Professional Service

MongoDB

Scheme Design workshop

Kunhan Choi
Senior Consulting Engineer at MongoDB

Topics we cover

BSON

Document Model

Relationship

Payload and Process fields

Schema Design Patterns

Case Study

What is Data Modeling ?

MongoDB data modeling refers to designing the structure of data stored in the database

It involves defining the format, **relationships**, and **access patterns** of data

MongoDB is a NoSQL database and stores data in JSON format, commonly referring to data as documents

Data modeling is a crucial factor in determining an **application's performance, maintainability, and scalability**

Appropriate modeling has a significant impact on **improving the performance of a database**

Data Modeling Process in General

Gather requirements

- Understand **what data needs to be stored** and how it will be used.

Identify entities and relationships

- Identify the entities (objects or concepts) that need to be stored and the relationships between them

Normalize the data model

- This reduces data redundancy and ensures data integrity.

Denormalize for performance

- This involves duplicating data across tables to minimize joins.

Data model review and iteration

- Ensure it meets the requirements and performs as expected
- The model may need to be revised as requirements change or new issues arise.

Data Modeling Process in MongoDB

Identifying access patterns

- **Access patterns define how data is queried and written**

Entity & Relationship modeling

- **Embedding(denormalization) or Linking(normalization) based on access patterns**

Query and index design

- Queries required by the application based on the access patterns are written

Data model review and iteration

- New access patterns can be identified or indexes can be adjusted to improve query performance

What is good model ?

Represent all aspects of data

Good query Performance

Flexibility to handle changing data and query requirements

Optimize hardware footprint (CPU / RAM / IO)

Developer Productivity

Scalability

BSON

BSON

MongoDB stores data in the form of BSON. BSON is a document format similar to JSON for storing data and supports more **data types** compared to regular JSON. It also allows storing binary data such as images or videos. BSON is the internal data format used by MongoDB to store and query data.

In MongoDB, you can store a single BSON up to 16MB.

Binary Serialized Object Notation (http://bsonspec.org/spec.html)

- Basically, a list of named, typed values:

  Type, Fieldname, <length>, Data

  Type, Fieldname, <length>, Data

Finding fields in BSON

According to the BSON spec, finding a specific field is

- Load BSON
- Read length of field and field name
- Check if a field name matches with given field name
- If it matches, retrieve the value
- If it does not match, jump length bytes and read next field until it matches

What if we have thousands of fields in BSON ?

What if we have tens of object fields with hundreds fields per each ?

BSON spec

```
document ::=  int32 e_list "\x00"
e_list   ::=  element e_list
element  ::=  "\x01" e_name double  64-bit binary floating point
           |  "\x02" e_name string  UTF-8 string
           |  "\x03" e_name document    Embedded document
           |  "\x04" e_name document    Array
```

Are Arrays and Embedded Documents both just types of document then?

Arrays vs Documents

The only difference between an array and an embedded document is just one byte. This means that an array and an embedded document in BSON have a very similar structure, and the only difference is that an array has a type code of 0x04, while an embedded document has a type code of 0x03.

```
{ 0 : "Red",

  1 : "Orange",

  2 : "Yellow" }

[ "Red", "Orange", "Yellow" ]
```

Document Model

Documents

Image Map/Object type in your program language

Documents can be nested

Documents can not contain duplicates in a same field

$project is faster and convenient when you need a portion of fields

Querying a single member is also faster than array

You can index on fields (single or compounded)

Arrays

Image List/Array/Slice type in your program language

Indexing in array is possible to search its element. (Multikey Index)

There’re many operators that handles array.

- $elemMatch / $all / $size / $push / $pop / $pull / $pullAll / $each ...
- Don’t update whole array, it can cause serious problems.

Arrays can contain objects as well as just primitive type

- Array of objects is very common for combining two collections into one.
- Powerful type to simplify your models

Arrays can not be sparse so you need to add null explicitly

Arrays can contain different types of elements, but don’t do that.

The number of element should be bounded not to exceed the document size.

Document Model

Relation model is form of two dimensional, normally it does not have arrays and documents types.

Document Model is form of multi-dimensional, it has arrays and documents types.

It brings us great advantages

- Fewer Joins
- Fewer Indexes
- Faster Updates
- Faster Retrievals
- Enabling Scalability

Relation vs Document Model

RDBMS – One correct design

- 3rd Normal Form (or 6th normal form)
- Design based on the data, not the usage
- Reality is more nuanced

Document – Design Patterns

- Many design options
- Designed for a usage pattern
- Retrieval (Fast retrieval of required info)
- Updates (Atomic update of business logic)

Schema Design in MongoDB

Schema is defined at the application-level

- Focus on the needs of the application
- Not the abstract nature of the data

Design is the part of each phase in its lifetime

Schema design should focus on the application and not the data

Four Considerations

The data the application needs

Application’s read usage of the data

Application’s write usage of the data

Data growth and possible outlier

RelationShip

Relationship

Document design offers two basic ways to represent relationship

Embedding involves nesting one document within another document

Linking involves connecting documents through references.

The choice between embedding and linking should be based on the nature of the data and the requirements of the application.

Embedding

Advantages

- Retrieve all relevant information in a single query/document
- Avoid implementing joins in application code (or $lookup)
- Update related information as a single atomic operation
- No need for transactions (code complexity, performance)

Limitations

- Possible data duplication
- Large documents mean more overhead if most fields are not relevant
- 16 MB document size limit

Linking

Advantages

- Smaller documents
- Less likely to reach 16 MB document limit
- Lightweight because it only contains the necessary data
- No duplication of data

Limitations

- Two queries (or $lookup) required to retrieve information
- Cannot update related information atomically (without transactions)

Link vs Embed

To Link or Embed?

- Do I want the embedded info a lot of the time?
- Do I need to search with the embedded info?
- Does the Embedded info change often?
- Do I need the latest version or the same version?

What about an Invoice Address, link, or embed?

Atomicity

Document operations are atomic (Embed)

```
db.patients.updateOne({_id: 12345},
   {
        $inc : {numProcedures : 1},
        $push : {procedures : “proc123”},
        $set : {“addr.state” : “TX”}
    }
)
```

Multi-document transactions (Link)

```
s1.startTransaction();
db.patients.updateOne({_id: 12345}, …);
db.procedure.insertOne({_id: “proc123”, …});
db.records.insertOne({_id: “rec123”, …});
s1.commitTransaction();
```

Relation Types

One to One Relationships

One to Many Relationships

Many to Many Relationships

One to One Linked

```
// students collections
{
  "_id": 1,
  "name": "John Doe",
  "age": 21,
  "address": "123 Main St",
  "student_info_id": 1
}
```

```
// student_info collection
{
  "_id": 1,
  "major": "Computer Science",
  "gpa": 3.8,
  "graduation_date": "2025-05-01”,
  “student_id” : 1
}
```

One to One Embedded

```
// students collections
{
  "_id": 1,
  "name": "John Doe",
  "age": 21,
  "address": "123 Main St",
  “student_info” : {
      "major": "Computer Science",
      "gpa": 3.8,
      "graduation_date": "2025-05-01”,
  }
}
```

One to One Relationship

When to Embed ?

- Usually
- Small Documents
- Related data often used together
- Need to update both collections atomically

When to Link ?

