**MongoDB Data Modeling**

In MongoDB, data has a flexible schema, It is totally different from SQL database where you had to determine and declare a table schema before inserting data. MongoDB collections do not enforce to declare document structure.

The main challenge in data modeling is balancing the need of your application , the performance characteristics of the database engine, and the data retrieval patterns. When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

We should consider following things while designing your schema in mongoDB.

1. Always design your schema according to user requirements.
2. Do join on write operations instead of read operations.
3. Objects which you want to use together, should be combined into one document. Otherwise they should be separated (make sure that there should not be need of joins)
4. Optimize your schema for more frequent use cases.
5. Do complex aggregation in the schema.
6. You should duplicate the data but in a limit, because disc space is cheaper than compute time.

Let us take an example of a client who needs a database design for his website. His website has the following requirements:

Every post is distinct (contains unique title, description and url).

Every post can have one or more tags.

Every post has the name of its publisher and total number of likes.

Each post can have zero or more comments and the comments must contain user name, message, data-time and likes.

For the above requirement, a minimum of three tables are required in RDBMS.

Comments :

comment\_id

post\_id

by\_user

message

date\_time

likes

Post

Id

Title

Description

url

likes

post\_by

tag\_list

id

post\_id

tag

But in MongoDB, schema design will have one collection post and has the following structure.

{

\_id: POST\_ID

title: TITLE\_OF\_POST,

description: POST\_DESCRIPTION,

by: POST\_BY,

url: URL\_OF\_POST,

tags: [TAG1, TAG2, TAG3],

likes: TOTAL\_LIKES,

comments: [

{

user: 'COMMENT\_BY',

message: TEXT,

datecreated: DATE\_TIME,

like: LIKES

},

{

user: 'COMMENT\_BY',

message: TEST,

dateCreated: DATE\_TIME,

like: LIKES

}}}

**Data Model Design**

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

**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.

For example, assume we are getting the details of employees in three different documents namely, Personal\_details, Contact and, Address, you can embed all the three documents in a single one as shown below –

{

\_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"

},

Address: {

city: "Hyderabad",

Area: "Madapur",

State: "Telangana"

}

}

**Normalized Data Model**

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"

}

**Schema Validation** :

MongoDB provides the capability to perform schema validation during updates and insertions.

Validation rules are on per collection basis.

To specify validation rules when creating a new collection, use db.createCollection() with the validator option.

validationLevel option, which determines how strictly MongoDB applies validation rules to existing documents during an update, and

validationAction option, which determines whether MongoDB should error and reject documents that violate the validation rules or warn about the violations in the log but allow invalid documents.

To specify JSON Schema validation, use the $jsonSchema operator in your validator expression.

For example, the following example specifies validation rules using JSON schema:

db.createCollection("students", {

validator: {

$jsonSchema: {

bsonType: "object",

required: [ "name", "year", "major", "address" ],

properties: {

name: {

bsonType: "string",

description: "must be a string and is required"

},

year: {

bsonType: "int",

minimum: 2017,

maximum: 3017,

description: "must be an integer in [ 2017, 3017 ] and is required"

},

major: {

enum: [ "Math", "English", "Computer Science", "History", null ],

description: "can only be one of the enum values and is required"

},

gpa: {

bsonType: [ "double" ],

description: "must be a double if the field exists"

},

address: {

bsonType: "object",

required: [ "city" ],

properties: {

street: {

bsonType: "string",

description: "must be a string if the field exists"

},

city: {

bsonType: "string",

"description": "must be a string and is required"

}

}

}

}

}

}

})

**Other Query Expressions**

In addition to JSON Schema validation that uses the $jsonSchema query operator, MongoDB supports validation with other query operators, with the exception of the $near, $nearSphere, $text, and $where operators.

For example, the following example specifies validator rules using the query expression:

db.createCollection( "contacts",

{ validator: { $or:

[

{ phone: { $type: "string" } },

{ email: { $regex: /@mongodb\.com$/ } },

{ status: { $in: [ "Unknown", "Incomplete" ] } }

]

}

} )