

MongoDB Schema Design:

MongoDB schema design works a lot differently than relational schema design. With MongoDB schema design, there is:

- No formal process
- No algorithms
- No rules

When you are designing your MongoDB schema design, the only thing that matters is that you design a schema that will work well for *your* application. Two different apps that use the same exact data might have very different schemas if the applications are used differently.

MongoDB Data Modelling:

- Data modeling is the process that creates a visual representation of the data components in a system. This visual representation not only helps identify all data components, but also helps determine the relationships among data elements while finding the best way to demonstrate those relationships.
- Data in MongoDB has a flexible schema documents in the same collection. They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.
- Two Types of Data Models can be used: **1. Embedded Data Model, 2. Reference Data Model**

1. Embedded Data Model:

- In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.
- Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document.

```
{  
  _id: <ObjectId>,  
  username: "123xyz",  
  contact: {  
    phone: "123-456-7890",  
    email: "xyz@example.com"  
  },  
  access: {  
    level: 5,  
    group: "dev"  
  }  
}
```



Embedded sub-
document



Embedded sub-
document

Example:

```
{  _id: ,  
  Emp_ID: "10025AE336"  
  Personal_details: {First_Name: "Radhika", Last_Name: "Sharma", Date_Of_Birth: "1995-09-26"},  
  Contact: {e-mail: "radhika_sharma.123@gmail.com", phone: "9848022338"}, //Embedded Sub Documents  
  Address: {city: "Hyderabad", Area: "Madapur", State: "Telangana"}  
}
```



Limitations of Embedded Data Modelling:


- There is a **16-MB document size limit in MongoDB**
- If you are embedding too much data inside a single document, you could potentially hit this limit.



2. Normalized Data Model or Reference Model :

- Okay, so the other option for designing our schema is referencing another document using a document's **unique object ID** and connecting them together using the **\$lookup** operator. Referencing works similarly as the JOIN operator in an SQL query. It allows us to split up data to make more efficient and scalable queries, yet maintain relationships between data.

Advantages

- By splitting up data, you will have smaller documents.
 - Less likely to reach **16-MB-per-document limit**.
 - Infrequently accessed information not needed on every query.
 - Reduce the amount of duplication of data. However, it's important to note that data duplication should not be avoided if it results in a better schema.
- 

user document

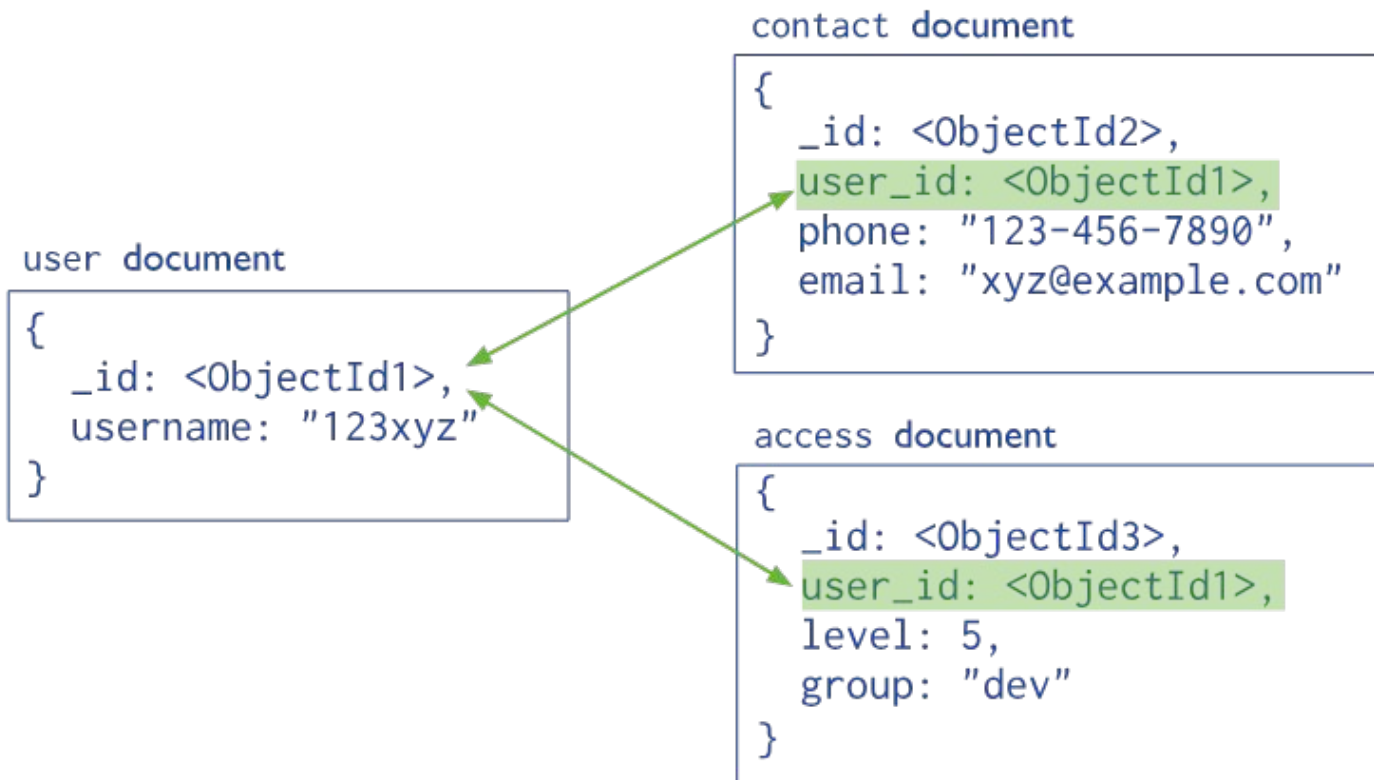
```
{  
  _id: <ObjectId1>,  
  username: "123xyz"  
}
```

contact document

```
{  
  _id: <ObjectId2>,  
  user_id: <ObjectId1>,  
  phone: "123-456-7890",  
  email: "xyz@example.com"  
}
```

access document

```
{  
  _id: <ObjectId3>,  
  user_id: <ObjectId1>,  
  level: 5,  
  group: "dev"  
}
```



