MongoDB Schema Design for Sakila Database

From Relational to Document-Oriented: A Moderate Denormalization Approach

Diogo Ribeiro
ESMAD - Instituto Politécnico do Porto
dfr@esmad.ipp.pt

December 17, 2025

Abstract

This document presents a comprehensive analysis of the schema design decisions for transforming the Sakila relational database into a MongoDB document-oriented structure. We explore the rationale behind choosing a moderately denormalized approach with four collections, balancing the trade-offs between read performance, write efficiency, and data consistency. The design principles presented here serve as a practical example for NoSQL database courses, demonstrating real-world application of MongoDB best practices.

1 Introduction

The transformation from a relational database schema to a document-oriented NoSQL schema requires fundamental reconsideration of data organization principles. Unlike the normalized approach of relational databases, MongoDB encourages denormalization to optimize read performance and reduce the need for joins. However, excessive denormalization can lead to data redundancy, update anomalies, and storage inefficiency.

Our approach for the Sakila database transformation adopts what we term "Goldilocks Denormalization" — not too normalized, not too denormalized, but just right for the specific use case of a DVD rental system.

2 Schema Design Overview

2.1 The Four-Collection Architecture

Our MongoDB schema consists of four primary collections, reduced from the original 16 tables in the relational model:

1. **films**: Movie catalog with embedded actors, categories, and language information
2. **customers**: Customer records with embedded address hierarchy
3. **stores**: Physical store locations with embedded manager and address details
4. **rentals**: Transaction records with embedded payments and references to other collections

Table 1: Collection Structure Summary

Collection	Document Count	Embedded Data	References
films	∼1,000	actors[], categories[], spoken_language	None
customers	∼600	address.city.country	store_id
stores	2	address, manager	None
rentals	∼16,000	payments[]	customer_id, film_id, staff_id, store_id

3 Design Rationale

3.1 Access Pattern Analysis

The schema design is fundamentally driven by query patterns observed in typical DVD rental operations. We analyzed the frequency and importance of various queries to inform our embedding decisions.

Table 2: Query Pattern Analysis and Design Decisions

Query Pattern	Frequency	Priority	Design Solution
Find films by actor	Very High	Critical	Actors embedded in films
Find films by category	Very High	Critical	Categories embedded in films
Retrieve customer details	High	Critical	Address embedded in customers
List customer rentals	High	Critical	Index on rentals.customer_id
Get film for rental	High	Important	Reference with $lookup when needed
Calculate customer payment	High	Critical	Payments embedded in rentals
Find rentals by date	Medium	Important	Index on rentals.rental_date
Update film information	Low	Normal	Single document update in films

3.2 Relationship Cardinality Analysis

The decision to embed or reference data is primarily influenced by the cardinality and volatility of relationships:

3.2.1 Embedding Criteria

We embed data when the following conditions are met:

- **Cardinality**: One-to-one or one-to-few relationships (typically < 100 items)
- **Volatility**: Data that rarely changes after creation
- **Access Pattern**: Data always or frequently accessed together
- **Size Constraint**: Embedded data doesn't risk exceeding MongoDB's 16MB document limit

3.2.2 Reference Criteria

We use references when:

- **Cardinality**: One-to-many or many-to-many relationships
- **Volatility**: Frequently updated data
- **Access Pattern**: Data sometimes accessed independently
- **Size Constraint**: Embedding would create excessively large documents

4 Detailed Collection Design

4.1 Films Collection

The films collection represents the movie catalog with full denormalization of related entities that have stable relationships.

```
{
  film_id: 1,
  title: "ACADEMY DINOSAUR",
  description: "A Epic Drama of a Feminist...",
  release_year: 2006,
  rental_duration: 6,
  rental_rate: 0.99,
  length: 86,
  replacement_cost: 20.99,
  rating: "PG",
  special_features: ["Deleted Scenes", "Behind the Scenes"],
  spoken_language: {
    language_id: 1,
    name: "English"
  },
  original_language: null,
  actors: [
    {
      actor_id: 1,
      first_name: "PENELOPE",
      last_name: "GUINESS"
    },
    // ... more actors (typically 5-10)
  ],
  categories: [
    {
      category_id: 6,
      name: "Documentary"
    }
    // ... typically 1-3 categories
  ]
}
```

Listing 1: Films Collection Document Structure

Design Justification:

- **Embedded Actors**: Films have a bounded set of actors (average 5-10), and the cast doesn't change after film creation
- **Embedded Categories**: Films belong to 1-3 categories that remain constant
- **Embedded Language**: Language is an inherent property of the film
- **Document Size**: Average document size ≈ 3-5KB, well below the 16MB limit

