mongoDB.

Document Schema Design Cheatsheet

MongoDB's document model provides the ultimate flexibility in schema design. These best practices for structuring data promote optimal usability, performance, and efficiency for growing systems.

Tip: Always analyze your application workloads, data size, and read/write load before you dig into schema design – and only add complexity when the rewards outweigh the costs.

1 Prefer Embedding

Data that is accessed together should be stored together.

Use structure to organize data within a document

```
db.user:
{_id: "abc",
 email: "xyz@example.com",
 preferences: {alerts:[{name: "morning",
                        frequency: "daily",
                        time: {h: 6, m: 0}}, …],
               colors: {bg: "#cccccc", …}}
}
```

Sub-documents allow you to cleanly separate sections of related fields within a document, and allow you to make atomic updates without resorting to multi-document transactions.

Include (bounded) arrays of related information

```
db.business:
{_id: "def",
 name: "Bake and Go",
 addresses: [{street: "40 Elm", state: "NY"},
             {street: "101 Main St", state: "VT"}]
}
```

Arrays of subdocuments allow you to include lists of related information, like addresses or accounts.

Finding the optimal degree of embedding can take practice, and evolve with your application. Start with these rules of thumb, and don't be afraid to iterate; MongoDB is designed to embrace change.

2 Know when not to Embed

What's used apart, should be stored apart.

Move frequently accessed embedded objects to their own collections

```
db.manufacturer:
{_id: "ghi",
 name: "Swaab Automotive",
 type: "auto",
 models: [{ name: "X",
            year: 2018,
            sku: "ABCDEF-123Z"}, …]
}
```

If your subdocuments (or arrays of them) are objects you frequently use outside of their parent documents, consider moving them to their own collection.

```
db.model:
{_id: "jkl",
 name: "X",
 year: 2018,
 sku: "ABCDEF-123Z",
 manufacturer_id: "ghi"
}
```

Now we can access models independently from their parent manufacturers. This is especially useful if we are either reading from or writing to the model documents at high volume.

Don't embed lists that grow without bounds

```
db.user:
{_id: "mno",
 username: "frankysezhey",
 login_times: [{d: "1/1/2020", t: "00:14:09"},
               {…}]
}
```

Any list that gets added to continuously shouldn't be embedded; it should be its own collection.

```
db.login_audit:
{_id: "pqr",
 time: {d: "1/1/2020", t: "00:14:09"},
 user_id: "mno"
}
```

Collections are much better places for lists that can grow, and you can use indexes for fast querying by related ID.

When in doubt, it's usually good to start by embedding data in objects, especially as you're getting used to document schemas. You can always factor sub-documents out into collections later.

3 Embrace Duplication

What's used together, should be stored together: 2nd edition.

Store useful data where it's commonly accessed

```
db.user:
{_id: "stu",
 email: "hello@example.com"
}

db.post:
{_id: "vwx",
 text: "Hello World",
 user_id: "stu",
 user_email: "hello@example.com"}
```

Avoid unnecessary application joins by duplicating commonly accessed fields, at the cost of some added complexity to keep them up to date. (AKA the "extended reference" pattern)

Mind your data consistency

```
db.user:
{_id: "stu",
 email: "hello_again@example.com"
}

db.post:
{_id: "vwx",
 text: "Hello World",
 user_id: "stu",
 user_email: "hello@example.com"}
```

In cases where you don't want duplicated data to get out of sync, it's up to you to propagate any changes to the canonical object. Check out Change Streams as a great mechanism for this.

Duplication, or storing the same pieces of data in multiple documents in your database, is a powerful technique. Always know what's canonical, and keep your data consistency rules in mind.

4 Don't be Scared to Relate

There's more than one way to relate.

Use arrays of ids or backreferences for one-to-many relationships

```
db.user:
{_id: "stu",
 email: "hello@aol.com",
 friend_ids: [("aaa", "bbb"]
}

db.post:
{_id: "vwx",
 text: "Hello World",
 user_id: "stu"
}
```

Depending on your access patterns, you can retrieve related documents via sub-queries from an array of keys (e.g. db.user.find(friend_ids) or query against a reference (e.g. db.post.find(user_id)). Application-level joins, when necessary, are nearly as high performance as in-database.

Remember upkeep on bidirectional relationships

```
db.user:
{_id: "aaa",
 email: "a@example.com",
 friend_ids: ["ccc", "bbb"]
}
{_id: "bbb",
 email: "b@example.com",
 friend_ids: ["eee", "aaa"]
}
```

When changing references on either side of a bidirectional relationship, you need to remember to update the other end, too. Bidirectional relationships are another good use case for Change Streams.

Just because MongoDB isn't a traditional "relational" database doesn't mean it can't do relations. By strategically denormalizing and optimizing your usage, you can have your cake, and eat it too.

Have more questions about schema design?
Want to virtually meet some like-minded developers?

Join the MongoDB community

Join the MongoDB community developer.mongodb.com/community