Amazon DynamoDB Fundamentals

AWS Solutions Architect Associate Exam Study Notes

Table of Contents

Introduction and Overview

What is Amazon DynamoDB?

Amazon DynamoDB is a **serverless, NoSQL, fully managed database service** that provides single-digit millisecond performance at any scale. DynamoDB was launched in 2012 and continues to help organizations move away from relational databases while reducing cost and improving performance at scale.

Key Characteristics

Serverless Database Service

- No servers to provision, patch, manage, install, maintain, or operate
- Zero downtime maintenance with no maintenance windows
- No versions (major, minor, or patch) to manage

NoSQL Database

- Purpose-built for improved performance, scalability, and flexibility
- Supports both key-value and document data models
- Flexible schema - not every item needs the same attributes
- No JOIN operations (data denormalization recommended)

Fully Managed Service

- AWS handles infrastructure, scaling, patching, monitoring
- Built-in high availability and durability
- Automatic scaling and performance optimization

High Performance

- Consistent single-digit millisecond performance at any scale
- Handles millions to billions of requests
- No cold starts with on-demand scaling

Core Concepts and Terminology

Fundamental Components

Tables

- Collection of data items (similar to tables in relational databases)
- Schemaless except for required primary key
- Can store unlimited number of items
- Data is automatically distributed across partitions

Items

- Individual records in a table (similar to rows)
- Group of attributes uniquely identifiable
- No limit to number of items per table
- Each item can have different attributes (flexible schema)

Attributes

- Basic data elements within items (similar to columns)
- Fundamental data that doesn't need further breakdown

- Support various data types: String, Number, Binary, Boolean, Null, Document, Set

Data Types

Scalar Types

- String (S): UTF-8 encoded text
- Number (N): Positive/negative numbers, decimals
- Binary (B): Binary data, images, compressed objects
- Boolean (BOOL): True or false
- Null (NULL): Represents blank/empty value

Document Types

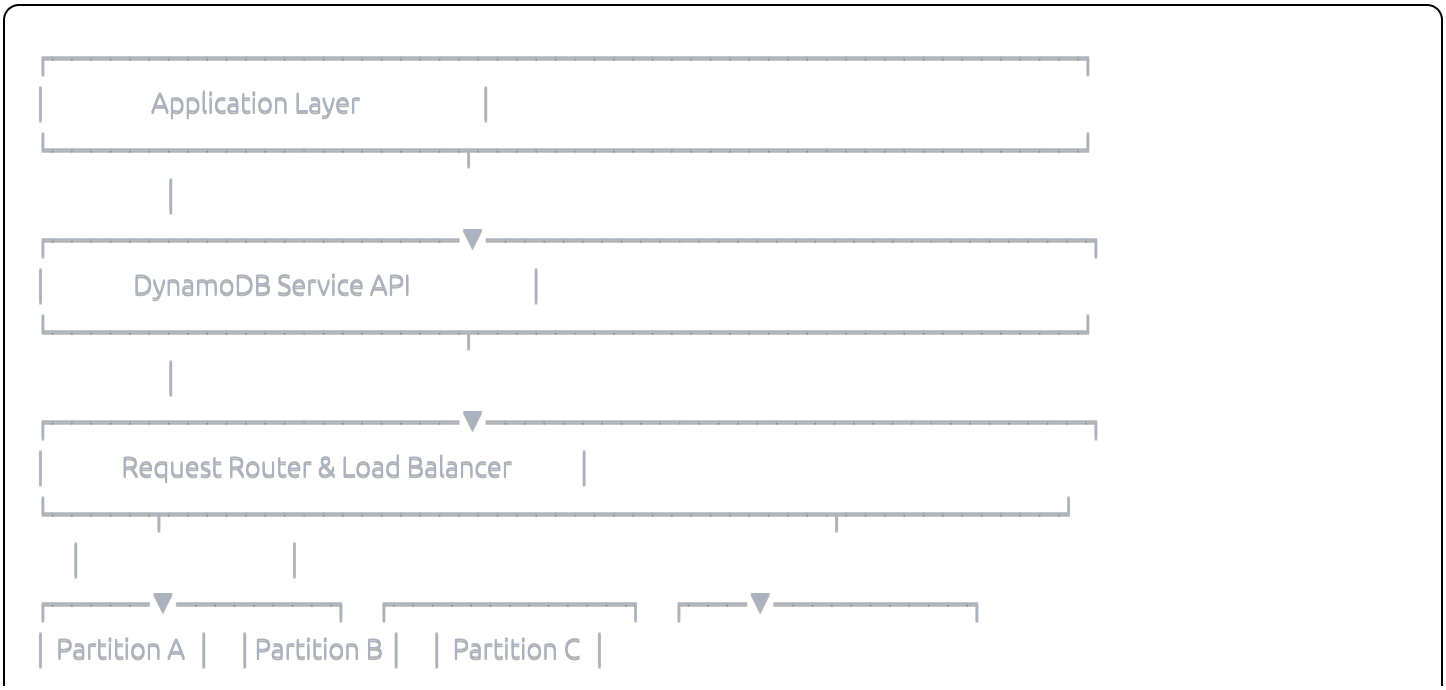
- List (L): Ordered collection of values
- Map (M): Unordered collection of name-value pairs