4.2 Customers Collection

The customers collection embeds the complete address hierarchy, eliminating three joins from the relational model.

```
{
  customer_id: 1,
  store_id: 1,
  first_name: "MARY",
  last_name: "SMITH",
  email: "MARY.SMITH@sakilacustomer.org",
  active: true,
  create_date: ISODate("2006-02-14T00:00:00Z"),
  address: {
    address_id: 5,
    address: "1913 Hanoi Way",
    address2: null,
    district: "Nagasaki",
    postal_code: "35200",
    phone: "28303384290",
    city: {
      city_id: 463,
      city: "Sasebo"
    },
    country: {
      country_id: 50,
      country: "Japan"
    }
  }
}
```

Listing 2: Customers Collection Document Structure

Design Justification:

- **Embedded Address**: One-to-one relationship, always accessed together
- **Nested City/Country**: Provides complete location context without joins
- **Reference to Store**: Many customers per store, store details not always needed

4.3 Stores Collection

The stores collection represents physical locations with embedded manager information.

```
{
  store_id: 1,
  address: {
    address_id: 1,
    address: "47 MySakila Drive",
    district: "Alberta",
    postal_code: "",
    phone: "",
    city: {
      city_id: 300,
      city: "Lethbridge"
    },
    country: {
      country_id: 20,
      country: "Canada"
    }
  },
  manager: {
    staff_id: 1,
    first_name: "Mike",
    last_name: "Hillyer",
    email: "Mike.Hillyer@sakilastaff.com",
    username: "Mike",
    active: true,
    address: {
      // ... manager's personal address
    }
  }
}
```

Listing 3: Stores Collection Document Structure

Design Justification:

- **Embedded Manager**: One-to-one relationship at any given time
- **Embedded Address**: Store location is fundamental to the store entity
- **Small Collection**: Only 2 documents, making normalization unnecessary

4.4 Rentals Collection

The rentals collection uses a hybrid approach: embedding for tightly coupled data (payments) and references for entities that exist independently.

```
{
  rental_id: 1,
  rental_date: ISODate("2005-05-24T22:53:30Z"),
  return_date: ISODate("2005-05-26T22:04:30Z"),
  customer_id: 130,      // Reference
  film_id: 334,          // Reference
  inventory_id: 1525,    // Reference
  staff_id: 1,           // Reference
  store_id: 1,           // Reference
  payments: [            // Embedded
    {
      payment_id: 1,
      amount: 4.99,
      payment_date: ISODate("2005-05-25T11:30:37Z")
    }
    // Usually 1-2 payments per rental
  ]
}
```

Listing 4: Rentals Collection Document Structure

Design Justification:

- **Referenced Entities**: Prevents massive duplication of film/customer data
- **Embedded Payments**: One-to-few relationship, always accessed with rental
- **Lean Documents**: Average size < 500 bytes, enabling efficient queries

5 Trade-off Analysis

5.1 Storage Efficiency

Table 3: Storage Comparison: Full Denormalization vs. Our Approach

Metric	Full Denormalization	Our Approach	Savings
Document Size	∼15KB per rental	∼500B per rental	96.7%
Total Storage	∼240MB	∼25MB	89.6%
Film Duplication	16,000 copies	1,000 copies	93.8%
Customer Duplication	Multiple per rental	Once per customer	∼95%

5.2 Update Complexity

Consider the impact of common update operations under different denormalization strategies:

Table 4: Update Operation Complexity

Update Scenario	Full Embed	Our Design	Full Normalize
Change actor name	1000s of updates	1 update	1 update
Update customer address	100s of updates	1 update	1 update
Modify film rating	1000s of updates	1 update	1 update
Add payment to rental	1 update	1 update	1 insert

5.3 Query Performance

Table 5: Query Performance Characteristics

Query Type	Joins Required	Index Usage	Performance
Find films by actor	0	actors.actor_id	Excellent
Get customer with address	0	customer_id	Excellent
List rentals by date	0	rental_date	Excellent
Get rental with film details	1 ($lookup)	film_id	Good
Customer rental history	1 ($lookup)	customer_id	Good

6 MongoDB Best Practices Applied

6.1 Principle 1: Data Accessed Together, Stored Together

Our design embeds data that is consistently accessed as a unit:

- Films are always displayed with their cast and categories
- Customers are always shown with their complete address
- Payments are always retrieved with their associated rental

6.2 Principle 2: Avoid Unbounded Arrays

All embedded arrays in our schema have natural bounds:

- `films.actors[]`: Typically 5-10 actors (bounded by production constraints)

- `films.categories[]`: Typically 1-3 categories (bounded by classification system)
- `rentals.payments[]`: Typically 1-2 payments (bounded by business logic)

