NoSQL Database Design Patterns: DynamoDB Single Table Design vs. Multi-Collection Approaches

Executive Summary

This report examines the architectural differences between DynamoDB's single table design pattern and the multi-collection/multi-table approaches used by other NoSQL databases like MongoDB and Cassandra. While DynamoDB strongly favors single table design due to its unique constraints, other NoSQL databases provide native support for relationships and complex queries that eliminate the need for such workarounds.

Introduction

Single table design is a data modeling pattern where all entity types are stored within a single DynamoDB table, using composite keys to organize and retrieve related data. This approach emerged as a response to DynamoDB's specific architectural limitations rather than being a universal NoSQL best practice.[1] [2]

DynamoDB's Architectural Constraints

Core Limitations Driving Single Table Design

Limited Query Capabilities

- No JOIN operations between tables
- Only key-based queries (partition key + optional sort key)
- No cross-table transactions without complex workarounds
- Limited secondary indexing (maximum 20 GSIs per table)[3]

400 KB Item Size Limit
DynamoDB restricts each item to a maximum of 400 KB, preventing the storage of rich nested documents that other NoSQL databases support. This constraint forces data flattening and denormalization strategies.[4]

GSI Operational Costs
Each Global Secondary Index doubles storage costs and requires separate provisioned capacity, making multi-table designs with extensive indexing economically prohibitive.[3]

Alternative NoSQL Database Approaches

MongoDB: Flexible Document Modeling

Embedded Document Strategy

MongoDB supports documents up to 16 MB, enabling rich nested structures:[5]

```
{
  "_id": "order123",
  "customer": {
    "name": "John Doe",
    "address": { "street": "123 Main St", "city": "Boston" }
  },
  "items": [
    { "product": "Laptop", "quantity": 2, "reviews": [...] }
  ]
}
```

Reference-Based Relationships

For larger datasets, MongoDB uses references between collections with application-level joins:[6]

```
// Two-step process
const post = db.posts.findOne({"_id": ObjectId("...")});
const author = db.users.findOne({"_id": post.author_id});
```

Advanced Query Capabilities

MongoDB provides aggregation pipelines with `$lookup` operations for complex cross-collection queries.[7]

Cassandra: Query-Driven Denormalization

Multiple Specialized Tables

Cassandra creates different tables optimized for specific query patterns:[8] [9]

```
-- Table for user's order history
CREATE TABLE orders_by_user (
    user_id UUID,
    order_date timestamp,
    order_id UUID,
    total decimal,
    PRIMARY KEY (user_id, order_date)
);

-- Table for order details
CREATE TABLE order_details (
    order_id UUID PRIMARY KEY,
    customer_info text,
    items list<frozen<item>>
);
```

Clarification: Choice vs. Requirement

Important Distinction: DynamoDB does not technically *force* single table design. Multi-table architectures are possible using:

- Primary keys from one table as foreign keys in another
- GSIs to support foreign key queries
- Application-level referential integrity

However, this approach incurs significant penalties:

- **No native foreign key constraints**[10]
- **GSI proliferation and associated costs**
- **Complex transaction management**
- **Multiple queries for related data retrieval**

These limitations make single table design the **recommended pattern** rather than an absolute requirement.

The 400 KB Constraint Impact

The 400 KB item limit significantly influences design decisions:

Forces Data Flattening
Unlike MongoDB's 16 MB documents, DynamoDB cannot store complex nested objects, requiring:

- Flattened attribute naming (`customer_name` vs. nested `customer.name`)
- Strategic data placement across multiple items
- Careful consideration of what to include in each item[4]

Workarounds for Large Objects
Three primary strategies for exceeding the 400 KB limit:

1. **S3 Storage with DynamoDB Pointers**: Store large objects in S3, keep metadata in DynamoDB[4]
2. **Item Chunking**: Split large objects across multiple items using sort keys[11]
3. **Compression**: Use GZIP or similar algorithms to reduce item size[12]