- One or both parts of document are huge
- Related information is rarely accessed
- To avoid hitting document limitation (16 MB)

One to Many Linked

```
// Patient collections
{
  "_id": 1,
  "name": "John Doe",
  "age": 21,
  "address": "123 Main St",
  “Procedures” : [
      1001, 1002
  ]
}
```

```
// Procedures collections
{
  "_id": 1001,
  "name": "PET Scan",
  "date": ISODate(“2023-02-23”)
}
{
  "_id": 1002,
  "name": "Blood Test",
  "date": ISODate(“2023-03-23”)
}
```

One to Many Linked - Child

```
// Patient collections
{
  "_id": 1,
  "name": "John Doe",
  "age": 21,
  "address": "123 Main St",
}
```

```
// Procedures collections
{
  "_id": 1001,
  "name": "PET Scan",
  "date": ISODate(“2023-02-23”),
  “patent_id” : 1
}
{
  "_id": 1002,
  "name": "Blood Test",
  "date": ISODate(“2023-03-23”),
  “patient_id” : 1
}
```

One to Many Embedded

```
// Patient collections
{
  "_id": 1,
  "name": "John Doe",
  "age": 21,
  "address": "123 Main St",
  “procedures” : [
   {
      "name": "PET Scan",
      "date": ISODate(“2023-02-23”)
   },
   {
      "name": "Blood Test",
      "date": ISODate(“2023-03-23”)
   }
   ]
}
```

One to Many Relationships

When to Embed ?

- The number of N's document per 1's is small (cardinality)
- Many document is small
- Related data often used together
- Need to update both collections atomically

When to Link ?

- The number of N's document per 1's is large
- One or both parts of document are huge
- Related information is infrequently accessed
- To avoid hitting document limitation (16 MB)

Many to Many Link

```
// patients collections
{
  "_id": 1,
  "name": "John Doe",
  "age": 21,
  “doctors” : [ 1,2 ]
}
{
  "_id": 2,
  "name": "Jane Doe",
  "age": 42,
  “doctors” : [ 2,3 ]
}
```

```
// doctors collections
{
  "_id": 1,
  "name": "Dr. Strange",
  "address": "123 Main St",
  “major” : “surgeon”,
  “patients” :[1,4,2]
}
{
  "_id": 2,
  "name": "Dr. House",
  "address": "456 Main St",
  “major” : “physician”,
  “patients” : [1,5,2,3]
}
```

Many to Many Embedded

```
// patients collections
{
  "_id": 1,
  "name": "John Doe",
  "age": 21,
  “doctors” : [
     {
       "_id": 1,
       "name": "Dr. Strange",
       "address": "123 Main St",
       “major” : “surgeon”
     },
     {
       "_id": 1,
       "name": "Dr. House",
       "address": "456 Main St",
       “major” : “physician”
     },
}
```

```
// doctors collections
{
  "_id": 1,
  "name": "Dr. Strange",
  "address": "123 Main St",
  “major” : “surgeon”,
  “patients” : [
     { "_id": 1,
       "name": "John Doe",
       "age": 21 },
     { "_id": 2,
       "name": "Jane Doe",
       "age": 23 }
   ]
}
```

Many to Many Relationships

When to Embed ?

- The number of N’s document per 1’s is small (cardinality)
- Duplicated data is infrequently update
- Duplicated data will significantly improve query performance
- Each embedded document is small
- Related data often used together

Many to Many Relationships

When to Link ?

- The number of N’s document per 1’s is large (cardinality)
- One or both parts of document are huge
- Related information is infrequently accessed
- To avoid hitting document limitation (16 MB)
- Duplicated data is updated often

Directions

```
// patients collections
{
  "_id": 1,
  "name": "John Doe",
  "age": 21,
  “doctors” : [ 1,2 ]
}
```

```
// doctors collections
{
  "_id": 1,
  "name": "Dr. Strange",
  "address": "123 Main St",
  “major” : “surgeon”,
  “patients” :[1,3,4,5,6,7,8,100,.....10321]
}
```

Which direction you should choose ? Why ?

Relationship Summary

Relation Type	1 to 1	1 to N	N to N
Embed	One Read No Join	One Read No Join	One Read No Join Data Duplication
Reference	Smaller Read or Join	Smaller Read or Join	Smaller Read Avoid Duplication

Exercise: Users and Book reviews

Design a schema for users and their book reviews. UserNames are immutable.

Users

- userName (string)
- email (string)

Reviews

- text (string)
- rating (int32)
- created_at (date)

Library Management Application

Type A : Reviews may be queried by user or book

```
// db.users (one document per user)
{
        _id: ObjectId("..."),
        userName: "bob",
        email: "bob@example.com"
}
```

```
// db.reviews (one document per review)
{
        _id: ObjectId("..."),
        user: ObjectId("..."),
        book: ObjectId("..."),
        rating: 5,
        text: "This book is excellent!",
        created_at: ISODate("2012-10-10T21:14:07.096Z")
}
```

Library Management Application

Type B : Optimized to retrieve reviews by user

```
// db.users, one document per user with all reviews
{
	_id: ObjectId("..."),
	userName: "bob",
	email: "bob@example.com",
	reviews: [{
	  book: ObjectId("..."),
	  rating: 5,
	  text: "This book is excellent!",
	  created_at: ISODate("2012-10-10T21:14:07.096Z")
	}]
}
```

Library Management Application

Type C : Optimized to retrieve reviews by book

```
// db.users (one document per user)
{
        _id: ObjectId("..."),
        userName: "bob",
        email: "bob@example.com"
}
```

```
// db.books, one document per book with all reviews
{
        _id: ObjectId("..."),
        // Other book fields...
        reviews: [{
          user: ObjectId("..."),
          rating: 5,
          text: "This book is excellent!",
          created_at: ISODate("2014-11-10T21:14:07.096Z")
        }]
}
```

Payload vs Process

Type of Fields

Payload

- Data we just store and retrieve

Processing

- Metadata we examine inside MongoDB
- Querying / Aggregating / Filtering & Projecting / using workflow / using implement patterns

Payload might be predefined, sometimes better to ignore that and add Processing fields
For example, existing corporate data/object models, might be used only for storage
JSON files you want to store is Payload, fields you want to filter by are processing fields

Type of Fields

Need to store XML objects defined as part of the business in a MongoDB collection

First idea: convert XML to JSON and keep it in MongoDB

- But only ever query a few fields
- Normally always want just the latest
- XML structures are complex or dynamic
- Need to write a reverse conversion too

Conversion XML <-> JSON is not a good idea, it’s expensive and difficult

Best option:

- Extract the processing data (fields needed for queries like create date, etc.)
- Store processing fields in documents
- Put the XML itself in a Binary (compressed) – it's a payload field

Quiz. Comparing Schemas

Redesign to be smaller

In this case, 30% smaller documents

Which model is a better choice? and Why?

```
> var doc = { results: [
{player: 'john', score: 25},
{player: 'fred', score: 20},
{player: 'sarah', score : 50}]}

> bsonsize(doc)
128

> var doc = { results: {
  john : {score: 25},
  fred : {score: 20},
  sarah: {score: 50}}}

> bsonsize(doc)
86
```

Schema Patterns

Common Design Patterns

Attributes

Extended Reference

Subset

Bucket

Computed

Schema Versioning

Attribute Pattern

BIG Documents, Many Fields, **Many Indexes**

```
{
    title: "Star Wars",
    director: "George Lucas",
    …
    release_US: ISODate("1977-05-20"),
    release_Korea: ISODate("1977-10-19"),
    release_Italy: ISODate("1977-10-20"),
    release_UK: ISODate("1977-12-27"),
    …
}
```