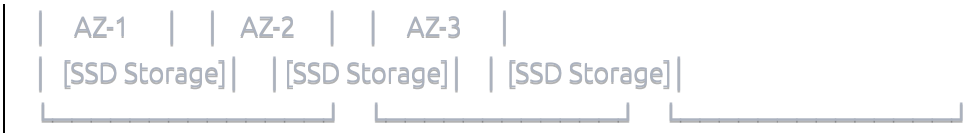
Set Types

- String Set (SS): Set of strings
- Number Set (NS): Set of numbers
- Binary Set (BS): Set of binary values

DynamoDB Architecture

Distributed Architecture





Data Distribution

Partitioning

- Data distributed across multiple partitions based on partition key
- Automatic partition management and scaling
- Even distribution ensures consistent performance

Replication

- Data automatically replicated across 3 Availability Zones
- Synchronous replication for durability
- Built-in fault tolerance

Primary Keys and Data Modeling

Primary Key Types

Simple Primary Key (Partition Key Only)

- Uses only partition key to uniquely identify items
- Partition key must be unique across all items
- Good for simple lookup patterns

Example:

```
Table: Users
Primary Key: UserId (Partition Key)
```

Composite Primary Key (Partition Key + Sort Key)

- Combination of partition key and sort key creates unique identifier
- Multiple items can have same partition key but different sort keys
- Enables range queries and sorting

Example:

```
Table: GameScores
Primary Key: UserId (Partition Key) + GameTitle (Sort Key)
```

Data Modeling Best Practices

Single Table Design

- Store multiple entity types in one table
- Use partition key and sort key strategically
- Reduces number of tables and simplifies access patterns

Access Pattern Driven Design

1. Identify all access patterns first
2. Design primary key structure around most common patterns
3. Use secondary indexes for alternative access patterns
4. Denormalize data to avoid JOINs

Denormalization

- Store related data together in single item
- Duplicate data across items when necessary
- Trade storage space for query performance

Basic Operations and Query Patterns

Core Operations

Item Operations

- **PutItem**: Create or replace an item
- **GetItem**: Retrieve an item by primary key
- **UpdateItem**: Modify attributes of an existing item
- **DeleteItem**: Remove an item from table

Batch Operations

- **BatchGetItem**: Retrieve up to 100 items across multiple tables
- **BatchWriteItem**: Put or delete up to 25 items in single request

Query Patterns

Query Operation

- Most efficient way to retrieve data
- Uses primary key or secondary index
- Returns items with same partition key
- Can filter by sort key range
- Supports forward and backward scanning

```sql
-- Example PartiQL Query
SELECT * FROM GameScores
WHERE UserId = '123' AND GameTitle = 'Football Manager'
```

Scan Operation

- Examines every item in table or index
- Less efficient than Query
- Can apply filters after retrieval
- Consumes read capacity for entire scan
- Use sparingly for large tables

Secondary Indexes

Global Secondary Index (GSI)

- Different partition key and sort key from base table
- Spans all partitions in table
- Eventually consistent reads only
- Has its own capacity settings

Local Secondary Index (LSI)

- Same partition key as base table
- Different sort key from base table
- Strongly consistent reads possible
- Shares capacity with base table

- Maximum 10 GB per partition

Capacity Management and Pricing

Capacity Modes

On-Demand Capacity Mode

- Pay-per-request pricing model
- Automatic scaling with no capacity planning
- Instantly accommodates traffic spikes
- Good for unpredictable workloads
- Scales down to zero when not in use

Provisioned Capacity Mode

- Reserve read and write capacity units
- More cost-effective for predictable traffic
- Auto Scaling available to adjust capacity
- Good for consistent, predictable workloads