In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
  _id: <ObjectId101>,
  Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
  _id: <ObjectId102>,
  empDocID: " ObjectId101",
  First_Name: "Radhika",
  Last_Name: "Sharma",
  Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{  
  _id: <ObjectId103>,  
  empDocID: " ObjectId101",  
  e-mail: "radhika_sharma.123@gmail.com",  
  phone: "9848022338"  
}
```

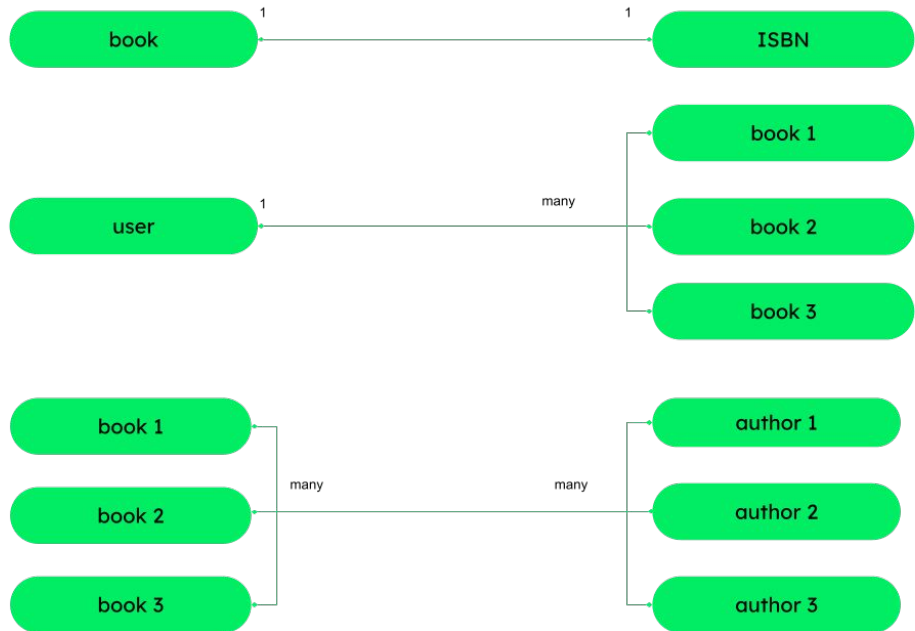
Address:

```
{  
  _id: <ObjectId104>,  
  empDocID: " ObjectId101",  
  city: "Hyderabad",  
  Area: "Madapur",  
  State: "Telangana"  
}
```

MongoDB supports multiple ways to model

relationships between entities:

- One to one (1-1): In this relationship, one value is associated with only one document. For example, a book can have only one ISBN.
- One to many (1-N): Here, one value can be associated with more than one document or value. For example, a user can borrow more than one book at a time.
- Many to many (N-N): In this type of model, multiple documents can be associated with each other. For example, a book can have many authors, and one author can write many different books. The relationship between author and book is many to many.



What does the process of data modeling look like?

The data modeling process is a series of steps taken to create one of the data models described above. These steps include:

Step -1: Gathering requirements:

The first step in the data modeling process is to gather all requirements for the application. This step helps uncover the underlying data structure that needs to be reviewed and it's important to not only analyze the data objects, but also the amount of data and the operations that will be performed on that data.

Books: The library has millions of books, and they all have a unique ISBN. The users will also need to search books by title or by author.

Users: This library has thousands of users and each user has a name and address. The library will also assign each user a unique number found on their library card.

Step - 2: Understand entity relationships:

Understanding how these data entities (e.g., book ISBNs, user names) will interact with each other is also key. These interactions will comprise the relationships in your model.

Interaction example:

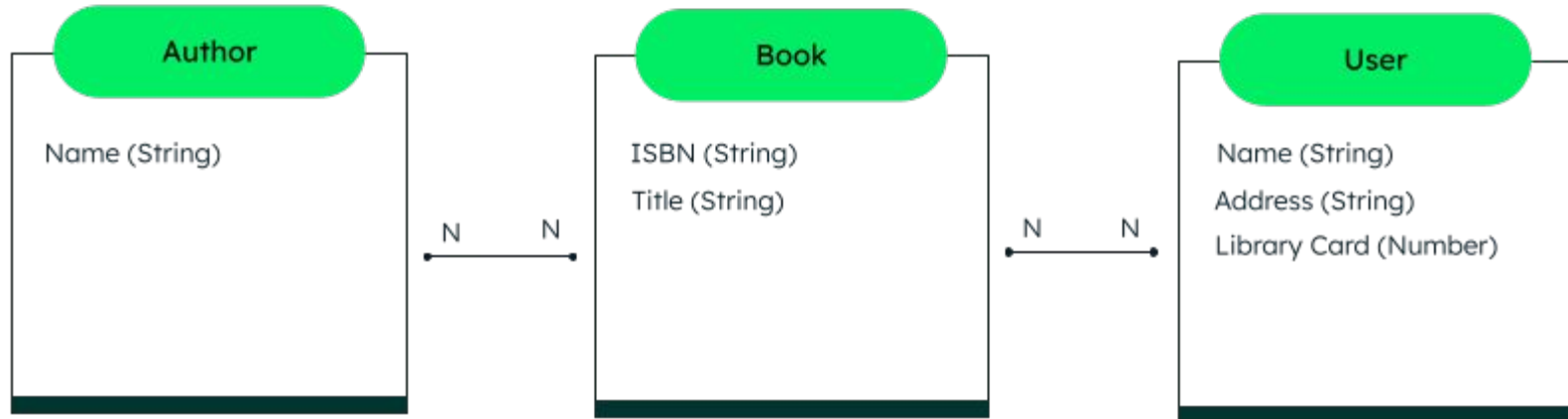
- Users borrow books: The library will need to know which books have been borrowed by which user. Each user is entitled to five borrowed books at a time.

These business rules enable organization of the information needed to build the conceptual model as you now understand the data necessary to build the first software iteration.




Step - 3: Identify the data structure:

It's now time to create a logical data model. As part of this modeling step, you might realize that some data structures are more complex and require new entities. For example, the author names may be better represented as their own entities in order to enable searching for books by author.



Step -4 Apply Data Modelling Patterns:

- Now you're ready to choose your DBMS and build your physical data model. At this point, thinking in terms of the type of database chosen will determine how the data is stored.
 - For example, if using a document database, such as MongoDB, you'll model relationships using embedding or document references. As you establish the relationships between various objects, you'll also find the IDs and unique values representing your items.
 - Returning to our library example in the following diagram, you can see that "author" was embedded in the "books." This will make it easier to create indexes.
- 

Book