We may require the following indexes…

```
{ release_USA: 1 }

{ release_Korea: 1 }

{ release_France: 1 }

…

{ release_UK: 1 }
…
```

Attribute Pattern

Require 1 index : {“releases.location” : 1, “releases.date”: 1}

```
{
    title: "Star Wars",
    director: "George Lucas",
    ...
    release_US: ISODate("1977-05-20"),
    release_Korea: ISODate("1977-10-19"),
    release_Italy: ISODate("1977-10-20"),
    release_UK: ISODate("1977-12-27"),
    ...
}
```

```
movies
_id: ObjectId
title: string
...
    releases:
    location: string
    date: date
```

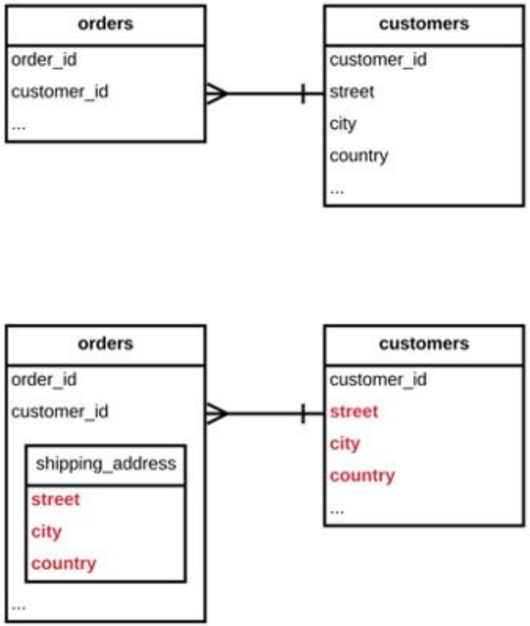
```
{
    title: "Star Wars",
    director: "George Lucas",
    ...
    releases: [
        { location: "USA",
          date: ISODate("1977-05-20")},
        { location: "Korea",
          date: ISODate("1977-10-19")},
        { location: "Italy",
          date: ISODate("1977-10-20")},
        { location: "UK",
          date: ISODate("1977-12-27")},
        ...
    ],
    ...
}
```

Attribute Patterns

Problems	There are many similar fields that need to be searched together and exist only as a small subset of a document.
Solution	Bundle the fields together in an array format and separate the field and value into sub-documents: • fieldA: field • fieldB: value Example: { "color": "blue", "size": "large" }, { "height": 16, "weight": 41 } [{"k": "color", "v": "blue"}, {"k": "size", "v": "large"}, {"k": "height", "v": 16}, {"k": "weight", "v": 41}]
Use case	• Product attributes such as "color", "size", "dimensions" • Events and releases in different countries for movies, festivals, and other occasions • A set of fields with the same value type
Benefits	**Easy to create indexes such as { "release.location": 1, "release.date": 1 }, { k: 1, v: 1 }** **Easy to expand using qualifiers such as :** **{ descriptor: "price", qualifier: "euros", value: Decimal(100.00) }** **{ descriptor: "price", qualifier: "won", value: Decimal(80000.00) }**

Extended Reference Pattern

The MongoDB Extended Reference Pattern is one of the patterns used to represent relational data in MongoDB databases. While reference fields are typically used to normalize data, it can be inefficient to fetch all the data for a referenced document when the document is very large or contains a lot of related data.



Extended Reference Pattern

Extended Reference Pattern

Problems	There is too much repetitive joining in the database, which leads to slow read times.
Solution	Define the fields on the lookup collection side and bring them into the main object to reduce JOIN and LOOKUP counts.
Use case	• Catalogs • Mobile applications • Real-time analytics
Benefits	**Fast read times** **Reduced JOIN and LOOKUP counts.**