Capacity Units

Read Capacity Units (RCU)

- 1 RCU = 1 strongly consistent read/second for item ≤ 4KB
- 1 RCU = 2 eventually consistent reads/second for item ≤ 4KB
- Larger items consume proportionally more RCUs

Write Capacity Units (WCU)

- 1 WCU = 1 write/second for item ≤ 1KB
- Larger items consume proportionally more WCUs

Pricing Components

Primary Costs

- Read and write request charges
- Data storage (per GB/month)

- Data transfer out of AWS

Additional Features

- Global tables cross-region replication
- DynamoDB Streams
- DynamoDB Accelerator (DAX)
- Backup and restore operations
- Point-in-time recovery

Free Tier

- 25 GB storage
- 25 RCUs and 25 WCUs (200M requests/month)
- Always available (not just first year)

Security Features

Authentication and Authorization

AWS Identity and Access Management (IAM)

- No DynamoDB usernames or passwords
- Centralized permission management
- Resource-based and identity-based policies
- Fine-grained access control at attribute level

Access Control Patterns

```
json
```

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:Query"
      ],
      "Resource": "arn:aws:dynamodb:region:account:table/GameScores",
      "Condition": {
        "ForAllValues:StringEquals": {
          "dynamodb:Attributes": ["UserId", "GameTitle", "Score"]
        }
      }
    }
  ]
}
```

Encryption

Encryption at Rest

- Default encryption using AWS-owned keys (no additional cost)
- AWS managed keys (KMS)
- Customer managed keys for compliance requirements
- Transparent encryption/decryption
- No performance impact

Encryption in Transit

- TLS 1.2 encryption for all API calls
- Secure communication between client and service
- VPC endpoints for private connectivity

Compliance Standards

- HIPAA eligible
- PCI DSS compliance
- GDPR compliance

- SOC certifications
- ISO 27001

Resilience and High Availability

Built-in Durability

Multi-AZ Replication

- Automatic replication across 3 Availability Zones
- Synchronous replication for consistency
- 99.99% availability SLA for single region

Global Tables

- Multi-active replication across AWS Regions with 99.999% availability
- Eventually consistent cross-region replication
- Automatic conflict resolution
- Local read/write performance in each region

Backup and Recovery

Point-in-Time Recovery (PITR)

- Continuous backups with per-second granularity
- Restore to any point within last 35 days
- No impact on table performance
- Configurable retention period (1-35 days)

On-Demand Backups

- Full table backups for long-term retention
- No size limitations
- Cross-account and cross-region backup copying
- Integration with AWS Backup service

Service Integrations

Serverless Integrations

AWS Lambda

- Event-driven triggers via DynamoDB Streams
- Serverless compute for data processing
- Automatic scaling with table traffic

API Gateway

- RESTful API endpoints for DynamoDB
- Authentication and authorization integration
- Request/response transformation

AWS AppSync

- GraphQL APIs with real-time subscriptions
- Offline synchronization capabilities
- Multi-data source federation

Analytics Integrations

Zero-ETL Integrations

- Amazon Redshift for complex analytics
- Amazon OpenSearch for full-text and vector search
- No impact on production workloads

Data Export/Import

- Export to Amazon S3 (full and incremental)
- Import from S3 to new tables
- Integration with analytics pipelines

Monitoring and Management

Amazon CloudWatch

- Performance metrics and alarms
- Capacity utilization monitoring

- Custom dashboards

AWS CloudTrail

- API call logging and auditing
- Compliance and governance
- Security event monitoring

Performance and Cost Optimization

Performance Optimization

Hot Partitions

- Distribute traffic evenly across partitions
- Use high-cardinality partition keys
- Avoid sequential patterns in partition keys

Query Optimization

- Use Query instead of Scan when possible
- Implement pagination for large result sets
- Use projection expressions to limit returned data
- Leverage eventually consistent reads when appropriate

DynamoDB Accelerator (DAX)

- In-memory caching cluster
- Up to 10x performance improvement (milliseconds to microseconds)
- Fully managed with automatic scaling
- Transparent to application code

Cost Optimization Strategies

Capacity Mode Selection

- Use On-Demand for unpredictable workloads
- Use Provisioned with Auto Scaling for predictable patterns
- Monitor capacity utilization with CloudWatch

Storage Optimization

- Use TTL (Time To Live) to automatically delete expired items
- Implement data archival strategies
- Consider DynamoDB Standard-IA for infrequent access

Feature Optimization

- Minimize global table regions
- Use appropriate backup retention periods
- Optimize secondary index usage

Best Practices

Design Patterns

Access Pattern Analysis

1. List all application queries and access patterns
2. Identify read vs write frequency
3. Design table schema around most common patterns
4. Use secondary indexes for alternative patterns

Data Modeling Principles

- Start with access patterns, not entity relationships
- Denormalize data for read performance
- Use single table design where appropriate
- Minimize round trips between application and database

Partition Key Design

- Choose high-cardinality attributes
- Ensure even distribution of requests
- Avoid hot partitions
- Consider composite keys for hierarchical data

Operational Best Practices

Monitoring and Alerting

- Set up CloudWatch alarms for throttling
- Monitor consumed vs provisioned capacity
- Track error rates and latency metrics
- Use AWS X-Ray for distributed tracing

Security Configuration

- Implement least privilege access policies
- Enable encryption at rest
- Use VPC endpoints for private access
- Regular access review and audit

Backup Strategy

- Enable point-in-time recovery for critical tables
- Schedule regular on-demand backups
- Test restore procedures regularly
- Document recovery procedures

Demo Scenario: Football League Management

Scenario Overview

Create a DynamoDB-based system to manage a football league with players, teams, matches, and statistics. This demonstrates core concepts and operations in a practical context.

Table Design

Primary Table: FootballLeague