6.3 Principle 3: Consider Document Size Limits

MongoDB enforces a 16MB document size limit. Our largest documents are in the films collection:

$$\text{Max Film Document Size} = \text{Base Fields} + (\text{Actors} \times \text{Actor Size}) + \text{Categories} \tag{1}$$

$$\approx 1\text{KB} + (20 \times 150\text{B}) + 200\text{B} \approx 4.2\text{KB} \ll 16\text{MB} \tag{2}$$

6.4 Principle 4: Design for Common Queries

Following the Pareto principle, we optimize for the 80% of queries that generate the majority of system load:

```
// Fast: No join required
db.films.find({"categories.name": "Action"})

// Fast: Complete customer info in one query
db.customers.findOne({"email": "john@example.com"})

// Acceptable: Single join for less common query
db.rentals.aggregate([
  {$match: {rental_id: 123}},
  {$lookup: {
    from: "films",
    localField: "film_id",
    foreignField: "film_id",
    as: "film"
  }}
])
```

Listing 5: Optimized Query Examples

7 Implementation Considerations

7.1 Index Strategy

Proper indexing is crucial for maintaining query performance in our denormalized schema:

```
// Films Collection
db.films.createIndex({"film_id": 1})            // Primary lookup
db.films.createIndex({"title": "text"})         // Text search
db.films.createIndex({"categories.name": 1})    // Category filtering
db.films.createIndex({"actors.actor_id": 1})    // Actor lookup
db.films.createIndex({"rating": 1})             // Rating filters

// Customers Collection
db.customers.createIndex({"customer_id": 1})    // Primary lookup
db.customers.createIndex({"email": 1})          // Email lookup
db.customers.createIndex({"address.city.city": 1})    // Geographic queries
db.customers.createIndex({"address.country.country": 1})

// Rentals Collection
db.rentals.createIndex({"rental_id": 1})        // Primary lookup
db.rentals.createIndex({"rental_date": -1})     // Date range queries
db.rentals.createIndex({"customer_id": 1})      // Customer history
db.rentals.createIndex({"film_id": 1})          // Film rental history
db.rentals.createIndex({                        // Compound for reports
  "customer_id": 1,
  "rental_date": -1
})
```

Listing 6: Recommended Indexes

7.2 Migration Strategy

The transformation from relational to document-oriented requires careful data migration:

1. **Extract Phase**: Load all MySQL tables into memory-indexed dictionaries
2. **Transform Phase**: Build denormalized documents using nested lookups
3. **Load Phase**: Batch insert documents (recommended batch size: 1000)
4. **Verify Phase**: Validate document counts and sample data integrity
5. **Index Phase**: Create indexes after data load for optimal performance

7.3 Consistency Maintenance

While MongoDB doesn't enforce foreign key constraints, our design minimizes consistency risks:

- **Immutable References**: Film and customer IDs in rentals never change
- **Embedded Redundancy**: Limited to slowly-changing data (actor names, addresses)
- **Application-Level Validation**: Enforce referential integrity in application layer

8 Performance Metrics

8.1 Query Response Times

Based on typical MongoDB deployments with our schema:

Table 6: Expected Query Performance

Operation	Response Time	Documents Scanned
Find film by ID	< 1ms	1
Films by category (indexed)	< 5ms	~50-200
Customer with address	< 1ms	1
Recent rentals (date range)	< 10ms	~100-500
Rental with film details ($lookup)	< 5ms	2
Full customer history with films	< 50ms	~50-100

8.2 Scalability Considerations

Our schema scales well for typical DVD rental operations:

- **Horizontal Scaling**: Rentals can be sharded by rental_date or customer_id
- **Read Replicas**: Film catalog queries can be distributed across replicas
- **Cache Efficiency**: Small, consistent document sizes improve cache hit rates

9 Comparison with Alternative Approaches

9.1 Alternative 1: Fully Normalized (16 Collections)

Pros:

- Minimal data redundancy
- Simple updates
- Familiar for SQL developers

Cons:

- Excessive $lookup operations
- Poor read performance
- Loses MongoDB's document model advantages

9.2 Alternative 2: Single Collection (Complete Denormalization)

Pros:

- No joins ever required
- Simplest possible queries
- Maximum read performance

Cons:

- Massive data duplication (10-50x storage)
- Update anomalies and complexity
- Risk of document size limits
- Difficult to maintain consistency

9.3 Alternative 3: Two Collections (Films + Transactions)

Pros:

- Simpler than our approach
- Clear separation of catalog and transactions

Cons:

- Customer data duplicated in each rental
- Store information not well organized
- Less flexible for customer-centric queries

10 Pedagogical Value

This schema design serves as an excellent teaching example for NoSQL courses because it demonstrates:

10.1 Core NoSQL Concepts

1. **Denormalization Trade-offs**: Balancing read/write performance with storage
2. **Document Modeling**: Choosing embedding vs. referencing
3. **Query-Driven Design**: Schema shaped by access patterns
4. **Polyglot Persistence**: When NoSQL is appropriate vs. RDBMS

10.2 Practical Challenges

1. **Reserved Field Names**: The "language" field conflict
2. **Type Conversions**: Handling MySQL SET types, decimals, dates
3. **Batch Processing**: Efficient ETL for large datasets
4. **Index Planning**: Strategic index creation for performance

10.3 Real-World Patterns

1. **Bounded vs. Unbounded Relationships**: When to embed arrays
2. **Reference Patterns**: Using $lookup for occasional joins
3. **Hybrid Approaches**: Combining embedding and referencing
4. **Migration Strategies**: Moving from relational to document stores

11 Conclusion

The transformation of the Sakila database from a 16-table relational model to a 4-collection MongoDB schema demonstrates the practical application of document-oriented design principles. Our "Goldilocks Denormalization" approach achieves:

- **90% storage reduction** compared to full denormalization
- **80% of queries require no joins**, compared to 0% in the relational model
- **Simple update patterns** with single-document modifications
- **Predictable performance** with bounded document sizes
- **Maintainable consistency** through limited, strategic embedding

This design serves not only as a functional MongoDB schema but also as a comprehensive teaching tool for understanding the principles, trade-offs, and best practices of NoSQL database design. The balance achieved between the extremes of full normalization and complete denormalization illustrates the nuanced decision-making required in real-world database architecture.

12 Future Considerations

12.1 Evolution Paths

As the system grows, consider these potential optimizations:

1. **Time-Series Optimization**: Separate historical rentals into time-bucketed collections
2. **Caching Layer**: Implement Redis for frequently accessed film data
3. **Search Enhancement**: Integrate Elasticsearch for advanced film search
4. **Analytics Pipeline**: Stream rentals to a data warehouse for business intelligence

12.2 Monitoring Metrics

Key performance indicators to track:

- Average document size growth
- Query response time percentiles (P50, P95, P99)
- Index hit ratios
- Update operation latency
- Storage growth rate

A Sample Aggregation Pipelines

```
db.rentals.aggregate([
  // Join with customer data
  {$lookup: {
    from: "customers",
    localField: "customer_id",
    foreignField: "customer_id",
    as: "customer"
  }},
  {$unwind: "$customer"},

  // Calculate total per customer
  {$unwind: "$payments"},
  {$group: {
    _id: "$customer_id",
    customer_name: {$first: {$concat: [
      "$customer.first_name", " ", "$customer.last_name"
    ]}},
    email: {$first: "$customer.email"},
    total_spent: {$sum: "$payments.amount"},
    rental_count: {$sum: 1},
    avg_payment: {$avg: "$payments.amount"},
    first_rental: {$min: "$rental_date"},
    last_rental: {$max: "$rental_date"}
  }},

  // Calculate customer lifetime in days
  {$addFields: {
    lifetime_days: {$divide: [
      {$subtract: ["$last_rental", "$first_rental"]},
      1000 * 60 * 60 * 24
    ]}
  }},

  // Sort by total spent
  {$sort: {total_spent: -1}},
  {$limit: 20}
])
```

Listing 7: Customer Lifetime Value Calculation

```
db.rentals.aggregate([
  // Group by film
  {$group: {
    _id: "$film_id",
    rental_count: {$sum: 1},
    total_revenue: {$sum: {$sum: "$payments.amount"}},
    avg_rental_duration: {$avg: {
      $divide: [
        {$subtract: ["$return_date", "$rental_date"]},
        1000 * 60 * 60 * 24
      ]
    }}
  }},

  // Join with film details
  {$lookup: {
    from: "films",
    localField: "_id",
    foreignField: "film_id",
    as: "film"
  }},
  {$unwind: "$film"},

  // Calculate ROI
  {$addFields: {
    roi: {$multiply: [
      {$divide: [
        "$total_revenue",
        "$film.replacement_cost"
      ]},
      100
    ]},
    revenue_per_rental: {$divide: [
      "$total_revenue",
      "$rental_count"
    ]}
  }},

  // Project final results
  {$project: {
    title: "$film.title",
    rating: "$film.rating",
    categories: "$film.categories.name",
    rental_count: 1,
    total_revenue: {$round: ["$total_revenue", 2]},
    roi_percentage: {$round: ["$roi", 1]},
    avg_rental_days: {$round: ["$avg_rental_duration", 1]},
    revenue_per_rental: {$round: ["$revenue_per_rental", 2]}
  }},

  // Sort by ROI
  {$sort: {roi_percentage: -1}},
  {$limit: 25}
])
```

Listing 8: Film Performance Analytics