Subset Pattern

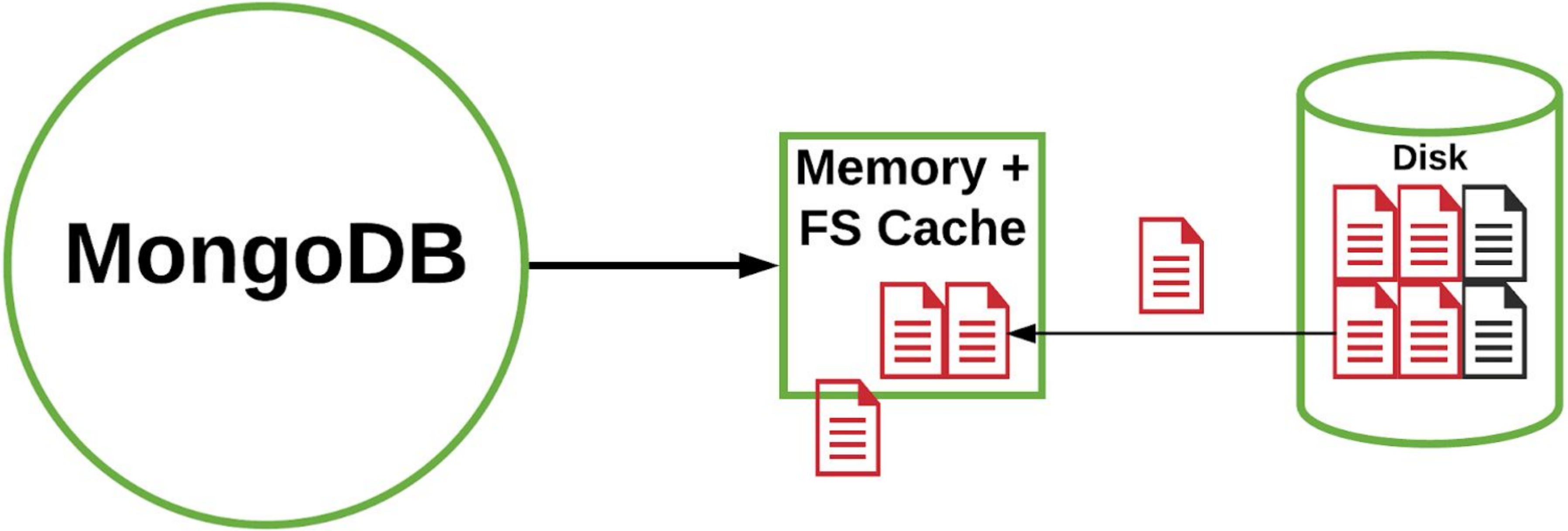
Use Linking to another document

Maintain a subset of the linked data embedded for speed

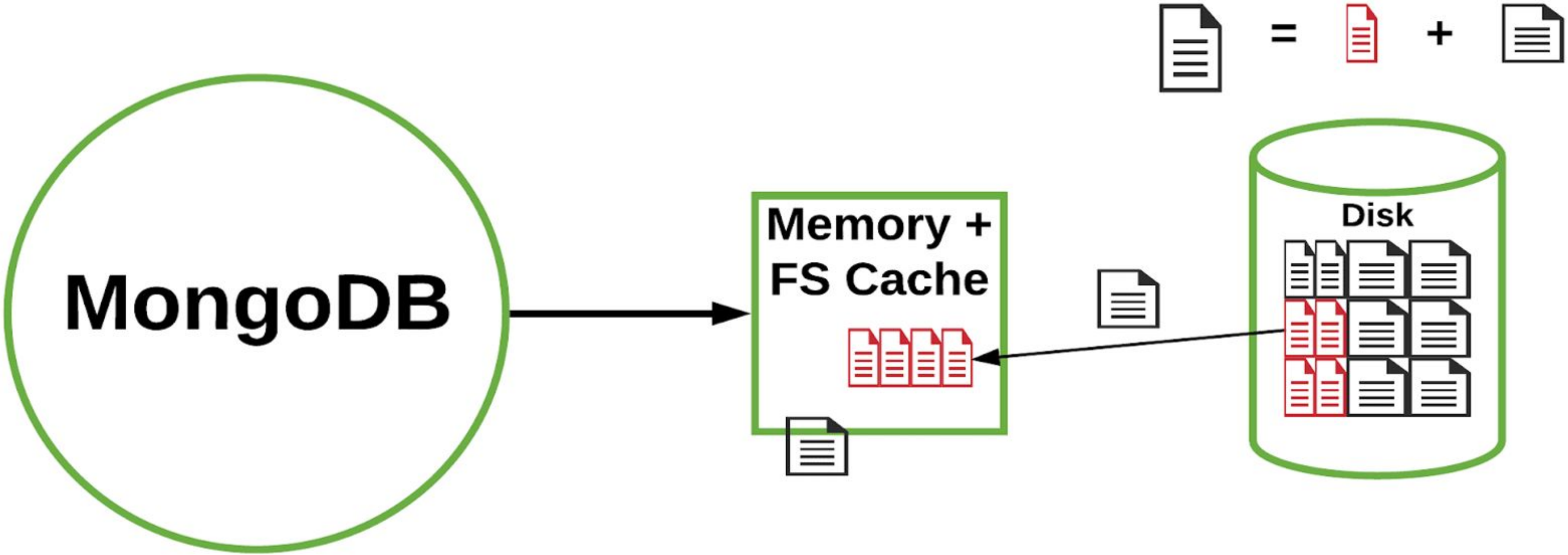
Hybrid of linking and Embedded - common caching pattern:

- Keep an embedded set of the linked data in the main document
- Read the most frequently/recently accessed docs directly from the parent doc
- Can contain only a **subset of all** linked documents or all of them

Subset Pattern

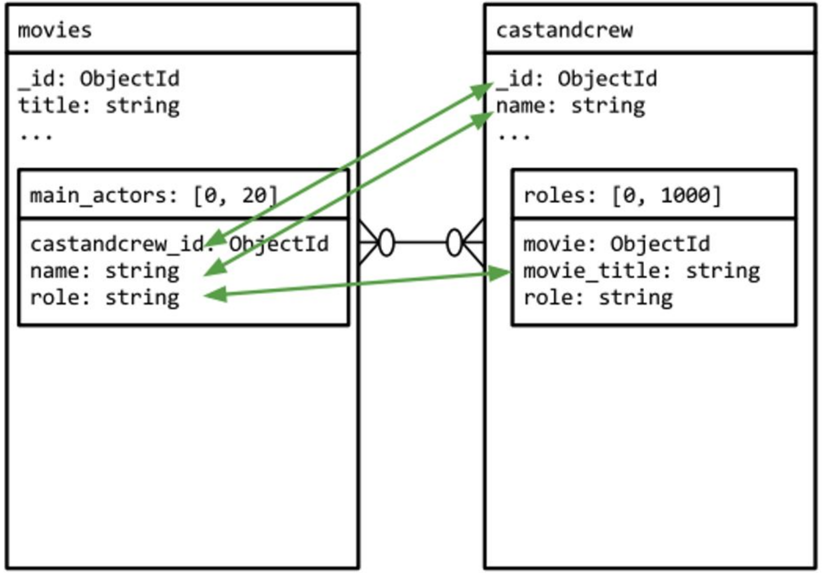


Subset Pattern



Subset Pattern

Example #1. 20 actors who appeared in the movie



Subset Pattern

Example #2.

Retrieve the latest 10 reviews for the product

```
{
    _id: ObjectId("507f1f77bcf86cd799439011"),
    name: "Super Widget",
    description: "This is the most useful item in your toolbox."
    price: { value: NumberDecimal("119.99"), currency: "USD" },
    reviews: [
        {
          review_id: 786,
          review_author: "Kristina",
          review_text: "This is indeed an amazing widget.",
          published_date: ISODate("2019-02-18")
        },
        {
          review_id: 785,
          review_author: "Trina",
          review_text: "Very nice product, slow shipping.",
          published_date: ISODate("2019-02-17")
        },
  ...
        {
          review_id: 1,
          review_author: "Hans",
          review_text: "Meh, it's okay.",
          published_date: ISODate("2017-12-06")
        }
    ]
}
```

```
{
    _id: ObjectId("507f1f77bcf86cd799439011"),
    name: "Super Widget",
    description: "This is the most useful item in your toolbox."
    price: { value: NumberDecimal("119.99"), currency: "USD" },
    reviews: [
          {
            review_id: 786,
            review_author: "Kristina",
            stars: 5
            review_text: "This is indeed an amazing widget.",
            published_date: ISODate("2019-02-18")
          },
...
          {
            review_id: 776,
            review_author: "Pablo",
            stars: 5
            review_text: "Wow! Amazing.",
            published_date: ISODate("2019-02-16")
          }
    ]
}
```

Product Collection

```
{
    review_id: 786,
    product_id: ObjectId("507f1f77bcf86cd799439011"),
    review_author: "Kristina",
    review_text: "This is indeed an amazing widget.",
    published_date: ISODate("2019-02-18")
}
{
    review_id: 785,
    product_id: ObjectId("507f1f77bcf86cd799439011"),
    review_author: "Trina",
    review_text: "Very nice product, slow shipping.",
    published_date: ISODate("2019-02-17")
}
{
    review_id: 1,
    product_id: ObjectId("507f1f77bcf86cd799439011"),
    review_author: "Hans",
    review_text: "Meh, it's okay.",
    published_date: ISODate("2017-12-06")
}
```

Review Collection

Subset Pattern

Problems	Working Set too large, causing frequent page evictions from memory. Many document fields are rarely used.
Solution	Split collection into two - one for frequently used fields, and another for rarely used fields.
Use case	Product reviews, article comments, movie cast.
Benefits	**Smaller working set** **Reduced disk access time when retrieving additional documents from the most frequently used collection.**