```
{  
  ISBN: String,  
  Title: String,  
  Author: {  
    FirstName: String,  
    LastName: String  
  }  
}
```

N

1

User

```
{  
  CardNum: Number,  
  Name: String,  
  Address: {  
    Street: String,  
    City: String  
  },  
  BooksBorrowed: [{  
    ISBN: String,  
    Title: String,  
    DueDate: Date  
  }]  
}
```

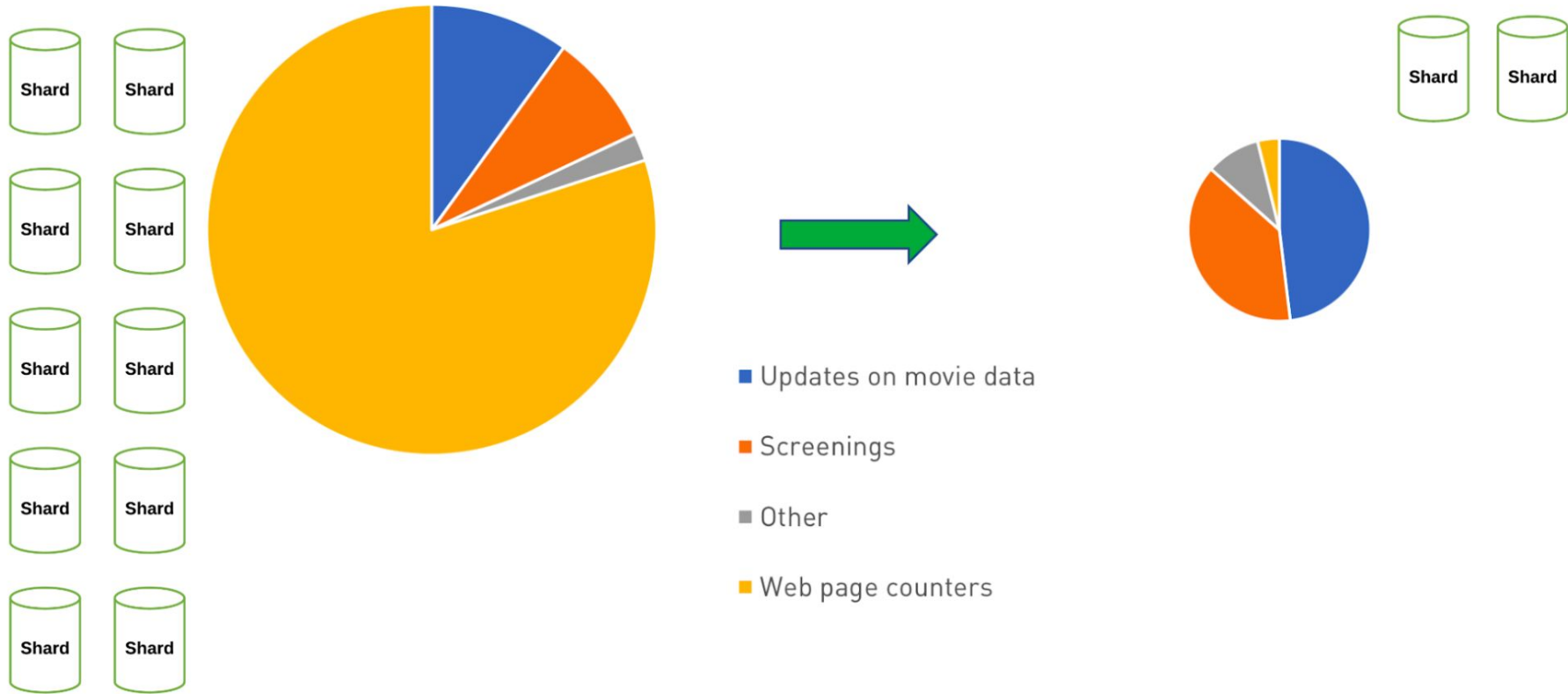

Data Modelling Patterns:

1. Approximation Pattern
2. Attribute pattern:
3. Bucket Pattern
4. Computed Pattern
5. Document Versioning
6. Extended Versioning
7. Pre-allocated
8. Polymorphic
9. Schema Versioning
10. Tree and Graph

4.1 Approximation Pattern:

- The Approximation Pattern is useful when expensive calculations are frequently done and when the precision of those calculations is not the highest priority.
- Example: Population patterns are an example of the *Approximation Pattern*. An additional use case where we could use this pattern is for website views. Generally speaking, it isn't vital to know if 700,000 people visited the site, or 699,983. Therefore we could build into our application a counter and update it in the database when our thresholds are met.
- This could have a tremendous reduction in the performance of the site. Spending time and resources on business critical writes of data makes sense. Spending them all on a page counter doesn't seem to be a great use of resources.






In the image above we see how we could use the *Approximation Pattern* and reduce not only writes for the counter operations, but we might also see a reduction in architecture complexity and cost by reducing those writes. This can lead to further savings beyond just the time for writing data.

4.2 Attribute Pattern:

- The **Attribute Pattern** is useful for problems that are based around having big documents with many similar fields but there is a subset of fields that share common characteristics and we want to sort or query on that subset of fields.
- When the fields we need to sort on are only found in a small subset of documents. Or when both of those conditions are met within the documents.

Pros

- Fewer indexes are needed.
 - Queries become simpler to write and are generally faster.
- 

- Let's think about a collection of movies. The documents will likely have similar fields involved across all of the documents: title, director, producer, cast, etc. Let's say we want to search on the release date.
- A challenge that we face when doing so, is *which* release date? Movies are often released on different dates in different countries.

