

INTRODUCTION TO PATTERNS

Lecturer: Trần Thế Trung

1. Introduction to Patterns;
2. Handling Duplication;
3. Handling Staleness;
4. Handling Referential Integrity;



Introduction to Patterns

In the previous chapters, we went over the basis of data modeling for MongoDB, from defining a flexible methodology to identifying the different representations for modeling relationships.

You should now have all the knowledge needed to attack this week's chapter on patterns.

Patterns are very exciting because they are the most powerful tool for designing schemas for MongoDB and NoSQL.

This chapter is going to help you produce a solution that can scale and perform under stress.

Introduction to Patterns

Patterns are not full solutions to problems. Patterns are a smaller section of those solutions. They are reusable units of knowledge.

The patterns we'll go over in this chapter will become a cookbook of powerful transformation to unleash the power of MongoDB schemas.

We will see how to optimize when:

- Faced with large documents with a **subset pattern**;
- Use the **computed pattern** to avoid repeated computations;
- Structure similar fields with the **attribute pattern**;
- Handle changes to your deployment without downtime with the **schema versioning pattern** and much more...



MongoDB Schema's

Those patterns will also serve as a common language for your teams working on schema designs.

Introduction to Patterns

Patterns	Use Case Categories						
	Catalog	Content Management	Internet of Things	Mobile	Personalization	Real-Time Analytics	Single View
	✓		✓	✓		✓	
	✓	✓					✓
			✓			✓	
	✓		✓	✓	✓	✓	✓
	✓	✓			✓		✓
	✓			✓		✓	
			✓	✓	✓		
			✓			✓	
	✓	✓		✓			✓
	✓	✓	✓	✓	✓	✓	✓
	✓	✓		✓	✓		
	✓	✓					

Finally, having well-defined patterns and understanding when and how to use them will remove a little bit of the are in data modeling for MongoDB and make the process more predictable.

Handling Duplication Staleness and Referential Integrity

Apply Patterns may lead to

1. Duplication

- Duplicating data across documents

2. Data Staleness

- Accepting staleness in some pieces of data

3. Data Integrity Issues:

- writing extra application side logic to ensure referential integrity

Handling Duplication Staleness and Referential Integrity

Let's answer some concerns the usage of patterns may arise.

Patterns are a way to get the best out of your data model. Often, the main goal is to optimize your schema to respond to some performance operation or optimize it for a given use case or access pattern. Although like many things in life, good things come at a cost. Many patterns lead to some situations that would require some additional actions.

For example, duplicating data across documents, accepting staleness in some pieces of data, writing extra application side logic to ensure referential integrity.

Choosing a pattern to be applied to your schema requires taking into account these three concerns. If these concerns are more important than the potential simplicity of performance gains provided by the pattern, you should not use the pattern.

Handling Duplication

- **Why?**
 - Result of embedding information in a given document for faster access.
- **Concern?**
 - Challenge for correctness and consistency



Handling Duplication

1. is the solution.

Let's start with a situation where duplicating information is better than not doing it.

```
{ // order
  date: "2009-02-14",
  status: "delivered",
  customer_id: "12345",
  items: [
    { item_id: "B000HC2LI0",
      quantity: 2 }
  ]
}

{ // customer
  _id: "12345",
  since: "2009-01-09",
  address: {
    street: "1600 Pennsylvania Ave",
    city: "Washington",
    state: "DC",
    country: "USA"
  }
}
```

```
{ // order
  date: "2009-02-14",
  status: "delivered",
  customer_id: "12345",
  items: [
    { item_id: "B000HC2LI0",
      quantity: 2 }
  ]
}

{ // customer
  _id: "12345",
  since: "2017-01-20",
  address: {
    street: "Hawaii Plantation Estate",
    city: "Paradise Point",
    state: "HI",
    country: "USA"
  }
}
```



Handling Duplication

1. is the solution.

Let's link orders of products to the address of the customer who placed the order by using a reference to a customer document. Updating the address for this customer updates information for the already fulfilled shipments, and orders that have been already delivered to the customer. This is not the desired behavior.

The shipments were made to the customer's address at that point in time, either when the order was made or before the customer changed their address. So the address reference in a given order is unlikely to be changed.

Embedding a copy of the address within the shipment document will ensure we keep the correct value. When the customer moves, we add another shipping address on file. Using this new address for new orders does not affect the already shipped orders.

The next duplication situation to consider is when the copy data does not ever change.

Handling Duplication


2. has minimal effect.