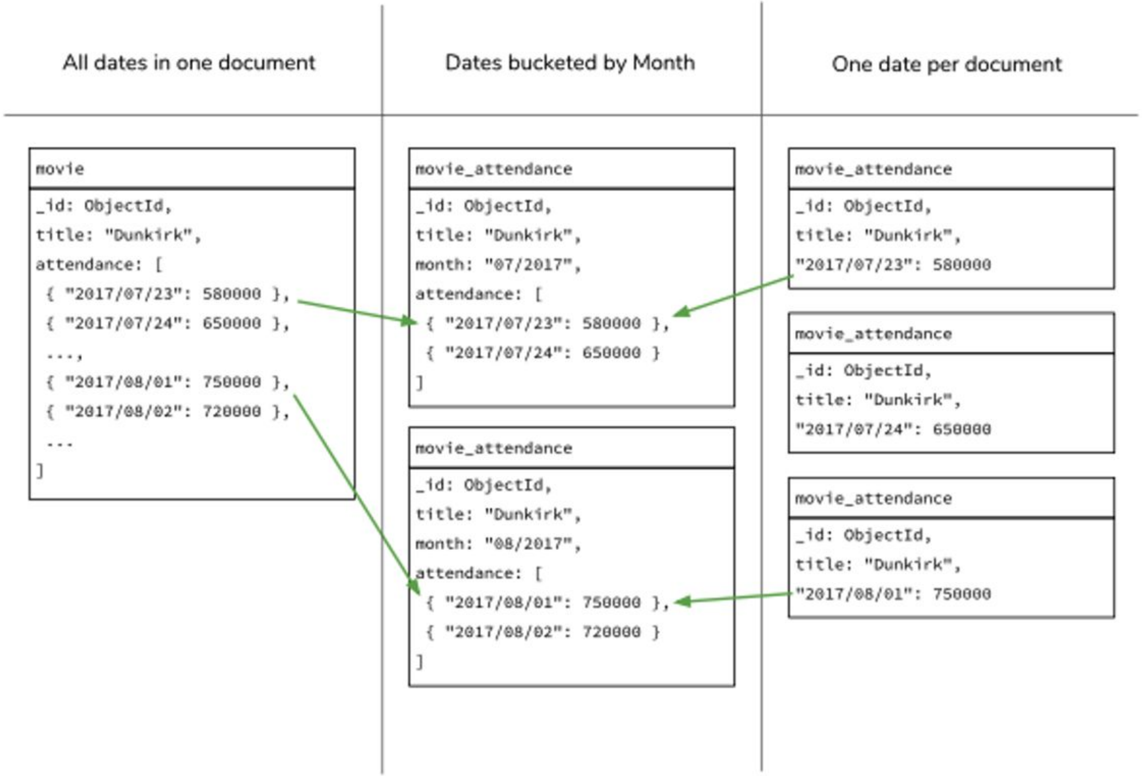
Bucket Patterns

The Bucket Pattern is used to handle large volumes of data by grouping them into smaller, fixed-size chunks called buckets.

Each bucket typically contains a set of documents with a common attribute or range of values, which allows for efficient querying and aggregation of data.

Starting MongoDB 5.0, you can use Timeseries collection instead of implementing bucket pattern by your own hands.

Bucket Patterns



Bucket Patterns

```
{
  sensor_id: 12345,
  timestamp: ISODate("2019-01-31T10:00:00.000Z"),
  temperature: 40
}

{
  sensor_id: 12345,
  timestamp: ISODate("2019-01-31T10:01:00.000Z"),
  temperature: 40
}

{
  sensor_id: 12345,
  timestamp: ISODate("2019-01-31T10:02:00.000Z"),
  temperature: 41
}
```

```
{
    sensor_id: 12345,
    start_date: ISODate("2019-01-31T10:00:00.000Z"),
    end_date: ISODate("2019-01-31T10:59:59.000Z"),
    measurements: [
       {
       timestamp: ISODate("2019-01-31T10:00:00.000Z"),
       temperature: 40
       },
       {
       timestamp: ISODate("2019-01-31T10:01:00.000Z"),
       temperature: 40
       },
       …
       {
       timestamp: ISODate("2019-01-31T10:42:00.000Z"),
       temperature: 42
       }
    ],
   transaction_count: 42,
   sum_temperature: 2413
}
```

Bucket Patterns

Problems	Too many documents or documents that are too large to embed 1:N relationship that cannot be embedded
Solution	Define the optimal amount of information to group together Create an array to store the information in the main object
Use case	IOT Data Warehouse When there is too much information to follow along with one object
Benefits	**Balance between the number of data accesses and data size** **Easier management of data** **Easy data cleanup**

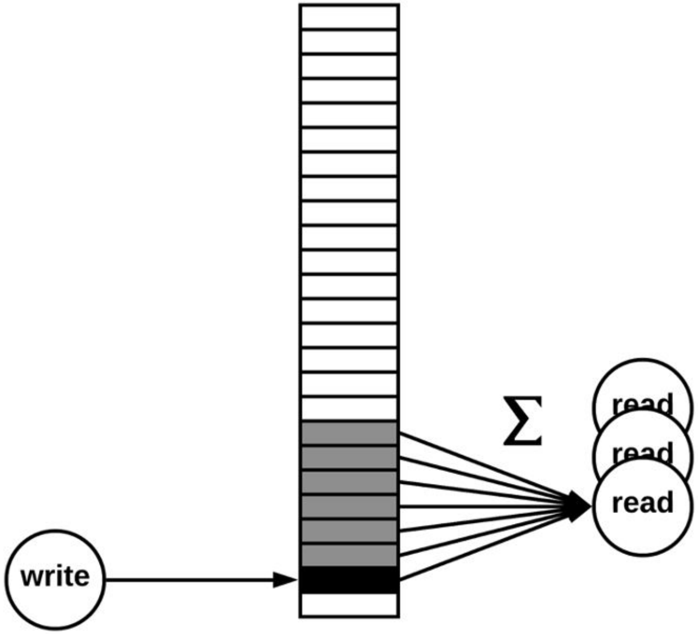
Compute Patterns

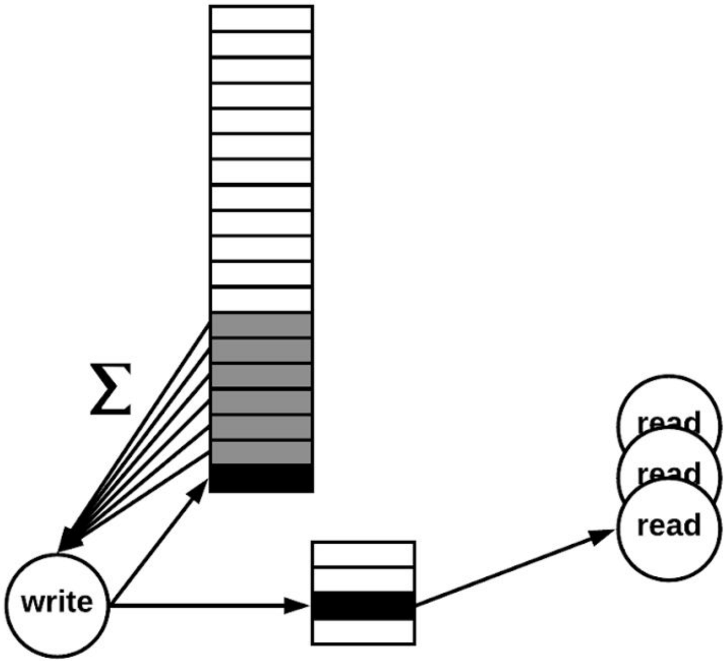
The Compute Pattern is that enables the computation of derived data from a collection at the time of querying.

This allows for more efficient querying and reduces the need for additional storage of redundant or computed data.

The compute pattern is particularly useful in scenarios where there is a need to perform complex calculations or aggregations on large data sets.

