## Should I normalize my data before storing it in MongoDB?

No. Schema design is very important when using MongoDB, but very different from schema design for relational databases. There is only some edge-cases where normalization can be necessary.

Normalizing your data like you would with a relational database is usually not a good idea in MongoDB.

Normalization in relational databases is only feasible under the premise that JOINs between tables are relatively cheap. The $lookup aggregation operator provides some limited JOIN functionality (before 2014, $lookup operator didn't exist yet), but it doesn't work with sharded collections. So, joins often need to be emulated by the application through multiple subsequent database queries, which is very slow.

For that reason, you should design your database in a way that the most common queries can be satisfied by querying a single collection, even when this means that you will have some redundancy in your database. A good tool to model 1:n relations are often arrays of sub-objects. However, try to avoid objects which grow indefinitely over time, because in that case they will be moved around the file a lot which will impact write performance.

## Write the pros and cons of Embedded data models vs Normalized data models

| | **Embedded data models** | **Normalized data models** |
|---|---|---|
| **Definition** | • All related data is encapsulated in a single document resulting in fewer queries to complete operations | • Related data is stored in separate collections and relationship is enforced by referencing one document to another |
| **Operations** | • Read operation: Retrieve data in a single database operation<br>• Write operation: Atomic as single write operation, can write or update the document | • Read operation: Multiple requests to server to resolve references<br>• Write operation: Since data is being split, multiple write operations are required that are not atomic collectively |
| **Data growth** | • Updating or inserting fields can increase the document size resulting in relocation on the disk | • Updating and inserting fields is done in separate collections |
| **Pros** | • Faster data reads<br>• Simpler queries advantageous to developer<br>• Require less compute on read operations<br>• Makes data available quickly | • Faster writes<br>• No redundant data<br>• Less database complexity<br>• Data always consistent |
| **Cons** | • Slower writes<br>• More database complexity<br>• Potential for data inconsistency<br>• Requires more storage, RAM | • Slower reads<br>• Heavy querying can overwhelm, crash hardware<br>• Table joins required since data isn’t duplicated<br>• Indexing not as efficient due to table joins |

| | | |
|---|---|---|
| **Usage** | • you have "contains" relationships between entities.<br>• you 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. | • when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.<br>• to represent more complex many-to-many relationships.<br>• to model large hierarchical data sets |