First, we need to get a brief idea about differences between MySQL and mongoDB. The major differences are known as the flexibility.

Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's collections, by default, does not require its documents to have the same schema. Which means that:

- The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure.

This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the document has substantial variation from other documents in the collection.

When we model with MySQL or also known as relational data base management system (RDBMS), there are 2 general steps, step 1 is to define the schema or as we usually do define to data model, and step 2 is to develop the application and the queries. But there are some disadvantages, or concerns. When we define our schema, we need to normalize it, so there is only 1 correct solution which means the data dictates our application. But what about the usage? We didn't take the usage into account when we develop the application, for instance how does the user interact with the database? How dose the user read or write the data? After we have normalized our application, many things would change. Then the application gets denormalized, and we have to restructure. Restructure leads to a performance drop. As the data model evolves, performance drops, which we don't want to see that happen. While in mongoDB, we have a different work sequence. We can develop the application first, so we can get an idea that this is my application I want it to do this and that, my user will need it to do this and that sort of things. When we done with the application, we then define the data model. Which means our data model is dictated by our ideas, our application and how that data is going to actually be used by the application. After that, we improve our application, and then we improve the data model and we can keep doing that last 2 steps again and again. They work complimenting each other and helping each other rather than in spite of each other.

But what are the rules to achieve that goal? So first, the data model is defined at the application level. Which means your data model should match your application but not in the reverse way. Second, your design is part of each phase of the application lifetime. As you are changing the application, you are changing the data model. You should always ask yourself what are the things that affect your data model and the changes in it? Is the data that your application needs, and the application's read and write usage of the data. Does it need more read to the data, or it needs more write to the data and how can we optimize each of them? Those are the things we need to think when we do the data modeling.

So, what are the detailed step by step for our data modeling? First, we need to evaluate the application workload. We need to know how our application is going to work, is it simulating workload or non-simulating workload. We need to consult business domain experts. Take their suggestions into account. Ask experts to help to figure out what are the current and predicted scenarios. After that, production logs and stats. Which is the actual written collected data about your data. After the evaluation, we get to know that is the data size gonna be, as well as what are the most important operations and how to rank them. We can get also get ideas about database queries and indexes we gonna use and what are the current operations and assumptions. Then, we are going to map out entities and their relationships. That is pretty much the same as we did in MySQL. Mapping entities with one-to-one, one-to-many, many-to-many or if we are going to use the embedded relations. Our last step is to finalize the data model for each collection, and we need to identify and apply relevant design patterns. After all these steps, we can have everything we need for our database. We get a list of collections with documents fields and shapes for each of these documents, we get the data size, we get the database queries and indexes and we get the current operations assumptions.

Document Structure

There are 2 types of document structures in MongoDB. The References and Embedded Data.

References store the relationships between data by including links or references from one document to another. Applications can resolve these references to access the related data. Generally speaking, these are normalized data models.

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. These denormalized data models allow applications to retrieve and manipulate related data in a single database operation.

Reference (Normalized Data)

We are pretty familiar with the reference structure, as we did a lot of exercises in MySQL. Normalized data models describe relationships using references between documents.

In general, we can use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication. Or
- when we want to represent more complex many-to-many relationships. Or
- when we want to model large hierarchical data sets.

Embedded Data

Embedded data models allow applications to store related pieces of information in the same database record. As a result, applications may need to issue fewer queries and updates to complete common operations.

In general, we can use embedded data models when:

- We have “contains” relationships between entities. Which is the same as One-to-One relationship. Or
- We have one-to-many relationships between entities. In these relationships the “many” or child documents always appear with or are viewed in the context of the “one” or parent documents.

In general, embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation. Embedded data models make it possible to update related data in a single atomic write operation.

After all these theories, we will now see some examples. If we are going to use the mongoDB to do the shopping database modeling, we can have multiple solutions. Our first solutions could be like this, we have the customers entity which have the id, name and sex attributes, and in our customer entities, we can also embed the customer contact information, as we know in MySQL the contact information must be another entity on a separate table, but in mongoDB we can simply embed it into customer as whole. And then we also have the address, phone and personal preference embedded into the contact information. We can also have a second solution as shown here, or even we can put that customers out like our third solution shown here. As we said, mongoDB have a flexible modeling design.