Compute Patterns

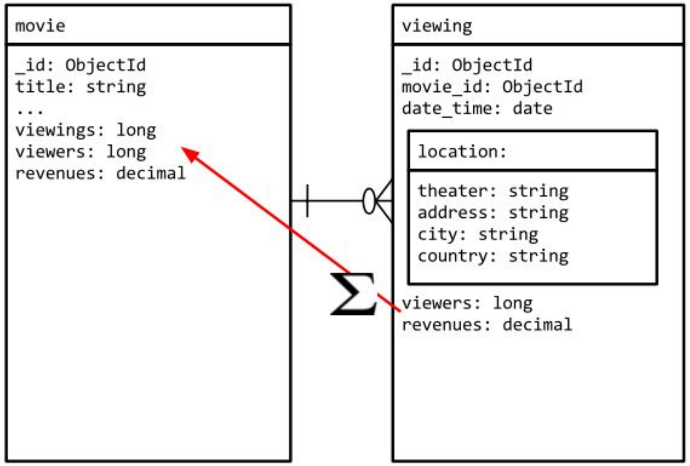




Compute Patterns

movie

{
ts: DateTime(XXX),
movie_title: "Avatar: The Way of Water",
viewers: 2600,
revenue: 33480
}



viewing

{
ts: DateTime(XXX),
theater: "Alger Cinema",
location: "Lakeview, OR",
movie_title: "Avatar: The Way of Water",
num_viewers: 344,
revenue: 3440
},
{
ts: DateTime(XXX),
theater: "City Cinema",
location: "New York, NY",
movie_title: "Avatar: The Way of Water",
num_viewers: 1498,
revenue: 22440
},
{
ts: DateTime(XXX),
theater: "Overland Park Cinema",
location: "Bolse, ID",
movie_title: "Avatar: The Way of Water",
num_viewers: 700,
revenue: 7800
}

Compute Patterns

Problems	Costly computation or manipulation of data on the same data repeatedly generating the same results.
Solution	Perform the operation and store the result in a separate document or collection, and keep the source for future re-execution.
Use case	IoT, event processing, time-series data, frequent Aggregation Framework queries.
Benefits	**Improved read query performance, and saving CPU & disk resources.**

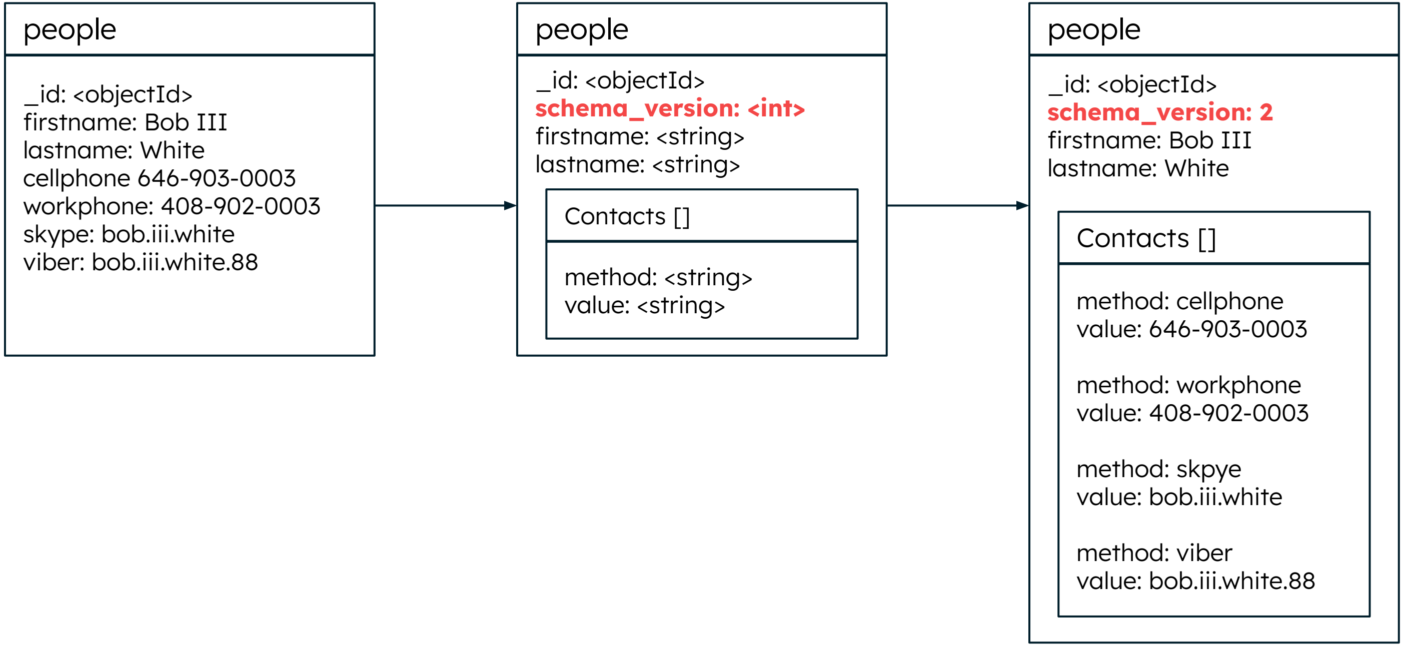
Schema Versioning Pattern

Schema Versioning Pattern is used to handle changes in the structure of the database schema without causing disruptions to the application.

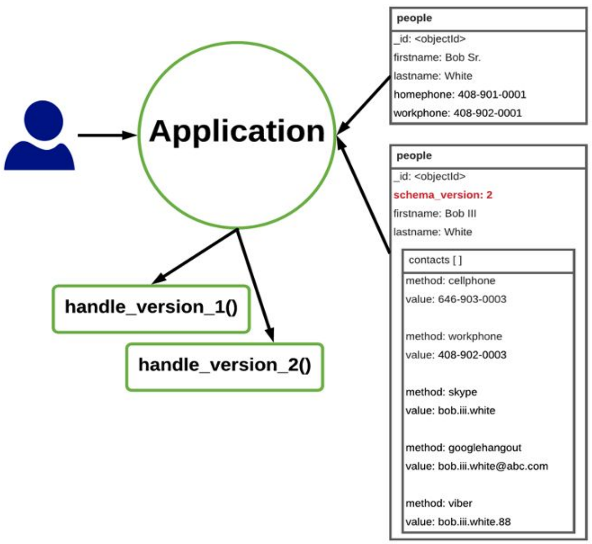
This pattern involves creating multiple versions of the schema and using a **version identifier** to track which version of the schema is currently being used.

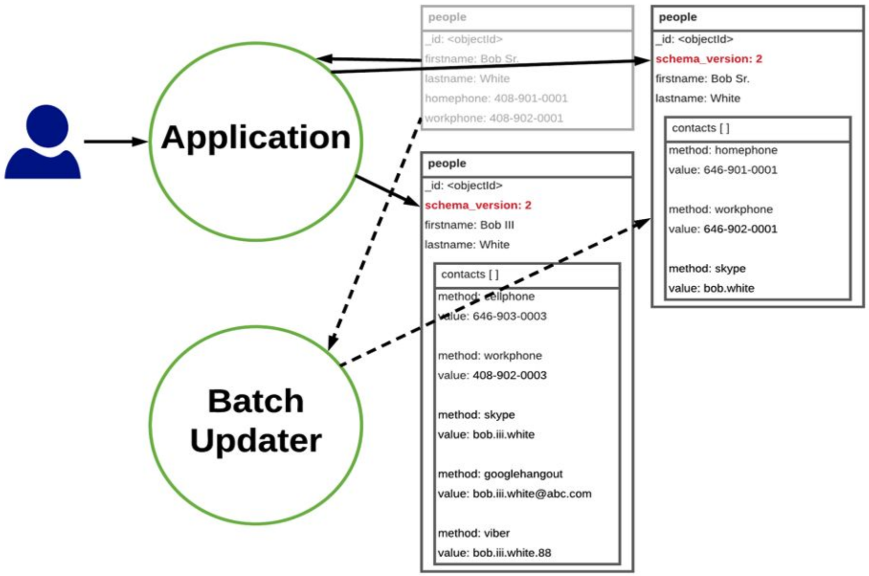
Newer versions can be created to add new fields or make changes to the structure of the schema, while older versions remain in place to maintain compatibility with existing data.

