AWS Project Bootcamp Week 5: DynamoDB & NoSQL - Comprehensive Summary

Executive Summary

This bootcamp session covers advanced DynamoDB implementation for a messaging/direct messaging system similar to Twitter DMs. The session emphasizes **single table design**, **access pattern planning**, and proper data modeling - critical skills for production DynamoDB implementations.

Key Instructor: Kirk (Former AWS DynamoDB Developer Advocate, 11-12 years NoSQL experience)

---

1. PROJECT OVERVIEW: MESSAGING SYSTEM REQUIREMENTS

System Features

- **Direct messaging** between users (similar to Twitter/LinkedIn DMs)
- **Message groups** (conversations) that can theoretically support multiple participants
- **Real-time message display** ordered by creation timestamp
- **Last message preview** in conversation list
- **User perspective handling** (always shows the "other" person's name, not your own)

Visual Data Flow

The session emphasized creating visual diagrams to map:

- What data is available at each UI interaction point
- What data flows to the backend
- What data must be stored vs. computed
- Access patterns for each user action

---

2. DYNAMODB FUNDAMENTALS & CORE CONCEPTS

What is DynamoDB?

DynamoDB is AWS's fully managed NoSQL database service that provides:

- **Sub-millisecond latency** at any scale
- **Automatic scaling** (on-demand or provisioned capacity)
- **Built-in security, backup, and restore**
- **Global tables** for multi-region replication

Key-Value vs Document Store

DynamoDB is classified as **both**:

- **Key-Value Store**: Stores data using partition keys and sort keys
- **Document Store**: Can store complex JSON documents within items
- **Visual appearance**: Looks like a table but must be thought of as key-value pairs

Critical DynamoDB Terminology

Term	Definition	Example
Item	A single data record (like a row in SQL)	One message or one conversation
Attribute	A data field within an item (like a column)	display_name, created_at, message
Partition Key (PK)	Primary identifier that determines data distribution	user_uuid or message_group_uuid
Sort Key (SK)	Optional secondary key for ordering items within a partition	created_at timestamp
Primary Key	Either PK alone (simple) OR PK+SK (composite)	user_uuid + 2023-05-21
GSI	Global Secondary Index - alternate query pattern on same data	Query by different attributes
LSI	Local Secondary Index - alternate sort key, same partition key	Must be created at table creation

---

3. THE DYNAMODB RULES & CONSTRAINTS

Core Limitations (Your Toolkit)

Querying Rules:

1. **Partition Key is ALWAYS required** in queries **-** no exceptions
2. **Sort Key is optional** but enables powerful filtering when used
3. **No joins between tables -** data must be pre-computed
4. **No full table scans in production** (too expensive and slow)
5. **Query operations only return items with matching Partition Key**

Sort Key Operations Available:

- `=` (equals)
- `<`, `<=`, `>`, `>=` (comparisons)
- `BETWEEN` (range)
- `begins_with` (substring matching - very powerful!)

Query Modifiers:

- **Limit**: Restricts number of items returned (server-side)
- **ScanIndexForward**: `false` for descending order, `true` for ascending
- **FilterExpression**: Post-query filtering (still charged for all items read!)

Capacity Units & Costs

Read Capacity Units (RCU):

- 1 RCU = 1 strongly consistent read of 4KB per second
- 1 RCU = 2 eventually consistent reads of 4KB per second

Write Capacity Units (WCU):

- 1 WCU = 1 write of 1KB per second

Cost Optimization Strategy: Keep item sizes small, minimize GSIs, and design queries to read only what's needed.

---

4. ACCESS PATTERNS: THE FOUNDATION OF NOSQL DESIGN

Critical Concept: Access Patterns Rule Everything

Definition: An access pattern is ANY interaction with the database, including:

- Reading/displaying data
- Creating new records
- Updating existing records
- Deleting records

Access Patterns for Messaging System

Pattern A: Show Single Conversation

- **User needs**: List of messages in a conversation, ordered by timestamp
- **Query**: Get all messages where `message_group_uuid` matches
- **Sort**: By `created_at` descending (newest first)

Pattern B: List of Conversations (Most Critical!)