```
{  
  title: "Star Wars",  
  director: "George Lucas", ...  
  release_US: ISODate("1977-05-20T01:00:00+01:00"),  
  release_France: ISODate("1977-10-19T01:00:00+01:00"),  
  release_Italy: ISODate("1977-10-20T01:00:00+01:00"),  
  release_UK: ISODate("1977-12-27T01:00:00+01:00"), ...  
}
```

- By using the Attribute Pattern, we can move this subset of information into an array and reduce the indexing needs. We turn this information into an array of key-value pairs:

```
{  
  title: "Star Wars",  
  director: "George Lucas", ...  
  releases: [  
    { location: "USA", date: ISODate("1977-05-20T01:00:00+01:00") },  
    { location: "France", date: ISODate("1977-10-19T01:00:00+01:00") },  
    { location: "Italy", date: ISODate("1977-10-20T01:00:00+01:00") },  
    { location: "UK", date: ISODate("1977-12-27T01:00:00+01:00") }, ...  
  ], ...  
}
```

Indexing becomes much more manageable by creating one index on the elements in the array:

```
{ "releases.location": 1, "releases.date": 1 }
```

4.3 Bucket Pattern:

The **Bucket Pattern** is a great solution for when needing to manage streaming data, such as time-series, real-time analytics, or Internet of Things (IoT) applications.

Pros

- Reduces the overall number of documents in a collection.
- Improves index performance.
- Can simplify data access by leveraging pre-aggregation.



- With data coming in as a stream over a period of time (time series data) we may be inclined to store each measurement in its own document.
- However, this inclination is a very relational approach to handling the data.
- If we have a sensor taking the temperature and saving it to the database every minute, our data stream might look something like:

```
{ 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 }
```

- We can "bucket" this data, by time, into documents that hold the measurements from a particular time span.

Taking the data stream from above and applying the Bucket Pattern to it, we would wind up with:

```
{ 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 }
```

- When working with time-series data it is frequently more interesting and important to know what the average temperature was from 2:00 to 3:00 pm in Corning, California on 13 July 2018 than knowing what the temperature was at 2:03 pm.
- By bucketing, we're more able to easily provide that information.

4.4 Computed Pattern:

- When there are very read intensive data access patterns and that data needs to be repeatedly computed by the application.
- The Computed Pattern is also utilized when the data access pattern is read intensive; for example, if you have 1,000,000 reads per hour but only 1,000 writes per hour, doing the computation at the time of a write would divide the number of calculations by a factor 1000.



- The *Computed Pattern* can be utilized wherever calculations need to be run against data.
- Datasets that need sums, such as revenue or viewers, are a good example, but time series data, product catalogs, single view applications, and event sourcing are prime candidates for this pattern too.
- This is a pattern that many customers have implemented. For example, a customer does massive aggregation queries on vehicle data and store the results for the server to show the info for the next few hours.
- A publishing company compiles all kind of data to create ordered lists like the "100 Best ...". Those lists only need to be regenerated once in a while, while the underlying data may be updated at other times.


4.5 Document Versioning:

When you are faced with the need to maintain previous versions of documents in MongoDB, the **Document Versioning** pattern is a possible solution.

Pros

- Easy to implement, even on existing systems.
- No performance impact on queries on the latest revision.

Cons

- Doubles the number of writes.
 - Queries need to target the correct collection.
- 

- Financial and healthcare industries are good examples. Insurance and legal industries are some others. There are many use cases that track histories of some portion of the data.
- Think of how an insurance company might make use of this pattern. Each customer has a “standard” policy and a second portion that is specific to that customer, a policy rider if you will.
- This second portion would contain a list of policy add-ons and a list of specific items that are being insured.
- As the customer changes what specific items are insured, this information needs to be updated while the historical information needs to be available as well. This is fairly common in homeowner or renters insurance policies.
- For example, if someone has specific items they want to be insured beyond the typical coverage provided, they are listed separately, as a rider.
- Another use case for the insurance company may be to keep all the versions of the "standard policy" they have mailed to their customers over time.

Inside our database, each customer might have a `current_policy` document — containing customer specific information — in a `current_policies` collection and `policy_revision` documents in a `policy_revisions` collection. Additionally, there would be a `standard_policy` collection that would be the same for most customers. When a customer purchases a new item and wants it added to their policy, a new `policy_revision` document is created using the `current_policy` document. A `version` field in the document is then incremented to identify it as the latest revision and the customer's changes added.