Schema Versioning Pattern



Schema Versioning Pattern





Schema Versioning

Problems	Preventing downtime during schema upgrade Upgrading all documents can be time-consuming when dealing with large amounts of data Not all documents need to be updated
Solution	Each document has a "schema_version" field The application manages all versions Upgrade existing documents to new versions: [batch, when reading]
Use case	When you have a large amount of existing data that cannot be changed at once When many applications use the database and need to be applied to operations or when there is a heavy load
Benefits	**No downtime**

What our Patterns did for us

Problem	Pattern
Using too much RAM	Subset
Big Document, Many Field, Many Indexes	Attribute
Using too much CPU	Computed
No downtime to upgrade schema	Schema Versioning
How to improve performance of series of data	Bucket

Other Pattern

Mixed Attributes*

- using key/values in arrays for allow searching on dozens of variable fields

Approximation*

- reducing frequency of calculations with approximate values

Trees

- store 1 or multiple levels as one document and/or use $graphLookup to recursively traverse

Polymorphism

- each document represents an item, but each item can have different fields (e.g. product catalog)

Outlier*

- avoid having a few documents drive the design, and impact performance for all

* = covered in other presentations on Mongodb.com

Use Case Categories

Patterns	Catalog	Content Management	Internet of Things	Mobile	Personalization	Real-Time Analytics	Single View
Approximation	✅		✅	✅		✅	
Attribute	✅	✅					✅
Bucket			✅			✅	
Computed	✅		✅	✅	✅	✅	✅
Document Versioning	✅	✅			✅		✅
Extended Reference	✅			✅		✅	
Outlier			✅	✅	✅		
Preallocated			✅			✅	
Polymorphic	✅	✅		✅			✅
Schema Versioning	✅	✅	✅	✅	✅	✅	✅
Subset	✅	✅		✅	✅		
Tree and Graph	✅	✅					

Case Study

Content Management System

Building a CMS with MongoDB

There are many tables for this example, with multiple queries required for every page load.

Potential tables

- article
- author
- comment
- tag
- link_article_tag
- link_article_article (related articles)
- etc.

Building a CMS with MongoDB

Sample JSON

```
{
  "_id" : 334456,
  "slug" : "/apple-reports-second-quarter-revenue",
  "headline" : "Apple Reported Second Quarter Revenue Today",
  "date" : ISODate("2015-03-24T22:35:21.908Z"),
  "author" : {
          "name" : "Bob Walker",
          "title" : "Lead Business Editor"
  },
  "copy" : "Apple beat Wall St expectations by reporting ...",
  "tags" : ["AAPL", "Earnings", "Cupertino"],
  "comments" : [
          { "name" : "Frank", "comment" : "Great Story", "date" : ISODate(...) },
          { "name" : "Wendy", "comment" : "+1", "date" : ISODate(...) }
  ]
}
```

Building a CMS with MongoDB

Benefits of the Relational Design
With Normalized Data:

- Updates to author information are inexpensive
- Updates to tag names are inexpensive

Benefits of the Design with MongoDB

- Much faster reads
- One query to load a page
- The relational model would require multiple queries and/or many joins.

Every System has Tradeoffs

- Relational design will provide more efficient writes for some data.
- MongoDB design will provide efficient reads for common query patterns.
- A typical CMS may see 1000 reads (or more) for every article created (write).

Building a CMS with MongoDB

Optimizing comments

- What happens when an article has one million comments?

Include more information associated with each tag

Building a CMS with MongoDB

Optimizing Comments Option 1

Changes:

- Include only the last N comments in the “main” document.
- Put all other comments into a separate collection
  - One document per comment

Considerations:

- How many comments are shown on the first page of an article?
  - This example assumes 10.
- What percentage of users click to read more comments?

```
{
  "_id": 334456,
  "slug": "/apple-reports-second-quarter-revenue",
  "headline": "Apple Reported Second Quarter Revenue
Today",
...
  "last_10_comments": [
   {
          "name": "Frank",
          "comment": "Great Story",
          "date": ISODate()
   },
   {
          "name": "Wendy",
          "comment": "When can I buy an Apple Watch?",
          "date": ISODate()
   }
  ]
}
```

Building a CMS with MongoDB

Optimizing Comments Option 2

Changes:

- Use a separate collection for comments, one document per comment.

Considerations:

- Now every page load will require at least 2 queries
- But adding new comments is less expensive than for Option 1.
- And adding a new comment is an atomic operation

```
> db.comments.insertOne({
"article_id": 334456,
"name " : "Frank ",
"comment": "Great Story",
"date": ISODate()
})
```

Building a CMS with MongoDB

Include More information With Each Tag

Considerations:

- Make each tag a document with multiple fields

```
{
"_id" : "/apple-reports-second-quarter-revenue",
...
"tags" : [
  { "type" : "ticker", "label" : "AAPL" },
  { "type" : "financials", "label" : "Earnings" },
  { "type" : "location", "label" : "Cupertino" }
]

> db.article.find({
    "tags": {
      "$elemMatch": {
        "type": "financials",
        "label": "Earnings"
      }
    }
})
```

Case Study

Social Network

Social Network

Design Considerations

- User relationships(followers, followees)
- Newsfeed requirements

What are the problems with the following approach?

```
db.users.find()
{
 "_id": "bigbird",
 "fullname": "Big Bird",
 "followers": [
        "oscar",
        "elmo"
        ],
        "following": [
        "elmo",
        "bert"
        ],
...
}
```

Social Network

Relationships must be split into separate documents:

- This will provide performance benefits.
- Other motivations:

  - Some users (e.g., celebrities) will have millions of followers.
  - Embedding a “followers” array would literally break the app: documents are limited to 16 MB.
  - Different types of relationships may have different fields and requirements.

```
> db.followers.find()
{ "_id" : ObjectId(), "user" : "bigbird", "following" : "elmo" }
{ "_id" : ObjectId(), "user" : "bigbird", "following" : "bert" }
{ "_id" : ObjectId(), "user" : "oscar", "following" : "bigbird" }
{ "_id" : ObjectId(), "user" : "elmo”", "following" : "bigbird" }
```

Social Network

Now meta-data about the relationship can be added:

```
db.followers.find() {
  "_id": ObjectId(),
  "user": "bigbird",
  "following": "elmo",
  "group": "work",
  "follow_start_date": ISODate("2015-05-19T06:01:17.171Z")
}
```

Social Network

Counting User Relationships

- Counts across a large number of documents may be slow
  - Option: maintain an active count in the user profile

- An active count of followers and followees will be more expensive for creating relationships
  - Requires an update to both user documents (plus a relationship document) each time a relationship is changed
  - For a read-heavy system, this cost may be worth paying

```
db.users.find()
{
  "_id": "bigbird",
  "fullname": "Big Bird",
  "followers": 2,
  "following": 2,
...
}
```

Social Network

Index needed on (followers.user, followers.following)

For reverse lookups, index needed on (followers.following, followers.user)

Indexes should be used for these graph lookups

May also want to maintain two separate collections: followers, followees

Case Study

Internet of Things

Data Modeling - Time Series

One Document per measurement

Things to Consider

- Relational approach?
- What happens if we scale?
- Data & Index Size?

```
{
        sensor_id :  12345,
        timestamp : ISODATE("2018-12-11T12:00:00.000Z"),
        temperature : 65,
        moisture : 546
}
{
        sensor_id : 12345,
        timestamp : ISODATE("2018-12-11T12:01:00.000Z"),
        temperature : 65,
        moisture : 651
}

{
        sensor_id : 12345,
        timestamp : ISODATE("2018-12-11T12:02:00.000Z"),
        temperature : 66,
        moisture : 828
}
```

