Data modelling using mongoDB

Topics to be Covered:

Schemas

Structuring Schemas

Data Types of MongoDB

Relations - One-One, One-Many, Many-Many

Schema Validation

Data Modelling

Unlike SQL databases, you must define and declare a table’s scheme before inserting data.But in MongoDB's collections we do not require the same schema in their documents by default.

Namely:

- Documents in the collection do not have to have the same type of fields and the data type for the field may differ in the documents in the collection.
- Update documents to a new structure to change the structure of documents in the collection, such as adding new fields, removing existing fields, or changing field values to a new type.

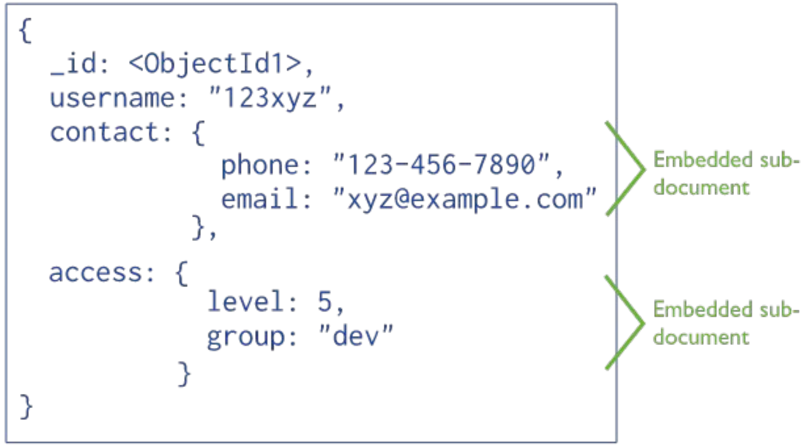
Although the document is significantly different from the other documents in the collection, each document can be matched with the data field of the representative entity.

Document Structure

Key decisions in creating data models for MongoDB applications revolve around the structure of documents and how applications' relationships relate to data. Allows MongoDB to embed relevant data in the document.

Embedded Data

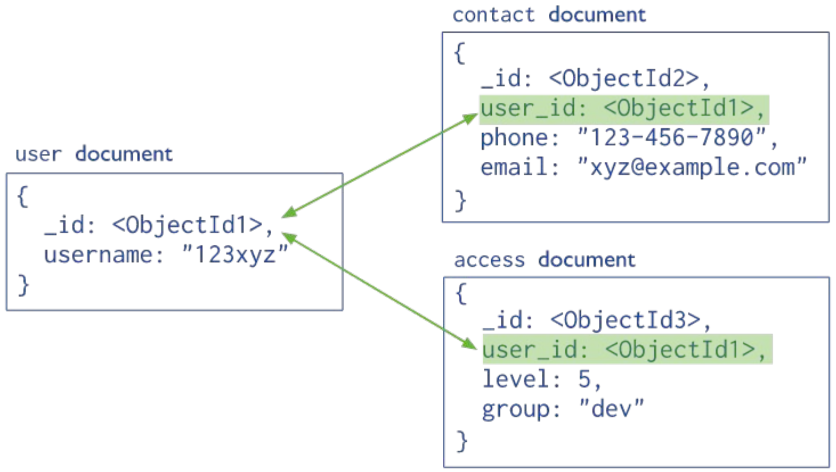
Embedded documents capture the relationships between data by storing relevant data in a document structure. MongoDB documents is useful to make embed document structures in a field or in a series of documents. These differentiated data models allow applications to retrieve and convert relevant data in a single database operation.



```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

References

References store connections between data by adding links or references from one document to another. Applications can resolve these situations to access relevant data. Broadly speaking, these are generalised data models.



```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

Atomization of written Operations

Single document Atomicity

In MongoDB, although the operation modifies many embedded documents in a single document, the write operation is atomic at the same document level.

The denormalized data model with embedded data combines the amount of relevant data into a single document rather than generalising multiple documents and collections. This data model facilitates nuclear operations.

Example:

```
$session.startTransaction()
   db.users.updateOne({_id:3 , {$set:{age:50}}})
session.commit_transaction()
```

Atomization of written Operations

Multi-document transaction

When editing multiple documents in a single written operation (such as db.Collection.updateMany ()), the transformation of each document is atomic, but not the whole operation atomic.
When performing multi-document writing operations, other operations may be interleaved, either by a single writing operation or by multiple writing operations.
For situations where the atomic power of reading and writing multiple documents (in one or multiple collections) is required, MongoDB supports multi-document transactions:

```
$session.startTransaction()
   db.users.updateMany({{_id:3 , {$set:{age:50}}},{_id:4 ,
{$set:{age:54}}}})
session.commit_transaction()
```

Schema

To define what a JSON schema is, we must first define what a JSON is.