```
{  
  _id: ObjectId<ObjectId>,  
  name: 'Bilbo Baggins',  
  revision: 1,  
  items_insured: ['Elven-sword'],  
  ...  
}
```

Original *current_policy* document

```
{  
  _id: ObjectId<ObjectId>,  
  name: 'Bilbo Baggins',  
  revision: 1,  
  items_insured: [  
    'Elven-sword'  
  ],  
  ...  
}
```

New *policy_revision* document

```
{  
  _id: ObjectId<ObjectId>,  
  name: 'Bilbo Baggins',  
  revision: 2,  
  items_insured: [  
    'Elven-sword',  
    'One Ring',  
  ],  
  ...  
}
```

New *current_policy* document

The newest revision will be stored in the `current_policies` collection and the old version will be written to the `policy_revisions` collection. By keeping the latest versions in the `current_policy` collection queries can remain simple. The `policy_revisions` collection might only keep a few versions back as well, depending on data needs and requirements.

```
{
  _id: ObjectId<ObjectId>
  name: 'Bilbo Baggins',
  revision: 2,
  ...
}

{
  _id: ObjectId<ObjectId>
  name: 'Gandalf',
  revision: 12,
  ...
}
```

current_policies collection

```
{
  _id: ObjectId<ObjectId>
  name: 'Bilbo Baggins',
  revision: 1,
  ...
}

{
  _id: ObjectId<ObjectId>
  name: 'Gandalf',
  revision: 11,
  ...
}

{
  _id: ObjectId<ObjectId>
  name: 'Gandalf',
  revision: 10,
  ...
}

{
  _id: ObjectId<ObjectId>
  name: 'Gandalf',
  revision: 9,
  ...
}
```

policy_revisions collection

4.6 Extended Reference:

You will find the Extended Reference pattern most useful when your application is experiencing lots of JOIN operations to bring together frequently accessed data.

Pros

- Improves performance when there are a lot of JOIN operations.
- Faster reads and a reduction in the overall number of JOINS.

Cons

- Data duplication.



If an entity can be thought of as a separate "thing", it often makes sense to have a separate collection. For example, in an e-commerce application, the idea of an order exists, as does a customer, and inventory. They are separate logical entities.

Order Collection

```
{
  _id: ObjectId("507f1f77bcf86cd799439011"),
  date: ISODate("2019-02-18"),
  customer_id: 123,
  order: [
    {
      product: "widget",
      qty: 5,
      cost: {
        value: NumberDecimal("11.99"),
        currency: "USD"
      }
    }
  ]
}
```

Customer Collection

```
{
  _id: 123,
  name: "Katrina Pope",
  street: "123 Main St",
  city: "Somewhere",
  country: "Someplace",
  ...
}
```

Inventory Collection

```
{
  _id: ObjectId("507f1f77bcf86cd111111111"),
  name: "widget",
  cost: {
    value: NumberDecimal("11.99"),
    currency: "USD"
  },
  on_hand: 98325,
  ...
}
```

- One customer can have N orders, creating a 1-N relationship. From an order standpoint, if we flip that around, they have an N-1 relationship with a customer. Embedding all of the information about a customer for each order just to reduce the JOIN operation results in a lot of duplicated information. Additionally, not all of the customer information may be needed for an order.
- The **Extended Reference** pattern provides a great way to handle these situations. Instead of duplicating all of the information on the customer, we only copy the fields we access frequently.
- Instead of embedding all of the information or including a reference to JOIN the information, we only embed those fields of the highest priority and most frequently accessed, such as **name and address**.

Customer Collection

```
{
  _id: 123,
  name: "Katrina Pope",
  street: "123 Main St",
  city: "Somewhere",
  country: "Someplace",
  date_of_birth: ISODate("1992-11-03"),
  social_handles: [
    twitter: "@somethingamazing123"
  ]
  ...
}
```

Order Collection