Data Modeling - Time Series

Bucketing

Things to Consider

- Relational approach?
- What happens if we scale?
- Data & Index Size?
- Size of Document?

```
{
 sensor_id :  12345,
 measurements : [
   {
      date : ISODATE("2018-12-11T12:00:00.000Z"),
      temperature : 65,
      moisture : 546
   },
  {
      date : ISODATE("2018-12-11T12:01:00.000Z"),
      temperature : 65,
      moisture : 651
   },
  {
      date : ISODATE("2018-12-11T12:02:00.000Z"),
      temperature : 66,
      moisture : 828
   },
 …
 ],
}
```

Data Modeling - Time Series

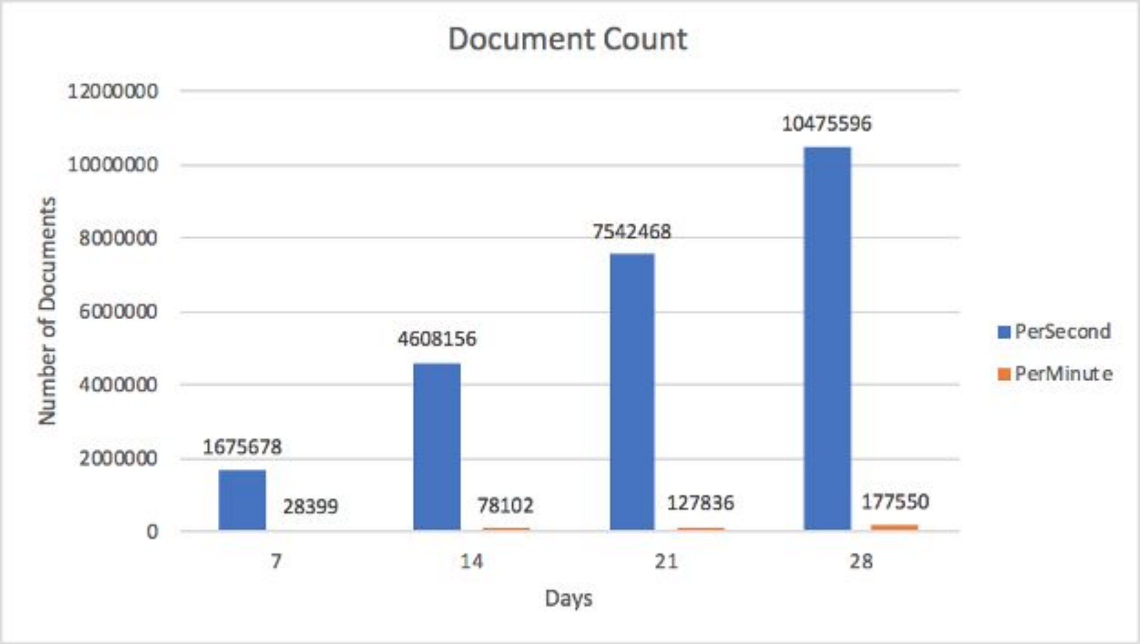
Bucketing by Time & Transactions

Things to Consider

- Relational approach?
- What happens if we scale?
- Data & Index Size?
- Size of Document?
- How will users access data?

```
{
 sensor_id :  12345,
 start_date : ISODATE("2018-12-11T12:00:00.00Z"),
 end_date : ISODATE("2018-12-11T13:00:00.00Z"),
 measurements : [
   {
     date : ISODATE("2018-12-11T12:00:00.000Z"),
     temperature : 65,
     moisture : 546
   },
  {
     date : ISODATE("2018-12-11T12:01:00.000Z"),
     temperature : 65,
     moisture : 651
   },
  …
 ],
txCount : 60,
…
 }
```

Effects on Document Count



Per Second vs Per Minute

Effects on Memory

Per Second vs Per Minute

Data Modeling - Time Series

\+ **Pre-Aggregation**

Things to Consider

- Relational approach?
- What happens if we scale?
- Data & Index Size?
- Size of Document?
- How will users access data?
- How will Data Scientists access the data?

```
{
 sensor_id :  12345,
 start_date : ISODATE("2018-12-11T12:00:00.00Z"),
 end_date : ISODATE("2018-12-11T13:00:00.00Z"),
 measurements : [
  {
  date : ISODATE("2018-12-11T12:00:00.000Z"),
  temperature : 65,
  moisture : 546
  },
  {
  date : ISODATE("2018-12-11T12:01:00.000Z"),
  temperature : 65,
  moisture : 651
  },
 …
 ],
txCount : 60,
 sum_temp : 4020,
 sum_moisture : 47700,
…
}
```

Average Temp = 4020 / 60 = 67

Case Study

Time Series Data

Building a Database Monitoring tool

Monitor hundreds of thousands of database servers

Ingest metrics every 1-2 seconds

Scale the system as new database servers are added

Provide real-time graphs and charts to users

Building a Database Monitoring tool

RDBMS row client “1234”, recording 50k database operations, at 2015-05-29(23:06:37):

```
"clientid" (integer): 1234
"metric (varchar): "op_counter"
"value" (double): 50000
"timestamp" (datetime): 2015-05-29T23:06:37.000Z
```

```
{
"clientid": 1234,
"metric": "op_counter",
"value": 50000,
"timestamp": ISODate("2015-05-29T23:06:37.000Z")
}
```

Building a Database Monitoring tool

Problems With This Design

- Aggregations become slower over time, as database becomes larger
- Asynchronous aggregation jobs won't provide real-time data
- We aren't taking advantage of other MongoDB data types

Solution:

- Storing one document per hour

- 1 minute granularity

```
{
"clientid" : 1234,
"timestamp":
ISODate("2015-05-29T23:06:00.000Z"),
"metric": "op_counter",
"values": {
        0: 0,
        ...
        37: 50000,
        ...
        59: 2000000
        }
}
```

Building a Database Monitoring tool

Update the exact minute in the hour where the op_counter was recorded:

```
db.metrics_by_minute.updateOne( {
"clientid" : 1234,
"timestamp": ISODate("2015-05-29T23:06:00.000Z"),
"metric": "op_counter"},
{ $set : { "values.37" : 50000 } })
```

Increment the counter for the exact minute in the hour where the op_counter metric was recorded:

```
db.metrics_by_minute.updateOne( {
"clientid" : 1234,
"timestamp": ISODate("2015-05-29T23:06:00.000Z"),
"metric": "insert"},
{ $inc : { "values.37" : 50000 } })
```

Building a Database Monitoring tool

Condensing a Day's Worth of Metric Data Into a Single Document

With one minute granularity, we can record a day's worth of data and update it efficiently with the following structure. (values.<HOUR_IN_DAY>.<MINUTE_IN_HOUR>)

```
{
 "clientid" : 1234,
 "timestamp": ISODate("2015-05-29T00:00:00.000Z"),
 "metric": "insert",
 "values": {
        "0": { 0: 123, 1: 345, ..., 59: 123},
        ...
        "23": { 0: 123, 1: 345, ..., 59: 123}
        }
}
```

Recap

Container Types in a collection define embedded (documents) or linked (arrays) models

Schema design focus on the application needs (Payload vs Process fields)

Design patterns can help on the schema design based on use cases

Different Schema Design patterns are:

Attribute pattern, Bucket pattern, Computed pattern, Versioning pattern, Subset pattern, other many patterns

Thank You!