But what are the benefit for doing that embed design, or why should we do that. The major benefit of embedded design is that when we are finding things, in MySQL, we need a bunch of joins, to show everything we need along so many different tables. But in mongoDB with embedded design the search becomes extremely easy. We just need 1 line of code to show everything we need.

Alright, so I have done the introduction part, as a summary we can see the table shown here, comparing MySQL and MongoDB, for the steps to create the model, MySQL will need to first define the schema, and then develop application and queries. While in mongoDB we identify the queries first and then define the schema, which is the opposite sequence as we did in MySQL. For our initial schema, we have to apply the third formal form to our schema and as we said earlier data dictate our design, so there is only one possible solution. But in mongoDB, we can have many possible solutions as the example shown before. For the final schema, since we need changes during our application development, so its likely to denormalized, but since in mongoDB we have design the application first, so there will be few changes needed. For the schema evolution, it is difficult and not optimal for MySQL, and it likely to have downtime, while in mongoDB, the evolution is easy and need no downtime. Finally for our query performance, the MySQL have a relatively mediocre performance while we can always optimize our performance in mongoDB.

Our next topic is schema validation. What is schema validation? Obviously, schema validation is schema plus validation. So, let’s talk about the schema first, the database schema of a database is its structure described in a formal language supported by the database management system. The term "schema" refers to the organization of data as a blueprint of how the database is constructed. The formal definition of a database schema is a set of formulas called integrity constraints imposed on a database. These integrity constraints ensure compatibility between parts of the schema. All constraints are expressible in the same language.

Then what is validation? data validation is the process of ensuring data have undergone data cleansing to ensure they have data quality, that is, that they are both correct and useful. It uses routines, often called "validation rules", "validation constraints", or "check routines", that check for correctness, meaningfulness, and security of data that are input to the system. The rules may be implemented through the automated facilities of a data dictionary, or by the inclusion of explicit application program validation logic of the computer and its application. As a summary, validation is a way of trying to lessen the number of errors during the process of data input.

Since we now know what schema and validation is, so it’s simple, we just add validation to our data structure, so called schema validation.

In mongoDB we use jsonSchema to do the validation. Why we should use the jsonSchema but not something else? I have no ideas, so far in my opinion it is because the json file is easy to read and understand. So, jsonSchema is pretty simple, just a set of formats which is easy to understand as the sample code shown here. Cuz we are using jsonSchema, so we put the jsonSchema at the top here. After that we have the required field. Same as the MySQL, we can define which field should not be null. After that, we define each of our attributes, which is called properties here, like the sample code shown here we have a name of type string and then have a optional description to tell other people or user that this field is of type string and is required. Then why do we have a bsonType here? I don't know, in my understanding that is because json is a human readable document type, and its text parsing is very low, which means that mongoDB cannot understand the type I entered in the jsonSchema, that why we need bsonType here to let the machine know that type is it and its easier for the validation which means bson act like bridge to link the human-readable code to the machine readable code. Actually, that is reason why is was invented. In the sample shown here we can also have an object for our user input, which is another sub-schema under address.

That’s pretty straightforward.

After we have knowledge of the jsonSchema we can just put that jsonSchema into a larger scope, and now we can embed the schema when we are doing the creation of collection or we could even insert the schema afterwards. The validator here allows us to specify validation rules or expressions for a collection. The *$jsonSchema* operator matches documents that satisfy the specified JSON Schema.

The validationLevel at the bottom determines how strictly MongoDB applies the validation rules to existing documents during an update. We can have there specified levels, those are off, strict and moderate. The "off" means no validation for inserts or updates. The "strict" which is by default and apply validation rules to all inserts and all updates. The "moderate" will apply validation rules to inserts and to updates on existing valid documents, but do not apply rules to updates on existing invalid documents.

The validationAction option determines whether to ***error*** on invalid documents or just ***warn*** about the violations but allow invalid documents.