```
Table Name: FootballLeague
Primary Key:
  - Partition Key: EntityType (String)
  - Sort Key: EntityId (String)

Sample Items:
- TEAM#manchester-united, TEAM#manchester-united
- PLAYER#ronaldo, PLAYER#ronaldo
- MATCH#2024-season, MATCH#001
```

Secondary Index: PlayersByTeam

```
GSI Name: PlayersByTeam-GSI
Partition Key: TeamId (String)
Sort Key: PlayerName (String)
```

Demo Steps

Step 1: Create the Football League Table

Console Navigation:

1. AWS Console → Search "DynamoDB" → Open service
2. Tables → Create table
3. Configuration:
   - Table name: `FootballLeague`
   - Partition key: `EntityType` (String)
   - Sort key: `EntityId` (String)
   - Table settings: Default settings
4. Create table → Wait for Active status

Step 2: Create Global Secondary Index

Index Creation:

1. Select FootballLeague table
2. Indexes tab → Create index
3. Configuration:
   - Partition key: `TeamId` (String)

- Sort key: `PlayerName` (String)
- Index name: `PlayersByTeam-GSI`
- Projected attributes: All attributes

4. Create index → Wait for Active status

Step 3: Add Sample Data

Create Team Item:

```json
{
  "EntityType": {"S": "TEAM"},
  "EntityId": {"S": "manchester-united"},
  "TeamName": {"S": "Manchester United"},
  "Stadium": {"S": "Old Trafford"},
  "Founded": {"N": "1878"},
  "League": {"S": "Premier League"}
}
```

Create Player Items:

```json
{
  "EntityType": {"S": "PLAYER"},
  "EntityId": {"S": "ronaldo"},
  "PlayerName": {"S": "Cristiano Ronaldo"},
  "TeamId": {"S": "manchester-united"},
  "Position": {"S": "Forward"},
  "Goals": {"N": "15"},
  "Assists": {"N": "8"}
}
```

Create Match Item:

```json
```

```
{
  "EntityType": {"S": "MATCH"},
  "EntityId": {"S": "2024-001"},
  "HomeTeam": {"S": "Manchester United"},
  "AwayTeam": {"S": "Liverpool"},
  "Date": {"S": "2024-08-15"},
  "HomeScore": {"N": "2"},
  "AwayScore": {"N": "1"},
  "Stadium": {"S": "Old Trafford"}
}
```

Step 4: Query Operations

Query by Entity Type (Teams):

```sql
-- Using PartiQL
SELECT * FROM FootballLeague
WHERE EntityType = 'TEAM'
```

Query Players by Team (Using GSI):

```sql
-- Using PartiQL on GSI
SELECT PlayerName, Position, Goals
FROM FootballLeague."PlayersByTeam-GSI"
WHERE TeamId = 'manchester-united'
ORDER BY PlayerName ASC
```

Query Specific Player:

```sql
SELECT * FROM FootballLeague
WHERE EntityType = 'PLAYER' AND EntityId = 'ronaldo'
```

Step 5: Update Operations

Update Player Statistics:

1. Select player item (EntityType=PLAYER, EntityId=ronaldo)

2. Edit item → Modify Goals from 15 to 18
3. Add new attribute: `LastGameDate = "2024-08-20"`
4. Save changes

Step 6: Scan Operations

Scan for High-Scoring Players:

1. Go to Scan operation
2. Add filter:
   - Attribute: Goals
   - Condition: Greater than 10
3. Run scan → View results

Step 7: Enable Point-in-Time Recovery

Backup Configuration:

1. Select FootballLeague table
2. Backups tab → Edit
3. Enable "Point-in-time recovery"
4. Set retention period: 35 days
5. Save changes

Step 8: Performance Testing

Load Testing Considerations:

- Monitor consumed vs provisioned capacity
- Watch for throttling events
- Observe query vs scan performance differences
- Test GSI query performance

Step 9: Cleanup

Resource Deletion:

1. Tables → Select FootballLeague
2. Actions → Delete table
3. Type "confirm" → Delete

4. Verify deletion complete

Advanced Concepts for Solutions Architect Associate

Consistency Models

Eventually Consistent Reads (Default)

- May not reflect most recent write operations
- Lower latency and higher throughput
- Default for all read operations

Strongly Consistent Reads

- Returns most up-to-date data
- Higher latency than eventually consistent
- Only available for primary key queries (not GSI)

Transaction Support

ACID Transactions

- Atomic, consistent, isolated, and durable transactions across multiple items and tables
- All operations succeed or fail together
- Support for complex business logic
- TransactWriteItems and TransactGetItems APIs

Streams and Change Data Capture

DynamoDB Streams

- Capture item-level change data in near-real time
- Event-driven architectures
- Lambda triggers for automatic processing
- 24-hour retention period

Kinesis Data Streams Integration

- Alternative to DynamoDB Streams
- Longer retention (up to 1 year)

- Integration with Kinesis ecosystem

Exam Focus Areas

Key Exam Topics

Core Concepts Understanding

- NoSQL vs SQL differences and use cases
- Primary key design and importance
- Capacity modes and when to use each
- Consistency models and trade-offs

Design Patterns

- Single table design principles
- Access pattern driven modeling
- Secondary index strategies
- Data denormalization techniques

Performance and Scaling

- Partition key distribution
- Hot partition identification and resolution
- Query vs Scan efficiency
- DAX caching strategies

Integration Scenarios

- Lambda integration patterns
- API Gateway integration
- CloudWatch monitoring setup
- Cross-service architectures

Security and Compliance

- IAM policy design
- Encryption options and requirements
- VPC endpoint configurations

- Compliance requirements

Common Exam Question Patterns

Scenario-Based Questions

- "Which primary key design provides best performance?"
- "How to optimize costs for irregular traffic patterns?"
- "What's the best approach for cross-region replication?"

Troubleshooting Questions

- "Application experiencing throttling - what's the cause?"
- "Query returning old data - what consistency model is being used?"
- "High costs despite low usage - what optimization strategies apply?"

Quick Reference Card

Feature	Use Case	Key Points
Simple Primary Key	User profiles, catalogs	Unique partition key only
Composite Primary Key	Time-series, hierarchical data	Partition + sort key combination
GSI	Alternative query patterns	Different keys, eventually consistent
LSI	Same partition, different sort	Strongly consistent, 10GB limit
On-Demand	Unpredictable traffic	Pay per request, automatic scaling
Provisioned	Predictable traffic	Pre-planned capacity, cost effective
Strong Consistency	Critical data accuracy	Higher latency, primary key only
Eventual Consistency	High throughput needs	Lower latency, default mode
DynamoDB Streams	Event-driven processing	24hr retention, Lambda triggers
Global Tables	Multi-region apps	99.999% availability, active-active

---

Summary

Amazon DynamoDB is a powerful NoSQL database service that excels in scenarios requiring high performance, scalability, and operational simplicity. For the AWS Solutions Architect Associate exam, focus on understanding when DynamoDB is the right choice, how to design efficient data models, and how it integrates with other AWS services to build complete solutions.

Key Takeaways for Exam Success:

- Understand NoSQL design patterns vs relational database design
- Know how to choose appropriate primary key structures
- Recognize scenarios where DynamoDB provides better solutions than RDS
- Understand capacity modes and their cost implications
- Know security best practices and compliance capabilities
- Understand integration patterns with Lambda, API Gateway, and other services

Practice Scenarios:

- Design DynamoDB schemas for various applications
- Calculate capacity requirements for given workloads
- Troubleshoot performance and cost optimization issues
- Architect solutions combining DynamoDB with other AWS services