JSON stands for "JavaScript Object Notation". JavaScript is present in most web browsers and since JSON is Javascript based, it is very easy to support there. However, it has proven to be very useful and simple, it is now used in many other contexts where I do not surf the web.

```
{
"type": "object",
   "properties": {
       "first_name": { "type": "string" },
       "last_name": { "type": "string" },
       "birthday": { "type": "string",
"format": "date" },
       "address": {
       "type": "object",
       "properties": {
           "street_address": { "type":
"string" },
           "city": { "type": "string" },
           "state": { "type": "string" },
           "country": { "type" : "string" }
       }
       }
   }
}
```

Structuring Schema

Now, the MongoDB schema design works much differently than the relational schema design. With MongoDB schema design, it is:

- There is no formal process
- No algorithm
- No terms

While designing your MongoDB schema design, it is only important to design a scheme that works good for application. There can be two different apps with the exact data have different schemes if used differently. When formulating the scheme, we would like to keep in mind the following:

- Store data
- Provide good query performance
- Adequate amount of hardware is required

```
{
 "first_name": "Paul",
 "surname": "Miller",
 "cell": "447557505611",
 "city": "London",
 "location": [45.123, 47.232],
 "profession": ["banking", "finance",
"trader"],
 "cars": [
      {
          "model": "Bentley",
          "year": 1973
      },
      {
          "model": "Rolls Royce",
          "year": 1965
      }
 ]
}
```

Embedding vs Referencing

Embedding

Pros

1. You can get all the relevant information in a single query.
2. Avoid entering the application code or using $lookups.
3. Update relevant information into a single molecular operation. By default, all CRUD operations in a single document are ACID compliant.
4. However, if you need transactions in multiple operations, you can use the transaction operator.
5. I have to add that even though transactions from 4.0 are available, relying too much on what you use in your application is the opposite pattern.

Cons

1. If most fields are not relevant, large documents mean more overhead. Limiting the number of documents that can be sent over the wire per query, you can increase query performance as well.
2. MongoDB has a 16-MB document size limit. If you are embedding a lot of data in a single document, you can reach this limit.

Embedding vs Referencing

Referencing

Pros

1. By splitting the data, you have smaller documents.
2. The chances of each document reaching the 16-MB-limit are slim.
3. Not every question requires frequently accessed information.
4. Reduce the amount of data duplication. Also, note that data duplication should not be avoided if it leads to a better scheme.

Cons

1. Recovering all the data in the prescribed documents requires at least two questions or $ lookups to retrieve all the information.

Data Types in MongoDB

Text

- String

Numeric

- 32-Bit Integer
- 64-Bit Integer
- Double
- Decimal128

Date/Time

- Date
- Timestamp

Other

- Object
- Array
- Binary Data
- ObjectId
- Boolean
- Null
- Regular Expression
- JavaScript
- Min Key
- Max Key

Types of Relationship

One-to-One

Take a look at our user documentation. This example contains some great one-to-one data. For example, in our system, one user can only have one name. So, this is an example of a relationship with each other. We can model all the data in our database into key-value pairs.

```
{
 "_id": "ObjectId('AAA')",
 "name": "Joe Karlsson",
 "company": "MongoDB",
 "twitter": "@JoeKarlsson1",
 "twitch": "joe_karlsson",
 "tik tok": "joe karlsson",
 "website": "joekarlsson.com"
}
```

Types of Relationship

- **One-to-Many**

Well, suppose you are creating a product page for an e-commerce website and you need to create a scheme where you can show product information. In our system, we save information about the many components that make up each product for repair services. How do you design a scheme to save all this data, but still make your product page work? You may want to consider one to several schemes as your one product is designed with many components. Now, with a scheme that can effectively save thousands of sub-components, we probably do not need to keep all the data for the components in each request, but it is still important to maintain this relationship in our schema. So, we may have a product collection with data related to each product in our e-commerce store and to keep the data linked to that part, we can put a series of object IDs that link to that document. It contains information about the component. If necessary these parts can be saved in a single collection or in a separate archive. Let's see how it goes.

Products:

```
{
 "name": "left-handed smoke shifter",
 "manufacturer": "Acme Corp",
 "catalog_number": "1234",
 "parts": ["ObjectID('AAAA')", "ObjectID('BBBB')",
"ObjectID('CCCC')"]
}
```

Parts:

```
{
 "_id" : "ObjectID('AAAA')",
 "part no" : "123-aff-456",
 "name" : "#4 grommet",
 "qty": "94",
 "cost": "0.94",
 "price":" 3.99"
}
```

Types of Relationship

- **Many-to-Many**

The final schema design pattern that we are going to cover in this post is from many to many relationships. This is another common schema pattern that we see from time to time in relational and Mongodb schema designs. For this model, suppose we are creating a to-do application. In our app, a user can have multiple tasks and assign multiple users to a task.