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV - A New Hope",
  cast: [
    "nm0000434",
    "nm0000148",
    ...
  ]
}

{ // actor
  _id: "nm0000434",
  name: "Mark Hamill",
  filmography: [
    { "tt0076759": "Luke Skywalker" },
    { "tt0080684": "Luke Skywalker" },
    ...
  ]
}
```

```
{ // movie 1
  title: "Star Wars - IV - A New Hope",
  cast: [
    { "Mark Hamill": "Luke Skywalker" },
    { "Harrison Ford": "Han Solo" },
    ...
  ]
}

{ // movie 2
  title: "Star Wars - V - The Empire Strikes Back",
  cast: [
    { "Mark Hamill": "Luke Skywalker" },
    { "Harrison Ford": "Han Solo" },
    ...
  ]
}
```



Handling Duplication

2. has minimal effect.

Let's say we want to model movies and actors. Movies have many actors and actors play in many movies. So this is a typical many-to-many relationship.

Avoiding duplication in a many-to-many relationship requires us to keep two collections and create references between the documents in the two collections.

If we list the actors in a given movie document, we are creating duplication. However, once the movie is released, the list of actors does not change.

So duplication on this unchanging information is also perfectly acceptable.

Handling Duplication

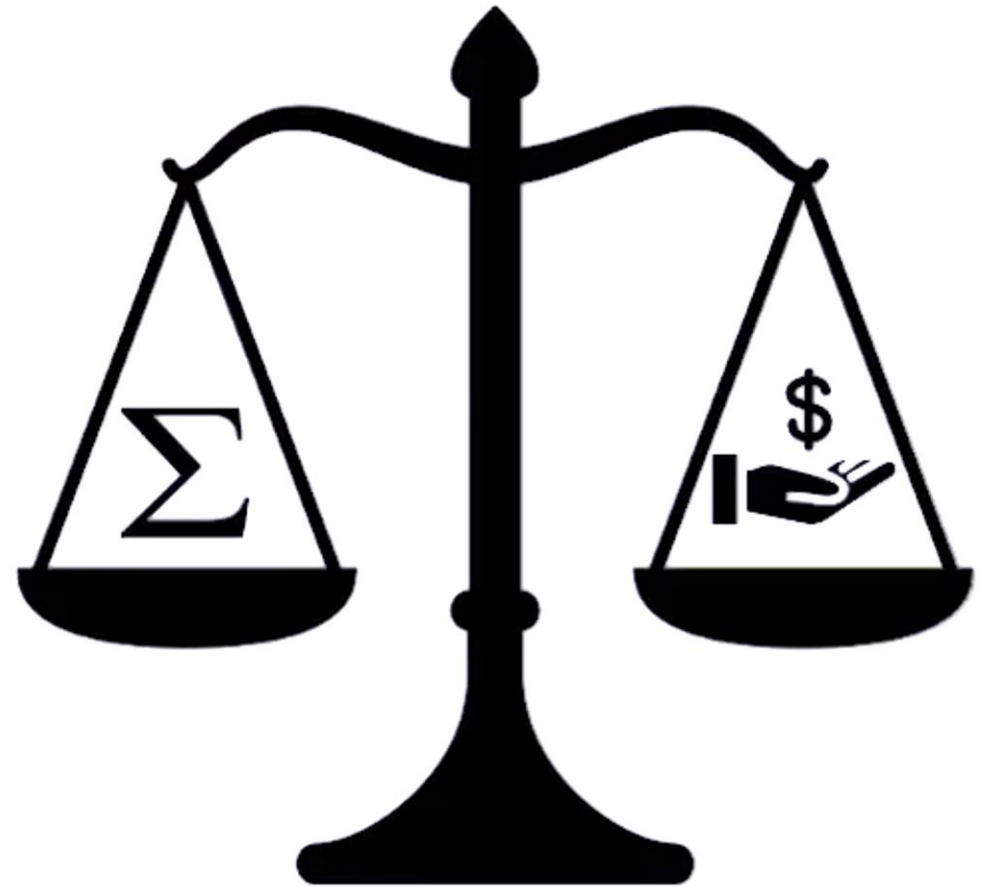
3. Should be handled

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV - A New Hope",
  gross_revenues: 775000000
}

{ // screenings
  date: "1977-05-30",
  revenues: 15554475,
}

...

{ // screenings
  date: "2019-05-30",
  revenues: 25000,
}
```



Handling Duplication

3. Should be handled

The duplication of a piece of information that needs to or may change with time. For this example, let's use the revenues for a given movie, which is stored within the movie, and the revenues earned per screening.

In this case, we have a duplication between the sum stored in the movie document and the revenue stored in the screening documents used to compute the total sum.

This type of situation, where we must keep multiple values in sync over time, makes us ask the question is the benefit of having this sum precomputed surpassing the cost and trouble of keeping it in sync?

If yes, then use this computed pattern. If not, don't use it.

