Data Modeling in MongoDB

1. Introduction

Data modeling in MongoDB is fundamentally different from relational databases.

In relational databases:

- Data is normalized
- Foreign keys enforce relationships
- Joins are common

In MongoDB:

- Data is modeled based on access patterns
- Relationships are handled using:
  - Embedding
  - Referencing
- No database-enforced foreign keys

MongoDB modeling is query-driven, not normalization-driven.

2. Core Differences: SQL vs MongoDB

Concept	SQL	MongoDB
Schema	Fixed	Flexible
Foreign Keys	Enforced	Not enforced
Joins	Native	Manual / populate
Normalization	Strict	Optional
Optimization Goal	Data integrity	Performance & scalability

3. Relationship Cardinality in MongoDB

MongoDB supports all conceptual relationship types:

- One-to-One (1–1)
- One-to-Many (1–N)
- Many-to-Many (N–N)

However, implementation differs because MongoDB does not enforce foreign keys.

4. Factors to Consider Before Choosing Design (Embedded or Referential)

Before deciding whether to embed or reference, consider the following factors:

4.1 Access Pattern

- Is the child data always fetched with the parent?
- Is the child queried independently?

4.2 Data Size

- Will the child array grow indefinitely?
- Can the document exceed size limits (16MB limit in MongoDB)?

4.3 Update Frequency

- Is the child updated frequently?
- Will updates affect large embedded arrays?

4.4 Lifecycle Independence

- Does the child exist independently?
- Can the child exist without the parent?

4.5 Sharing Across Parents

- Can multiple parents reference the same child document?
- If yes, embedding may cause duplication problems.

4.6 Query Complexity

- Do you need filtering, sorting, or indexing directly on child documents?
- If yes, referencing is usually better.

4.7 Transactional Requirements

- Do you need atomic updates across parent and child?
- Embedding allows atomic updates within a document.

These factors guide whether embedding or referencing is the correct choice.

5. Strategy 1: Embedding (Denormalization)

What is Embedding?

Embedding means storing related data inside the same document.

Example:

```
{
  name: "John",
  addresses: [
    {
      city: "Chennai",
      pincode: "600001"
    }
  ]
}
```

Addresses are embedded inside User.

When to Use Embedding

Use embedding when:

- Child data belongs strictly to one parent
- Data is always fetched together
- Child data size is small
- No independent lifecycle
- Limited growth

One-to-One (1–1) with Embedding

Example:

```
User {
  name: "John",
  profile: {
    bio: "Developer",
    age: 25
  }
}
```

Use when:

- Profile only belongs to user
- Always retrieved together

One-to-Many (1–N) with Embedding

Example:

```
Customer {
  addresses: [
    {...},
    {...}
  ]
}
```

Use when:

- Child documents are small
- No need to query addresses separately

Advantages of Embedding

- Faster reads
- No joins required
- Atomic updates
- Simple structure

Disadvantages of Embedding

- Document size can grow
- Hard to update deeply nested large arrays
- Not suitable for large-scale child collections

6. Strategy 2: Referencing

What is Referencing?

Referencing means storing ObjectId of another document.

Example:

```
Product {
  sellerId: ObjectId("...")
}
```

This creates a logical relationship.

One-to-One (1–1) with Referencing

Example:

```
Customer {
  userId: ObjectId("...")
}
```

Use when:

- Both documents have independent lifecycle
- Separation of concerns is needed

One-to-Many (1–N) with Referencing

Example:

```
Product {
  sellerId: ObjectId("...")
}
```

Use when:

- Many products per seller
- Products queried independently
- Data large or growing

Many-to-Many (N–N) with Referencing

Example:

```
Student {
  courseIds: [ObjectId("...")]
}

Course {
  studentIds: [ObjectId("...")]
}
```

Or use a separate collection if the relationship contains metadata.

Advantages of Referencing

- Avoids large documents
- Scalable
- Independent querying
- Clean separation

Disadvantages of Referencing

- Requires multiple queries
- No foreign key enforcement
- Application must ensure integrity

7. MongoDB and Data Integrity

MongoDB does not enforce foreign keys.

Integrity must be handled at:

- Application layer
- Service layer
- Business logic

Example:

Before creating Product:

- Verify seller exists

Before creating Order:

- Verify product exists
- Verify inventory exists

Integrity is programmatic, not database-enforced.

8. Populate in Mongoose

Mongoose supports:

.populate("sellerId")

This performs a logical join in application memory.

Important:

- Populate is not a true database join
- Overuse can impact performance
- Should be used carefully

9. Denormalization in MongoDB

MongoDB allows duplication intentionally.

Example:

In Order:

- productName
- priceAtPurchase

Even though Product exists separately.

Why?

- Orders must remain historically accurate
- Avoid data inconsistency if product changes later

Denormalization is a design decision for performance and correctness.

10. Golden Rules of MongoDB Data Modeling

1. Model based on query patterns.
2. Embed for containment.
3. Reference for independence.
4. Denormalize for performance.
5. Keep documents within size limits.
6. Avoid unnecessary deep nesting.
7. Think in terms of read optimization.

11. Summary

MongoDB data modeling is:

- Flexible
- Query-driven
- Performance-oriented
- Application-managed for integrity

Unlike relational databases, MongoDB modeling requires architectural thinking, not just table design.