To maintain these relationships between users and tasks, there must be instructions from one user to multiple tasks and instructions from task to multiple users. Let's see how it works for the to-do list application.

User:

```
{
 "_id": ObjectID("AAF1"),
 "name": "Kate Monster",
 "tasks": [ObjectID("ADF9"), ObjectID("AE02"),
ObjectID("AE73")]
}
```

Tasks:

```
{
 "_id": ObjectID("ADF9"),
 "description": "Write blog post about MongoDB
schema design",
 "due_date": ISODate("2014-04-01"),
 "owners": [ObjectID("AAF1"), ObjectID("BB3G")]
}
```

Schema Validation

Specify the validation rules
Verification rules are based on each collection.

db.createCollection() is used with the validator option to mention the validation rules while creating a new collection.

For adding document verification to an existing collection, use the CollMod command with the validator option.

MongoDB also offers the following related options:

- validationLevel selection, which determines how accurately Mongodb applies validation rules to existing documents at the time of update.
- validationAction option, which determines whether MongoDB incorrectly rejects documents that violate the authentication rules or warn of violations in the log but allow invalid documents.

Schema Validation

JSON Schema

MongoDB supports JSON schema validation. The $jsonSchema operatoris being used in your validator statement To mention JSON schema validation.

For example, the following example specifies a validation rule using the JSON scheme:

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         properties: {
            name: {
               description: "string and is mandatory",
               bsonType: "string"
            },
            year: {
               description: "integer between [ 2017, 3017 ] and
mandatory",
               maximum: 3017,
               bsonType: "int",
               minimum: 2017,
            },
            major: {
               description: "can only be one of the enum values
and mandatory",
               enum: [ "Maths", "English", "Computer Science",
"History", null ]
            },
            gpa: {
               bsonType: [ "double" ],
               description: "double if the field exists"
            },
```

```
          address: {
                bsonType: "object",
                required: [ "city" ],
                properties: {
                   street: {
                      bsonType: "string",
                      description: "string if the
field exists"
                   },
                   city: {
                      bsonType: "string",
                      description: "string and
mandatory"
                   }
                }
             }
          },
          required: [ "name", "year", "major",
"address" ],
          bsonType: "object"
       }
    }
})
```

Schema Validation

Other Query expressions

In addition to the JSON schema validation that uses the $ jsonSchema query operator, MongoDB supports authentication with other query operators, excluding:

- $near
- $nearSphere
- $text
- $where
- $expr with $function expressions.

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

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   }
} )
```

Schema Validation

Accept or reject an invalid document

The option of validationActions defines how MongoDB handles documents that violate the validation rules:

- If the validationAction is an error (default), MongoDB will reject any insert or update that violates the validation criteria.
- If the verification process is alerted, MongoDB will log any violations, but allow the insertion or update to continue.

For example, create the Contacts2 collection with the following JSON schema validator:

```
db.createCollection( "contacts2", {
   validator: { $jsonSchema: {
      bsonType: "object",
      required: [ "phone" ],
      properties: {
         phone: {
            bsonType: "string",
            description: "string and mandatory"
         },
         email: {
            bsonType : "string",
            pattern : "@mongodb\.com$",
            description: "must be a string and match the regular expression pattern"
         },
         status: {
            enum: [ "Unknown", "Incomplete" ],
            description: "enum values only"
         }
      }
   } },
   validationAction: "warn"
} )
```

MCQ

1. Embedded data model is used for.

A. Isa relationship.

B. contains relationship

C. Inheritance relationship

D. All of the above.

2. Normalised data models between document is defined by

A. References

B. Relativeness

C. Evaluation

D. All of the above

3. Relationship between connected data.

A. One-to-One

B. One to many

C. Many to many

D. None of the above.

MCQ

4. Which of these are property of BSON.

A. The field names cannot start with the dollar sign ($) character

B. The field names cannot contain the dot (.) character

C. The field names cannot contain the null character

D. All of the above

5. Primary key field?

A. _uuid

B. _uid

C. _id

D. None of the above

Practice Problems

1. Avoid mutable, growing arrays in the following schema by updating/deleting some lines in the json?

```
{
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA",
   books: [12346789, 234567890, ...]
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}
```

Practice Problems

```
{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English"
}
```

2. Optimize schema to embed the address data entities in the patron data for following schema representing one to many relationships?

```
{
   _id: "joe",
   name: "Joe Bookreader"
}
```

Practice Problems

```
{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: "12345"
}

{
   patron_id: "joe",
   street: "1 Some Other Street",
   city: "Boston",
   state: "MA",
   zip: "12345"
}
```

Thank You!