Handling Duplication

3. Should be handled

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV",
  gross_revenues: 775000000
}
```

```
{ // screenings
  date: "1977-05-30",
  revenues: 15554475,
}
...
{
  date: "2019-05-30",
  revenues: 5000,
}
```

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV",
  gross_revenues: 775000000
}
```

```
{ // screenings
  date: "1977-05-30",
  revenues: 15554475,
}
...
{
  date: "2019-05-30",
  revenues: 25000,
}
...
{
  date: "2019-06-17",
  revenues: 25000,
}
```



```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV",
  gross_revenues: 775025000
}
```

```
{ // screenings
  date: "1977-05-30",
  revenues: 15554475,
}
...
{
  date: "2019-05-30",
  revenues: 25000,
}
...
{
  date: "2019-06-17",
  revenues: 25000,
}
```


Handling Duplication

3. Should be handled

Here, if we want the sum to be synchronized, it may be the responsibility of the application to keep it in sync.

Meaning, that whenever the application writes a new document to the collection or updates the value of an existing document, it must update the sum.

Alternatively, we could add another application or job to do it. But how often should we actually recalculate the sum?

This brings us to the next concern we must consider when using patterns, staleness.

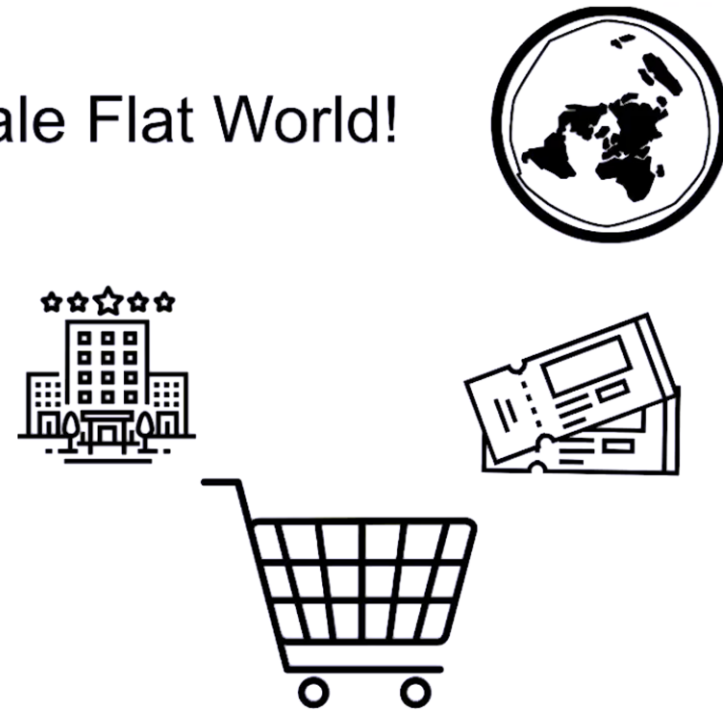
Handling Staleness

Staleness is about facing a piece of data to a user that may have been out of date. We now live in a world that has more staleness than a few years ago.

Due to globalization and the world being flatter, systems are now accessed by millions of concurrent users, making the ability to display up-to-the-second data to all these users more challenging.

For example, the availability of a product that is shown to a user may still have to be confirmed at checkout time. The same goes for prices of plane tickets or hotel rooms that change right before you book them.

It is a Stale Flat World!



Handling Staleness

- **Why?**
 - New events come along at such a rate that updating some data constantly causes performance issues.
- **Concern?**
 - Data quality and reliability.

Long analytic queries?

Queries on secondary nodes?





Handling Staleness

The right question is, for how long can the user tolerate not seeing the most up-to-date value for a specific field.

For example, the user's threshold for seeing if something is still available to buy is lower than knowing how many people view or purchase a given item.

When performing analytic the queries it is often understood that the data may be stale and that the data being analyzed is based on some past snapshot.

Analytic queries are often run on the secondary node, which often may have stale data. It may be a fraction of a second or a few seconds out of date.

However, it is enough to break any guarantee that we're looking at the latest data recorded by the system.

Handling Staleness

Resolve Staleness

Batch Updates



Find the changes through a
Change Stream



Handling Referential Integrity

- **Why?**
 - Linking information between documents or tables
 - No support for cascading deletes
- **Concern?**
 - Challenge for correctness and consistency





Handling Referential Integrity

Our third concern, when using patterns, is referential integrity. Referential integrity has some similarities to staleness.

It may be OK for the system to have some extra or missing links, as long as they get corrected within the given period of time.

Why do we get referential integrity issues? Frequently, it may be the result of deleting a piece of information from the document. for example, without deleting the references to it.

Handling Referential Integrity

Resolve referential integrity

A. Change Streams



B. Single Document



C. Multi Documents Transaction





Handling Referential Integrity

Recap:

For a given piece of data...

- Should or could the information be duplicated or not?
 - Resolve with bulk updates
- What is the tolerated or acceptable staleness?
 - Resolve with updates based on change streams
- Which pieces of data require referential integrity?
 - Resolve or prevent the inconsistencies with change streams or transactions