```
{
  _id: ObjectId("507f1f77bcf86cd799439011"),
  date: ISODate("2019-02-18"),
  customer_id: 123,
  shipping_address: {
    name: "Katrina Pope",
    street: "123 Main St",
    city: "Somewhere",
    country: "Someplace"
  },
  order: [
    {
      product: "widget",
      qty: 5,
      cost: {
        value: NumberDecimal("11.99"),
        currency: "USD"
      }
    }
  ],
  ...
}
```

Something to think about when using this pattern is that data is duplicated. Therefore it works best if the data that is stored in the main document are fields that don't frequently change. Something like a user_id and a person's name are good options. Those rarely change.

4.7 Pre-allocation Pattern:

When you know your document structure and your application simply needs to fill it with data, the **Pre-Allocation Pattern** is the right choice.

Pros

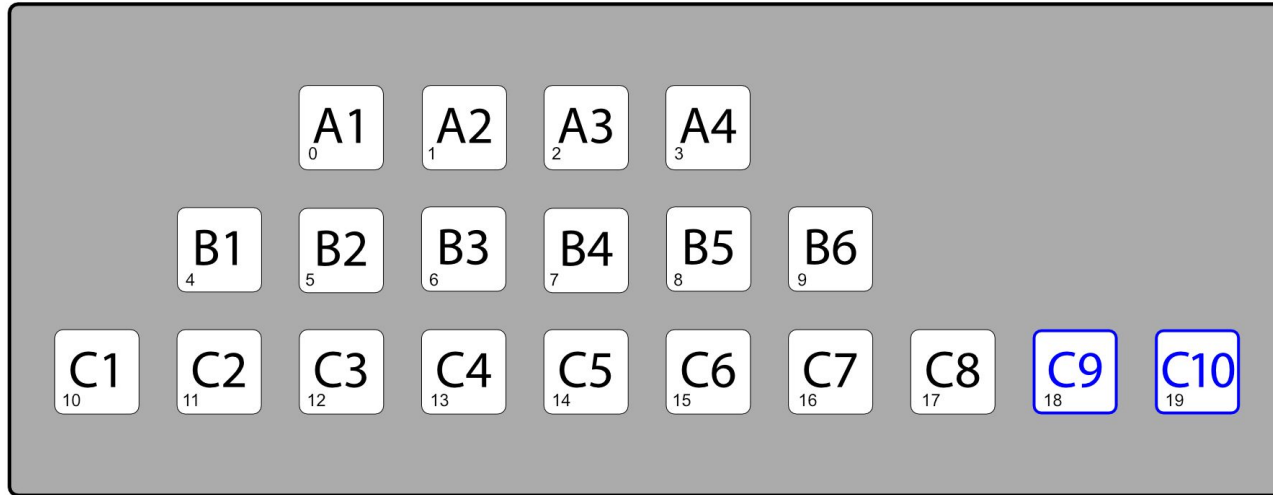
- Design simplification when the document structure is known in advance.

Cons





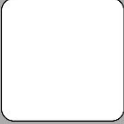

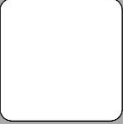
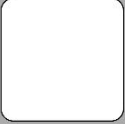
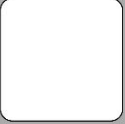

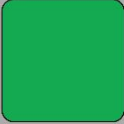
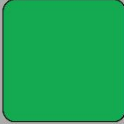
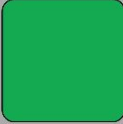
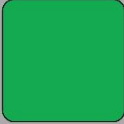
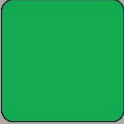
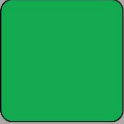
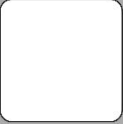
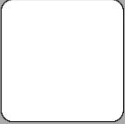
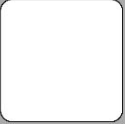
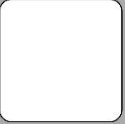
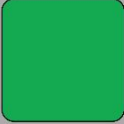
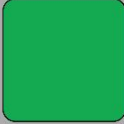
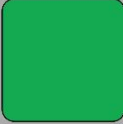
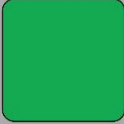
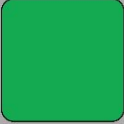
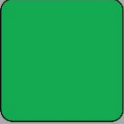
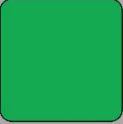
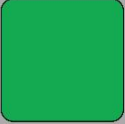

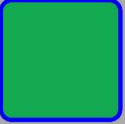
- Simplicity versus performance.



- This pattern simply dictates to create an initial empty structure to be filled later.
- Let's say there is a need to represent a theater room as a 2-dimensional array where each seat has a "row" and "number", for example, the seat "C7".
- Some rows may have fewer seats, however finding the seat "B3" is faster and cleaner in a 2-dimensional array, than having a complicated formula to find a seat in a one-dimensional array that has only cells for the existing seats.
- Being able to identify accessible seating is also easier as a separate array can be created for those seats.



One dimensional representation of venue, accessible seats shown in blue.

	1	2	3	4	5	6	7	8	9	10
A										
B										
C										

*Two dimensional representation of venue, valid seats available in green.
Accessible seating notated with a blue outline.*

Another example could be a reservation system where a resource is blocked or reserved, on a per day basis.

4.8 Polymorphic Pattern:

The **Polymorphic Pattern** is the solution when there are a variety of documents that have more similarities than differences and the documents need to be kept in a single collection.

Pros

- Easy to implement.
- Queries can run across a single collection.



- When all documents in a collection are of similar, but not identical, structure, we call this the Polymorphic Pattern. As mentioned, the Polymorphic Pattern is useful when we want to access (query) information from a single collection. Grouping documents together based on the queries we want to run (instead of separating the object across tables or collections) helps improve performance.
- Imagine that our application tracks professional sports athletes across all different sports.

Common fields

```
{  
  "sport": "ten_pin_bowling",  
  "athlete_name": "Earl Anthony",  
  "career_earnings": {value: NumberDecimal("1441061"), currency: "USD"},  
  "300_games": 25,  
  "career_titles": 43,  
  "other_sports": "baseball"  
}  
  
{  
  "sport": "tennis",  
  "athlete_name": "Martina Navratilova",  
  "career_earnings": {value: NumberDecimal("216226089"), currency: "USD"},  
  "event": {  
    "type": "singles",  
    "career_tournaments": 390,  
    "career_titles": 167  
  }  
}
```

Professional athlete records have some similarities, but also some differences. With the Polymorphic Pattern, we are easily able to accommodate these differences.

- If we were not using the Polymorphic Pattern, we might have a collection for **Bowling Athletes** and a collection for **Tennis Athletes**.
- When we wanted to query on all athletes, we would need to do a time-consuming and potentially complex join.
- Instead, since we are using the Polymorphic Pattern, all of our data is stored in one Athletes collection and querying for all athletes can be accomplished with a simple query.
- This design pattern can flow into embedded sub-documents as well. In the previous example, Martina Navratilova didn't just compete as a single player, so we might want to structure her record as follows:

```
{
  "sport": "tennis",
  "athlete_name": "Martina Navratilova",
  "career_earnings": {value: NumberDecimal("216226089"), currency: "USD"},
  "career_tournaments": 390,
  "career_titles": 167,
  "event": [ {
    "type": "singles",
    "career_tournaments": 390,
    "career_titles": 167
  },
  {
    "type": "doubles",
    "career_tournaments": 233,
    "career_titles": 177,
    "partner": ["Tomanova", "Fernandez", "Morozova", "Evert", ...]
  },
  ...
}
```

Polymorphic Sub-Documents

- One example use case of the Polymorphic Pattern is Single View applications.
- Imagine working for a company that, over the course of time, acquires other companies with their technology and data patterns.
- For example, each company has many databases, each modeling "insurances with their customers" in a different way.
- Then you buy those companies and want to integrate all of those systems into one. Merging these different systems into a unified SQL schema is costly and time-consuming.

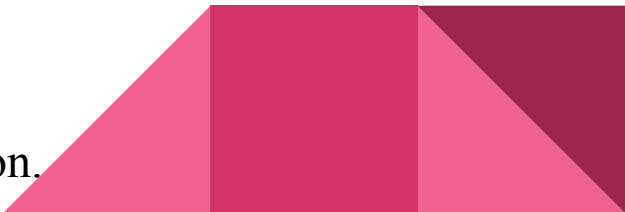
4.9 Schema Versioning:

Just about every application can benefit from the **Schema Versioning Pattern** as changes to the data schema frequently occur in an application's lifetime. This pattern allows for previous and current versions of documents to exist side by side in a collection.

Pros

- No downtime needed.
- Control of schema migration.
- Reduced future technical debt.

Cons

- Might need two indexes for the same field during migration.
- 

As stated, just about every database needs to be changed at some point during its lifecycle, so this pattern is useful in many situations. Let's take a look at a customer profile use case. We start keeping customer information before there is a wide range of contact methods. They can only be reached at home or at work:

```
{  
  "_id": "<ObjectId>",  
  "name": "Anakin Skywalker",  
  "home": "503-555-0000",  
  "work": "503-555-0010"  
}
```

As the years go by and more and more customer records are being saved, we notice that mobile numbers are needing to be saved as well. Adding that field in is straight forward.

```
{  
  "_id": "<ObjectId>",  
  "name": "Darth Vader",  
  "home": "503-555-0100",  
  "work": "503-555-0110",  
  "mobile": "503-555-0120"  
}
```

More time goes by and now we're discovering that fewer and fewer people have a home phone, and other contact methods are becoming more important to record. Items like Twitter, Skype, and Google Hangouts are becoming more popular and maybe weren't even available when we first started keeping contact information.

In doing so, we create a new schema version.

```
{  
  "_id": "<ObjectId>",  
  "schema_version": "2",  
  "name": "Anakin Skywalker (Retired)",  
  "contact_method":  
  [  
    { "work": "503-555-0210" }, { "mobile": "503-555-0220" }, { "twitter": "@anakinskywalker" }, { "skype":  
    "AlwaysWithYou" }  
  ]  
}
```



4.10 Tree Pattern:

- When data is of a hierarchical structure and is frequently queried, the Tree Pattern is the design pattern to implement.
- There are many ways to represent a tree in a legacy tabular database. The most common ones are for a node in the graph to list its parent and for a node to list its children. Both of these representations may require multiple access to build the chain of nodes.
- MongoDB provides the \$graphLookup operator to navigate the data as graphs

Pros

- Increased performance by avoiding multiple JOIN operations.

Cons

- Updates to the graph need to be managed in the application.
- 

Example:

employee_id	employee_name	position	reports_to
1	David Wallace	CEO	
2	Jan Levinson	VP, NE Sales	1
3	Michael Scott	Regional Manager	2
4	Dwight Schrute	Sales Rep	3
5	Jim Halpert	Sales Rep	3
6	Pam Beesly	Receptionist	3

Corporate structure with Parent nodes

employee_id	employee_name	position	direct_reports
1	David Wallace	CEO	2
2	Jan Levinson	VP, NE Sales	3
3	Michael Scott	Regional Manager	4
3	Michael Scott	Regional Manager	5
3	Michael Scott	Regional Manager	6
4	Dwight Schrute	Sales Rep	
5	Jim Halpert	Sales Rep	
6	Pam Beesly	Receptionist	

Corporate structure with Child nodes

In this case, we'd basically be storing the "parents" for each node. In a tabular database, it would likely be done by encoding a list of the parents. The approach in MongoDB is to simply represent this as an array.

```
{  
  employee_id: 5,  
  name: "Jim Halpert",  
  reports_to: [  
    "Michael Scott",  
    "Jan Levinson",  
    "David Wallace"  
  ]  
}
```

One-to-One:

Let's take a look at our User document. This example has some great one-to-one data in it. For example, in our system, one user can only have one name. So, this would be an example of a one-to-one relationship. We can model all one-to-one data as key-value pairs in our database.

```
{  
  "_id": "ObjectId('AAA')",  
  "name": "Joe Karlsson",  
  "company": "MongoDB",  
  "twitter": "@JoeKarlsson1",  
  "twitch": "joe_karlsson",  
  "tiktok": "joekarlsson",  
  "website": "joekarlsson.com"  
}
```

One-to-Few

Okay, now let's say that we are dealing a small sequence of data that's associated with our users. For example, we might need to store several addresses associated with a given user. It's unlikely that a user for our application would have more than a couple of different addresses. For relationships like this, we would define this as a one-to-few relationship.

```
{
  "_id": "ObjectId('AAA')",
  "name": "Joe Karlsson",
  "company": "MongoDB",
  "twitter": "@JoeKarlsson1",
  "twitch": "joe_karlsson",
  "tiktok": "joekarlsson",
  "website": "joekarlsson.com",
  "addresses": [
    { "street": "123 Sesame St", "city": "Anytown", "cc": "USA" },
    { "street": "123 Avenue Q", "city": "New York", "cc": "USA" }
  ]
}
```

One-to-Many Relationship with Embedded Documents:

- Using embedded documents we can create one-to-many relationships between the data so, that we can easily retrieve data using few read operations.
- Now we will discuss the one-to-many relationship with embedded documents with the help of an example.
- Sometimes, there are possibilities of a person containing multiple addresses like having a current address (the place where he/she stays) and the residential address (the place where he/she having own house or permanent address).
- During that time, one to many relationship possibilities occurs. So, we can use an embedded document model to store permanent and current address in a single document.

// Student document

```
{   StudentName: ABC, StudentId: g_f_g_1209, Branch:CSE }
```

// Permanent Address document