Comparative Analysis: E-Commerce Example

DynamoDB Single Table Approach

```
// All entities in one table with composite keys
{
  "PK": "USER#john123",
  "SK": "PROFILE",
  "name": "John Doe",
```

```
  "email": "john@example.com"
}
{
  "PK": "USER#john123",
  "SK": "ORDER#2024-001",
  "order_id": "2024-001",
  "total": 299.99
}
```

MongoDB Multi-Collection Approach

```
// Separate collections with natural relationships
// Users collection
{ "_id": "john123", "name": "John Doe", "email": "john@example.com" }

// Orders collection
{ "_id": "2024-001", "user_id": "john123", "total": 299.99 }

// Complex aggregation queries supported natively
db.users.aggregate([
  { $match: { "_id": "john123" }},
  { $lookup: { from: "orders", localField: "_id", foreignField: "user_id", as: "orders" }
])
```

Cassandra Specialized Tables

```
-- Multiple optimized tables
CREATE TABLE users (user_id UUID PRIMARY KEY, name text, email text);
CREATE TABLE orders_by_user (
    user_id UUID,
    order_date timestamp,
    order_id UUID,
    PRIMARY KEY (user_id, order_date)
);
```

Decision Framework

When Single Table Design is Optimal

- **Clear, stable access patterns**
- **Performance-critical applications** requiring minimal latency
- **Cost-sensitive environments** where GSI proliferation is prohibitive
- **Simple to moderate complexity** data relationships

When Multi-Table Approaches Excel

- **Evolving applications** with unclear access patterns
- **Complex many-to-many relationships**
- **Rich nested data structures**
- **Strong consistency requirements** across entities
- **Analytics-heavy workloads**

Key Takeaways

1. **Single table design is uniquely advantageous for DynamoDB** due to its architectural constraints, not a universal NoSQL best practice.
2. **Other NoSQL databases provide sophisticated query engines** that eliminate the need for complex single-table workarounds.
3. **The 400 KB item limit compounds DynamoDB's design constraints**, preventing rich document structures possible in other systems.
4. **Design choice depends on database capabilities**: DynamoDB's limitations make single table design often optimal, while other NoSQL databases offer more flexible alternatives.
5. **Understanding the trade-offs is crucial**: Performance benefits versus operational complexity, cost implications versus query flexibility.

Conclusion

While DynamoDB's single table design pattern has proven effective for high-scale, performance-critical applications, it represents a database-specific solution to architectural constraints rather than a universal NoSQL principle. Other NoSQL databases like MongoDB and Cassandra provide native capabilities for handling relationships and complex queries that reduce the need for such design patterns. The choice between approaches should be based on specific application requirements, data complexity, and the capabilities of the chosen database system.

This report synthesizes analysis of DynamoDB documentation, NoSQL design patterns, and comparative database capabilities as of September 2025.

⁂

1. https://aws.amazon.com/blogs/compute/creating-a-single-table-design-with-amazon-dynamodb/
2. https://alexdebrie.com/posts/dynamodb-single-table/
3. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-indexes-general.html
4. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-use-s3-too.html
5. https://www.geeksforgeeks.org/mongodb-data-modelling/
6. https://www.mongodb.com/docs/manual/applications/data-models-relationships/
7. https://www.mongodb.com/resources/basics/databases/data-modeling
8. https://www.datastax.com/blog/basic-rules-cassandra-data-modeling

9. https://www.instaclustr.com/blog/cassandra-data-modeling/
10. https://advancedweb.hu/foreign-key-constraints-in-dynamodb/
11. https://aws.amazon.com/blogs/database/large-object-storage-strategies-for-amazon-dynamodb/
12. https://reintech.io/blog/handling-large-items-attributes-dynamodb
13. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html
14. https://www.reddit.com/r/aws/comments/xu64gd/dynamodb_single_table_deign_simple_guide_to/
15. https://www.simform.com/blog/dynamodb-best-practices/
16. https://www.linkedin.com/pulse/aws-dynamodb-part-2-design-patterns-best-practices-rasel
17. https://www.youtube.com/watch?v=lWCch8GEK4E
18. https://madhead.me/posts/std/
19. https://emshea.com/post/part-1-dynamodb-single-table-design