Fundamental Mapping from Relational Database to MongoDB

Tuan Dung Lai
Faculty of Science, Engineering and Technology
Swinburne University of Technology
Hawthorn, Victoria 3122
Email: tuandunglai@gmail.com

***Abstract*—NoSQL database are becoming a fundamental role of the database landscape nowadays, first version released in 2007, at that time, MySQL, an open-source relational database management system (RDBMS) has been developed extensively for more than 10 years. NoSQL has a handful of advantages including lower cost, easier scalability and open sources features. However, NoSQL is still a relatively young technology, making it difficult for IT engineer and student to learn.**

I. Introduction

With the less constrained structure, scalable schema design of NoSQL and faster access compared to traditional RDBMS. Coming from an SQL background, it could be challenging and time-consuming to map database management concept from SQL to NoSQL. This report will go through fundamental concept of MongoDB functionality, terms and simple query syntax. The difference of this report is that we assume that you have basic knowledge of SQL, and everything from this report will be built and map based on that knowledge you have on SQL.

II. Mapping Tables, Rows, Columns

A. Quick SQL review

In SQL databse, tables are objects that are being managed, all data or information of objects and database are stored in tables. Table contains column (a set of data values of a particular type) and row (attribute type and their value). The columns provide the structure according to which the rows are composed.

B. Collection

Each database in MongoDB consists of collections which are equivalent to an RDBMS database consisting of SQL tables. Collection stores data in the form of documents which is equivalent to tables storing data in rows.

SQL	NoSQL
Table	Collection
Column	JSON structure
Row	Field

```
{
"_id": ObjectId("5146bb52d8524270060001f3"),
"age": 25,
"city": "Los Angeles",
"email": "mark@abc.com",
"user_name": "Mark Hanks"
}
```

The document above is equivalent to a row in RDBMS, a collection contains multiple document above. This document use JSON format, it consists of key value pairs.
Notice that the field above has a unique id. It is considered as primary key when comparing to MySQL.

Fig. 1: Mapping table to collection [4]

C. Dynamic Schema

In NoSQL, documents within a collection can have different schema. The fields in MongoDB can be easily added, removed and modified. For example, typical documents have 4 fields, like students have name, class, gender and date of birth, now, it is possible to create a student that have name, faculty and parent details. The number of fields as well as structure can be different. There is no constraint on data types of the fields. This functionality to use dynamic schema allows us to generate dynamic documents at run time.

For instance, consider the following two documents inside the same collection but having different schemas (Fig 2):

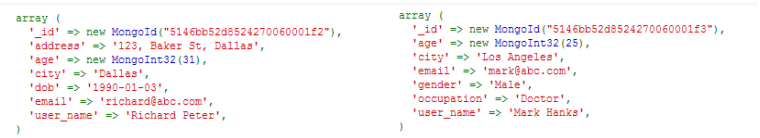

Fig. 2: Dynamic schema example [4]

The first document contains the fields address and dob which are not present in the second document while the second

document contains fields gender and occupation which are not present in the first one. Imagine if we would have designed this thing in SQL, we would have kept four extra columns for address, dob, gender and occupation, some of which would store empty (or null) values, and hence occupying unnecessary space.

This model of dynamic schema is the reason why NoSQL databases are highly scalable in terms of design. Various complex schemas (hierarchical, tree-structured, etc) which would require number of RDBMS tables can be designed efficiently using such documents. A typical example would be to store user posts, their likes, comments and other associated information in the form of documents. An SQL implementation for the same would ideally have separate tables for storing posts, comments and likes while a MongoDB document can store all these information in a single document.

III. Mapping Join and Relationships

A. Quick SQL review

Relationships in RDBMS are achieved using primary and foreign key relationships and querying those using joins. There is no such straightforward mapping in MongoDB but the relationships here are designed using embedded and linking documents.

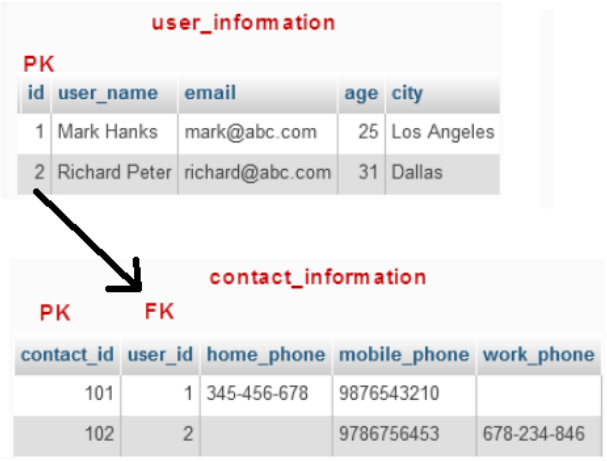

Fig. 3: Relation in SQL

B. Linking Documents and Embedding Documents

1) Linking Documents: Linking documents is a method in which multiple collections are created with some similar fields both collection, these similar fields are equivalent to foreign key in SQL.

Fig. 4: Linking Document in NoSQL

The user id field in our document in Fig 4 is simply a field that holds some data and all the logic associated with it has to be implemented by us. For example, even if you will insert some user id in the contact information document that does not exist in the user information collection, MongoDB is not going to throw any error saying that corresponding user id was not found in the user information collection(unlike SQL where this would be an invalid foreign key constraint).

2) Embedding Documents: Embedding documents mean instead of creating separate collections as we did in Linking Document, we will try to merge every information into one collection. The fields that are being copied from one collection to another collection will be extended in only one collection.

Fig. 5: Embedded Documents in NoSQL

In the above example, we have embedded a small document of contact information inside the user information. In the similar manner, large complex documents and hierarchical data can be embedded like this to relate entities.