```
{   StudentName: ABC, PermanentAddress: XXXXXXXX, City: Delhi, PinCode:202333 }
```

// Current Address document

```
{   StudentName: ABC, CurrentAddress: XXXXXXXX, City: Mumbai, PinCode:334509 }
```

Now instead of writing three documents, we can embed them into a single document.

// Student document

```
{
  StudentName: ABC, StudentId: g_f_g_1209, Branch:CSE,
  Address: [
    {
      StudentName: ABC,
      PermanentAddress: XXXXXXXX, City: Delhi, PinCode:202333
    },
    {
      StudentName: ABC, CurrentAddress: XXXXXXXX, City: Mumbai,
      PinCode:334509
    }
  ]
}
```

As we keep all the data(even if there are more than 2 address kind of information, we can keep them in a JSON array) in a single collection, we can query in a single call and get a whole set of data, which leads to getting whole information and no loss occurs.

Now we will display all the addresses of the student

```
db.student.find({StudentName:"ABC"},  
  { "Address.permaAddress":1,  
    "Address.currAddress":1})
```

```
{  
  "_id" : ObjectId("60263475f19652db63812e98"),  
  "Address" : [  
    {  
      "permaAddress" : "XXXXXXXXXXXX"  
    },  
    {  
      "currAddress" : "PPPPPPPPPPPP"  
    }  
  ]  
}
```

One-to-Many Relationship with Document Reference:

We can also perform a one-to-many relationship using the document reference model. In this model, we maintain the documents separately but one document contains the reference of the other documents.

Now we will discuss the one-to-many relationship with embedded documents with the help of an example. Let us considered we have a teacher which teaches in 2 different classes. So, she has three documents:

// Teacher document

```
{  teacherName: Sunita,  
    TeacherId: g_f_g_1209,  
}
```

// Class 1 document

```
{  TeacherId: g_f_g_1209, ClassName: GeeksA, ClassId: C_123, Studentcount: 23, Subject: "Science",  
}
```

// Class 2 document

```
{  TeacherId: g_f_g_1209, ClassId: C_234, ClassName: GeeksB, Studentcount: 33, Subject: "Maths",  
}
```

Now retrieving data from these different documents is cumbersome. So, here we use the reference model which will help teacher to retrieve data using single query.

// **Teacher document**

```
{
  teacherName: Sunita,
  TeacherId: g_f_g_1209,
  classIds: [
    C_123,
    C_234
  ]
}

[> db.teacher.find().pretty()
{
  {
    "_id" : ObjectId("60263b49f19652db63812e9a"),
    "className" : "GeeksB",
    "studentCount" : 33,
    "subject" : "Maths"
  }
  {
    "_id" : ObjectId("60263b49f19652db63812e9b"),
    "className" : "GeeksA",
    "studentCount" : 23,
    "subject" : "Science"
  }
  {
    "_id" : ObjectId("60263b6bf19652db63812e9c"),
    "name" : "Sunita",
    "TeacherId" : "g_f_g_1209",
    "classId" : [
      ObjectId("60263b49f19652db63812e9a"),
      ObjectId("60263b49f19652db63812e9b")
    ]
  }
}
```

Now we will display the values of classId field

```
> db.teacher.findOne({name:"Sunita"}, {classId:1})
{
  "_id" : ObjectId("60263b6bf19652db63812e9c"),
  "classId" : [
    ObjectId("60263b49f19652db63812e9a"),
    ObjectId("60263b49f19652db63812e9b")
  ]
}
```

Many-to-Many:

- The last schema design pattern we are going to be covering in this post is the *many-to-many* relationship.
- This is another very common schema pattern that we see all the time in relational and MongoDB schema designs.
- For this pattern, let's imagine that we are building a to-do application. In our app, a user may have *many* tasks and a task may have *many* users assigned to it.
- In order to preserve these relationships between users and tasks, there will need to be references from the *one* user to the *many* tasks and references from the *one* task to the *many* users. Let's look at how this could work for a to-do list application.

USers:

```
{ "_id": ObjectId("AAF1"),  
  "name": "Kate Monster",  
  "tasks": [ObjectId("ADF9"), ObjectId("AE02"), ObjectId("AE73")]  
}
```

Tasks:

```
{ "_id": ObjectId("ADF9"),  
  "description": "Write blog post about MongoDB schema design",  
  "due_date": ISODate("2014-04-01"),  
  "owners": [ObjectId("AAF1"), ObjectId("BB3G")]  
}
```

Example:

One-to-One

User works for a
single company
(embedding)

```
User
{
  "_id": "ObjectId('AAA')",
  "name": "Joe Karlsson",
  "company": "MongoDB"
}
```

Product has
many parts
(child
referencing)

Product

```
{  
  "name": "left-handed smoke shifter",  
  "manufacturer": "Acme Corp",  
  "catalog_number": "1234",  
  "parts": [  
    "ObjectID('AAAA')",  
    "ObjectID('BBBB')",  
    "ObjectID('CCCC')"  
  ]  
}
```

Part

```
{  
  "_id" : "ObjectID('AAAA')",  
  "partno" : "123-aff-456",  
  "name" : "#4 grommet",  
  "qty": "94",  
  "cost": "0.94",  
  "price": " 3.99"  
}
```

User has many tasks and a task can be assigned to many users (mutual referencing)

User

```
{
  "_id": ObjectID("AAF1"),
  "name": "Kate Monster",
  "tasks": [
    ObjectID("ADF9"),
    ObjectID("AE02"),
    ObjectID("AE73")
  ]
}
```

Task

```
{
  "_id": ObjectID("ADF9"),
  "description": "Write blog post about schema design",
  "due_date": ISODate("2014-04-01"),
  "owners": [ObjectID("AAF1"), ObjectID("BB3G")]
}
```




Thank You