- **User needs**: All conversations for a user, with last message preview
- **Query**: Get all message groups where `user_uuid` matches
- **Sort**: By `last_reply_at` descending
- **Frequency**: Highest usage - optimize for base table

Pattern C: Create Message

- **Action**: Insert new message
- **Complexity**: Low for existing conversations, high for new conversations
- **Requirements**: Must create message + update message group headers

Pattern D: Update Message Group

- **Action**: Update last message preview and timestamp
- **Complexity**: High - requires finding exact items to update
- **Challenge**: Must get current records first, then delete/recreate with new sort key

Pattern Prioritization Strategy

Optimize in this order:

1. **Most frequent operations** (e.g., viewing conversation list)
2. **Operations requiring low latency** (user-facing features)
3. **Operations with high data volume** (prevent expensive scans)

---

5. SINGLE TABLE DESIGN PHILOSOPHY

What is Single Table Design?

Definition: Storing multiple entity types (users, messages, conversations) in one DynamoDB table using clever key design.

Why Use Single Table Design?

Benefits:

- ✅ Reduces operational complexity (one table to manage)
- ✅ Enables atomic transactions across related data
- ✅ Simplifies replication (Global Tables)
- ✅ Related data stays together (better performance)
- ✅ Reduces number of GSIs needed

Drawbacks:

- ❌ Steeper learning curve
- ❌ Harder to modify later
- ❌ Requires extensive upfront planning
- ❌ Can be confusing for new developers

When NOT to Use Single Table Design:

- Data is truly unrelated
- Access patterns are unclear or rapidly changing
- Team lacks DynamoDB expertise
- Quick prototyping phase
- Multiple tables would be simpler without performance loss

Generic Column Naming Convention

Instead of descriptive names, use **generic identifiers**:

- `PK` (Partition Key) instead of `user_id`, `message_group_id`, etc.
- `SK` (Sort Key) instead of `created_at`, `updated_at`, etc.
- `data` for additional flexible attributes

Why?: Allows storing different entity types in the same columns.

---

6. DATA MODELING PROCESS (STEP-BY-STEP)

Step 1: Map All Access Patterns

Start by documenting:

```
  Pattern A: Show single conversation
    - Input: message_group_uuid
    - Output: List of messages ordered by created_at DESC
    - Frequency: Medium (when user clicks into conversation)

  Pattern B: List conversations for user
    - Input: user_uuid
    - Output: Message groups with last_message, last_reply_at
    - Frequency: HIGH (every time user opens messages)
```

Step 2: Visualize Data Flow

Create diagrams showing:

- What data exists at each UI state
- What data is passed in URLs/forms
- What data backend has access to
- Order of operations (create → display → update)

Step 3: Design Primary Keys

Process:

1. **Identify what you have when querying** (for Partition Key)
2. **Determine sort requirements** (for Sort Key)
3. **Use begins_with for flexible filtering** (prefix patterns)
4. **Validate against ALL access patterns**

Step 4: Test Against Each Pattern

For each access pattern, ask:

- Can I query this with just PK?
- Do I need SK filtering?
- Is the sort order correct?
- Do I have all required data in the result?

Step 5: Add GSIs for Remaining Patterns

If base table can't support a pattern:

- Create GSI with different PK/SK
- Project only needed attributes (cost savings)
- Remember: GSIs are eventually consistent

---

7. ACTUAL IMPLEMENTATION DESIGN

Messages Table Structure

Message Items

```
PK: message_group_uuid         (e.g., "a1b2c3d4-uuid")
SK: MSG#2023-05-21T10:30:00Z   (prefixed timestamp)
display_name: "Andrew Brown"
handle: "@andrewbrown"
message: "Hey, how are you?"
user_uuid: "user123-uuid"
```

Why MSG# prefix?: Enables `begins_with("MSG#")` filtering to get only messages, not message groups.

Message Group Items (in same table!)

```
PK: grp#user_uuid                   (e.g., "grp#user123-uuid")
SK: last_reply_at                   (e.g., "2023-05-21T10:30:00Z")
message_group_uuid: "a1b2c3d4-uuid"
other_user_display_name: "Andrew Bayko"
other_user_handle: "@bayko"
last_message: "Hey, how are you?"    (AKA of message field)
```

Critical Design Decision: Store TWO message group items per conversation:

- One from User A's perspective (shows User B's name)
- One from User B's perspective (shows User A's name)

Key Design Patterns Used

1. Prefixing for Entity Differentiation

```
grp#user_uuid     → Message Group entity
MSG#timestamp     → Message entity
usr#user_id       → User entity (if needed)
```

2. Concatenated Sort Keys

```
SK: MSG#2023-05-21T10:30:00Z
      ↑   ↑
    Type  Sortable timestamp
```

Enables:

- `begins_with("MSG#")` - Get all messages
- `begins_with("MSG#2023")` - Get messages from 2023
- `begins_with("MSG#2023-05")` - Get messages from May 2023

3. Attribute Overloading (AKA pattern)

Reusing attributes for different purposes:

- `display_name` in messages = sender name
- `display_name` in message groups = other user's name (AKA other_user_display_name)
- `SK` can be `created_at` OR `last_reply_at` depending on entity type

---

8. HANDLING UPDATES (The Trickiest Part)

Challenge: Updating Sort Keys

Problem: When a new message is sent, must update `last_reply_at` in message group. But `last_reply_at` IS the sort key!

Solution Options:

Option 1: Delete & Recreate (Recommended)

python

```
# Transaction to ensure atomic operation
transaction = [
  {
    "Delete": {
      "PK": "grp#user123",
      "SK": "2023-05-20T15:00:00Z"  # Old timestamp
    }
  },
  {
    "Put": {
      "PK": "grp#user123",
      "SK": "2023-05-21T10:30:00Z",  # New timestamp
      # ... all other attributes
    }
  }
]
```

Option 2: Query Then Update (Two Operations)

1. Query to find the current item
2. Delete old item + Create new item with updated SK

Kirk's Guidance: Delete + Recreate is fine if it fits your access patterns!

Update Pattern Requirements

To update message groups, you need:

1. **user_uuid** (to find the message group items)
2. **message_group_uuid** (to uniquely identify which conversation)
3. **Both users' perspectives** (must update 2 items per conversation)

This led to requiring a **GSI** for efficient lookups by message_group_uuid.

---

9. GLOBAL SECONDARY INDEX (GSI) DESIGN

When GSIs Are Needed

Create a GSI when:

- Base table can't support an access pattern
- Need to query by different attributes
- Need different sort order than base table

GSI for Message Groups

```
GSI Name: message-group-uuid-index
PK: message_group_uuid
SK: user_uuid (or timestamp, depending on pattern)
Projected Attributes: ALL or KEYS_ONLY (minimize cost)
```

Use Case: Find all message group items for a specific conversation to update them.

GSI vs LSI Decision Matrix

Feature	GSI	LSI
Partition Key	Can be different	Must be same as base
Sort Key	Can be different	Must be different
Consistency	Eventually consistent	Strongly consistent
When to create	Anytime	Table creation only
Modification	Can delete/recreate	Permanent
Use Case	New query patterns	Alternative sort on same partition

---

10. ADVANCED CONCEPTS & BEST PRACTICES

Storing JSON Documents

Kirk's Tip: If you're not indexing on specific fields, store as JSON string in a single attribute.

Benefits:

- Faster read/write performance
- Cheaper (fewer attributes to scan)
- Simpler schema

Example:

json

```json
{
  "PK": "user#123",
  "SK": "profile",
  "data": "{\"address\":{\"street\":\"123 Main\",\"city\":\"Toronto\"},\"preferences\":{...}}"
}
```

PartiQL for DynamoDB

What is PartiQL?: SQL-compatible query language for DynamoDB.

Example Query:

sql

```sql
SELECT * FROM MessagesTable
WHERE PK = 'grp#user123'
  AND begins_with(SK, 'MSG#2023')
ORDER BY SK DESC
LIMIT 20
```

Use Case: Easier for developers familiar with SQL, but learn native API for production.

Data Duplication is OK!

In NoSQL, repeating data is acceptable when:

- It supports access patterns
- It improves performance
- Storage cost is negligible compared to compute

Example: Storing `last_message` in both message item AND message group item.

Partition Key Distribution

Critical Rule: Design partition keys to distribute data evenly.

Bad Example:

```
PK: country (90% of users in one country)
```

Good Example:

```
PK: user_uuid (evenly distributed)
```

The 4KB Rule

- Items larger than 4KB consume multiple RCUs
- Keep items under 4KB when possible
- Use JSON documents for nested data
- Project only needed attributes in GSIs

---

11. SCANNING vs QUERYING

Query (Preferred)

- **Requires**: Partition Key
- **Optional**: Sort Key conditions
- **Returns**: Only items in specified partition
- **Cost**: Based on data read (predictable)
- **Speed**: Milliseconds

Scan (Avoid in Production)

- **Requires**: Nothing (reads entire table)
- **Returns**: All items, then filters
- **Cost**: Charged for ENTIRE table read
- **Speed**: Seconds to minutes
- **Use Cases**:
  - Infrequent reporting
  - When creating GSI costs more than occasional scan
  - Batch processing

Kirk's Example: If a GSI would be used rarely, a scan might be cheaper.

