2 hours, well under the 4-hour requirement.