MongoDB Schema Design

There are no formal rules to design schemas in MongoDB.
The discussion in this document will therefore be over the ways in which to design the schema such that it leads to efficient CRUD operations. [1]

We need to design a schema such that it works well for our application. [1]

While designing a schema we need to keep the following in mind:

- Store the data [1]
- Provide good query performance [1]
- Require reasonable amount of hardware [1]

For every piece of data, there are 2 design choices in MongoDB:

- Embedding [1]
- Referencing [1]

Embedding refers to directly including the data in the schema as a key-value pair. [1]
Referencing refers to referencing a piece of data from another schema using the **$lookup** operator. [1]

Advantages of Embedding:

- All info can be retrieved in a single query. [1]
- We don't need to implement joins or use $lookup. [1]
- Information can be updated in a single operation. [1]
- All CRUD ops. On a single document are ACID compliant. [1]

Disadvantages:

- Large documents result in greater overhead. [1]
- A document can be of a maximum size of 16MB. [1]

Advantages of Referencing:

- Splitting up data will lead to smaller documents. [1]
- Less accessed values aren't needed in every query. [1]
- Reduces data duplication. [1]
- Having a better schema is more important than data duplication. [1]

Disadvantage:

- Using the **$lookup** operator is expensive.

Types of Relationships:

One-to-One

- These kinds of relationships can be modelled as key-value pairs.
- This kind of relationship should be embedded as a key-value pair in the document. Eg - One user has only one name. So, name: value. Eg - 1 employee can work only for 1 department.

One-To-Few

- Eg - one user has 2 addresses. Here, we should create an “addresses” key and store the different addresses in an array that is a value to that key.

The above 2 relationships and examples lead us to the first rule: **Favour Embedding unless you don’t have to.**

In other words, we **Reference** data only when we need to access that data on its own; the data is too large or it is rarely needed.

We continue below with more types of relationships.

One - To - Many

- This kind of relationship is based on the premise that 1 entity may be connected to hundreds (or more) of other entities. Eg - A product that has many parts and even more (thousands) of sub-parts. Here, we have to assign an array of ObjectIDs of the documents which have information about parts to the “parts” key. This key will be in the main document. In short, this is **Referencing**.

This leads us to Rule 2 : **Needing access to an object on its own is a compelling reason to not embed it**.

It also leads us to Rule 3: **Avoid joins/lookups if possible, but don't be afraid if they can provide a better schema design**.

One-to-Squillions

- In this kind of relationship, we have a situation where an entity is connected to millions of other entities. An example would be a logging application which could be deployed on several host servers. Each host server would create millions of log entries. It isn’t possible to store the log entries through reference as even if we referenced the log entries in the main document through ObjectIDs as that approach would breach the 16MB limit of the document.
- So, we would have to solve this problem by storing the ObjectID of the document containing the host server’s info in the document storing the log entry.

This leads us to Rule 4: **Arrays should not grow without bound. If there are more than a couple of hundred documents on the "many" side, don't embed them; if there are more than a few thousand documents on the "many" side, don't use an array of ObjectID references. High-cardinality arrays are a compelling reason not to embed.**

Many-to-Many

- This kind of relationship can be directly implemented in MongoDB.
- Eg - We have an application where a user is assigned many tasks and a task is assigned to many users. Here, in the document of each user, we can maintain a “task” key with the value being an array of ObjectIDs of the documents of the tasks that that user is assigned and vice-versa.

Next, we will look at MongoDB Schema Design Anti-Patterns, i.e., **what not to do**

Massive Arrays
A common design pattern in MongoDB is the pattern: **data that is accessed together should be stored together**. However, if the amount of data to be stored is large, then we shouldn't use embedding to implement this pattern. Eg - If we have to access employees and building data together and thus, need to store all the employees who work at a particular building, then we should do so by storing the building name in each employee's document.

If we did so the other way round, then we would get a building document with a massive array of employees. **That is the wrong thing to do**.

Separate Documents
If our use case didn't need to store building and employee info together, then we might create separate documents for each building and each employee. However, if we ever need to display the 2 data together, then we would need to use the $lookup operator to "join" the 2 documents, which would be expensive, if a lot of these queries were needed. To solve this, we would need to use the extended reference approach where we duplicate some of the data. Eg - If our application has a user profile page that displays information about the user as well as the name of the building and the state where they work, we may want to embed the building name and state fields in the employee document.

REFERENCES
[1] MongoDB Schema Design: Data Modeling Best Practices | MongoDB
[2] Massive Arrays | MongoDB