IV. Mapping Query

In this section, we will translate all fundamental query from MySQL to MongoDB along with example in syntax. Let's start with a brief on terminology and concepts:

SQL	NoSQL
database	database
table	collection
row	document or JSON document
column	field
index	index
table joins	lookup, embedded documents
primary key	primary key
aggregation	aggregation pipeline

A. Create

In MongoDB, it is not possible and not necessary to define the structure of a collection as we normally did in MySQL (create table, naming column, define data type). MongoDB use JSON format which is an unstructured format to store data.
The structure of the document is automatically created when the first insert occurs in the collection. However, you can create an empty collection using createCollection command.

```
CREATE TABLE `posts` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `post_text` varchar(500) NOT NULL,
    `user_name` varchar(20) NOT NULL,
    `post_privacy` varchar(10) NOT NULL,
    `post_likes_count` int(11) NOT NULL,
    PRIMARY KEY (`id`)
)
```

Fig. 6: Create table in SQL

In MongoDB, we can only create an empty collection:

db.createCollection("posts").

B. Insert

```
INSERT INTO people(user_id,
                   age,
                   status)
VALUES ("bcd001",
        45,
        "A")
```

Fig. 7: Insert data in SQL

Data in NoSQL has JSON format, we can either insert the JSON file to collection or insert key-value pair into collection.

db.people.insert(userid: "bcd001", age: 45, status: "A").

Note: We can create a new field for id. However, The inserted document will contain the auto generated id field.

C. Read

1) Read All Data: MongoDB uses the find method which is equivalent to the SELECT command in SQL. The following statements simply read all the documents from the posts collection.

```
SQL: SELECT * FROM `posts`

MongoDB: db.posts.find()
```

Fig. 8: Read all data

2) Read specific data in collection: The following query fetches specific columns, post text and post likes count as specified in the second set of braces . We mark the value as 1 in the second bracket to include the field in the output.

```
SQL: SELECT `post_text` , `post_likes_count` FROM `posts`

MongoDB: db.posts.find({},{post_text:1,post_likes_count:1})
```

Fig. 9: Read specific data

3) Read data with condition: The following query does a conditional search for documents having username field as mark. All the criteria for fetching the documents have to be placed in the first braces separated by commas.

```
SQL: SELECT * FROM `posts` WHERE `user_name` = 'mark'

MongoDB: db.posts.find({user_name:"mark"})
```

Fig. 10: Read data with condition

4) Read auto generate ID: In MongoDB, every time we input new data, one specific id will be auto generated and this id will be displayed by all read commands that we described above. Set id to 0 to disable showing id with find command.

db.posts.find(,posttext:1,postlikescount:1,id:0).

```
SQL: SELECT `post_text` , `post_likes_count` FROM `posts` WHERE `user_name` = 'mark'

MongoDB: db.posts.find({user_name:"mark"},{post_text:1,post_likes_count:1})
```

Fig. 11: Read specific data with condition

5) Read specific data with condition:

6) Logical Operator: We will examine how to do an **OR** operator in MongoDB, same syntax applied to other logical operators.

```
SELECT post_text , post_likes_count
FROM posts
WHERE user_name = 'mark'
    OR post_privacy = 'public'
```

Fig. 12: Logical operator OR in MySQL

In MongoDB, we use the following command:

db.posts.find($or:[username:'mark',postprivacy:'public'],posttext:1,postlikescount:1).

7) Sort: Next, we will use the sort method which sorts the result in ascending order of postlikescount(indicated by 1).

```
SELECT *  FROM `posts`
WHERE `user_name` = 'mark'
order by post_likes_count ASC
```

Fig. 13: Sort in MySQL

db.posts.find(username:"mark").sort(postlikescount:1)).

To sort the results in descending order, we specify -1 as the value of the field.

db.posts.find(username:"mark").sort(postlikescount:-1)).

8) Limit display output data: To limit the number of documents to be returned, we use the limit method specifying the number of documents.

```
SQL: SELECT *  FROM `posts` LIMIT 10

MongoDB: db.posts.find().limit(10)
```

Fig. 14: Limit number of output data being displayed

D. Update

The first parameter to the update method specifies the criteria to select the documents. The second parameter specifies the actual update operation to be performed. For example, the following query selects all the documents with username as mark and sets their postprivacy as private.
One difference here is that by default, MongoDB update query updates only one (and the first matched) document. To update all the matching documents we have to provide a third parameter specifying multi as true indicating that we want to update multiple documents.

db.posts.update(username:"mark",$set:postprivacy:"private",multi:true).

```
UPDATE posts
SET post_privacy = "private"
WHERE user_name='mark'
```

Fig. 15: Update in SQL

E. Remove

```
SQL: DELETE FROM posts WHERE user_name='mark'

MongoDB:  db.posts.remove({user_name:"mark"})
```

Fig. 16: Remove comparison

V. Conclusion

This report has shown you fundamental concept of MongoDB, most of the core concepts in term of data management are similar to MySQL. Upon this report, reader should try out more complex queries such as map reduce in order to gain more knowledge of NoSQL.
Attached to this report is an example where the author of this report convert an University database in SQL to NoSQL. Reader can check out that report for further details.

Acknowledgment

The author would like to thank Dr. Irene Moser, my lecture for the unit called Fundamental of Data Management, unit code COS20015. She has provided a great series of useful lecture which help me a lot in term of understanding data management theory. Also, I would like to thank my tutor Rasoul Rahmani for great support during the semester.

References

[1] https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints
[2] https://www.javatpoint.com/mongodb-history
[3] https://code.tutsplus.com/articles/mapping-relational-databases-and-sql-to-mongodb–net-35650
[4] https://docs.mongodb.com/manual/reference/sql-comparison/
[5] https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-foreign-key-relationships