---

12. PRACTICAL IMPLEMENTATION LESSONS

Common Mistakes (Avoid These!)

1. **Starting without access patterns** → Results in complete redesign
2. **Using UUIDs as partition keys without reason** → Can't query efficiently
3. **Not planning for updates** → Discover you can't modify sort keys easily
4. **Forgetting about GSI costs** → Surprised by AWS bill
5. **Not testing all patterns** → Missing critical access requirements
6. **Assuming one pattern fits all** → Over-engineering or under-engineering

When to Use Multiple Tables Instead

- **Unrelated data**: User profiles vs. application logs
- **Different access patterns**: Hot data vs. cold data
- **Compliance requirements**: Separate sensitive data
- **Team structure**: Different teams own different domains
- **AppSync optimization**: Sometimes simpler with separate tables

DynamoDB is HARD

Key Takeaway from session: Even experts struggle with design!

- Andrew and Bayko worked for hours and weren't confident
- Required multiple iterations and expert validation
- Visual diagrams were essential
- Documentation is sparse on real-world patterns

---

13. WEEK 5 ADDITIONAL TOPICS

Serverless Caching with Momento

The bootcamp also covers implementing **Momento** (serverless cache) with DynamoDB:

- **Purpose**: Reduce DynamoDB read costs
- **Implementation**: Cache frequently accessed message groups
- **Pattern**: Cache-aside pattern
- **Benefit**: Sub-millisecond responses without DynamoDB charges

---

14. KEY TAKEAWAYS & WISDOM

Kirk's Golden Rules

1. **"Access patterns rule everything"** - Know what you're building first
2. **"Pre-compute your answers"** - Don't ask questions, fetch answers
3. **"Storage is cheap, compute is expensive"** - Duplicate data when needed
4. **"Model for your application, not the database"** - Opposite of SQL
5. **"80-90% in base table, 10-20% in GSIs"** - Optimization target

The Mindset Shift

SQL Thinking:

```
"What tables do I need to store this data properly?"
```

NoSQL Thinking:

```
"What data does my app need, when does it need it, and how fast?"
```

Planning Checklist

Before implementing DynamoDB:

- [ ] Document ALL access patterns (read, write, update, delete)
- [ ] Visualize data flow through application
- [ ] Identify what data is available at each query point
- [ ] Design partition keys for even distribution
- [ ] Test design against every access pattern
- [ ] Consider frequency of each operation
- [ ] Plan for updates (especially to sort keys)
- [ ] Minimize GSIs (but use when needed)
- [ ] Document your design choices (for future developers!)

---

15. RESOURCES & NEXT STEPS

Tools Mentioned

- **NoSQL Workbench**: Visual design tool for DynamoDB
- **AWS Documentation**: Official PartiQL and API references
- **Alex DeBrie's Book**: "The DynamoDB Book" (highly recommended)
- **Rick Houlihan's Talks**: Advanced single-table design patterns

Implementation Notes

The actual code implementation will cover:

- Creating the DynamoDB table
- Implementing access patterns with boto3 (AWS SDK)
- Transaction handling for atomic updates
- Pagination for large result sets
- Error handling and retries
- Capacity unit monitoring
- Cost optimization techniques

Project Context

This messaging system is part of a larger "Cruddur" application (Twitter clone) being built throughout the bootcamp, integrating with:

- AWS Cognito (authentication)
- PostgreSQL (user data)
- AppSync (GraphQL API)
- Lambda functions (serverless compute)

FINAL THOUGHTS

DynamoDB mastery requires:

- Extensive upfront planning (invest time in design)
- Deep understanding of access patterns

- Comfort with data duplication
- Acceptance that it's fundamentally different from SQL
- Willingness to iterate and refine

The payoff:

- Consistent sub-millisecond performance at any scale
- Predictable costs
- Zero server management
- Global replication capabilities

Remember: Single table design isn't mandatory! Use it when it makes sense for your specific use case, and don't be afraid to use multiple tables if that's simpler and meets your requirements.

This summary represents 2 hours of intensive DynamoDB design discussion from Week 5 of the AWS Project Bootcamp. The session emphasized that proper planning and understanding access patterns is 90% of successful DynamoDB implementation.