tuts+

CODE > DATABASES

Mapping Relational Databases and SQL to MongoDB

by Ashish Trivedi 6 Feb 2014

Length: Long Languages: English

Databases Web Development

NoSQL databases have emerged tremendously in the last few years owing to their less constrained structure, scalable schema design, and faster access compared to traditional relational databases (RDBMS/SQL). MongoDB is an open source document-oriented NoSQL database which stores data in the form of JSON-like objects. It has emerged as one of the leading databases due to its dynamic schema, high scalability, optimal query performance, faster indexing and an active user community.

If you are coming from an RDBMS/SQL background, understanding NoSQL and MongoDB concepts can be bit difficult while starting because both the technologies have very different manner of data representation. This article will drive you to understand how the RDBMS/SQL domain, its functionalities, terms and query language map to MongoDB database. By mapping, I mean that if we have a concept in RDBMS/SQL, we will see what its equivalent concept in MongoDB is.

We will start with mapping the basic relational concepts like table, row, column, etc and move to discuss indexing and joins. We will then look over the SQL queries and discuss their corresponding MongoDB database queries. The article assumes that you are aware of the basic relational database concepts and SQL, because throughout the article more stress will be laid on understanding how these concepts map in MongoDB. Let's begin.

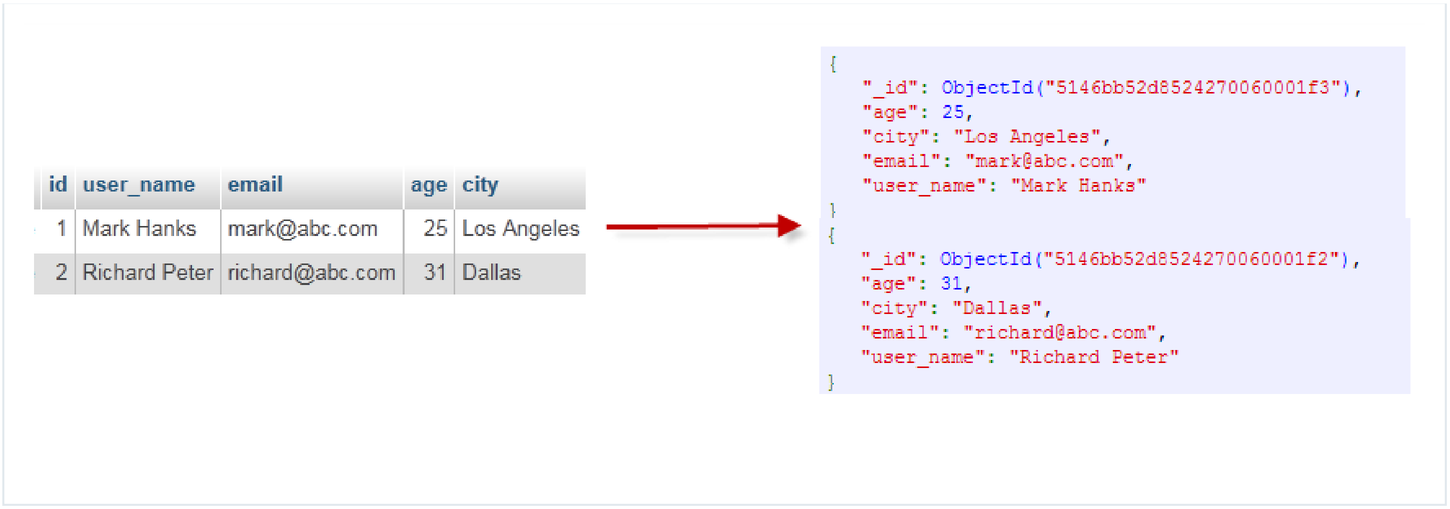
Mapping Tables, Rows and Columns

Each database in MongoDB consists of collections which are equivalent to an RDBMS database consisting of SQL tables. Each collection stores data in the form of documents which is equivalent to tables storing data in rows. While a row stores data in its set of columns, a document has a JSON-like structure (known as BSON in MongoDB). Lastly, the way we have rows in an SQL row, we have fields in MongoDB. Following is an example of a document (read row) having some fields (read columns) storing user data:

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

This document is equivalent to a single row in RDBMS. A collection consists of many such documents just as a table consists of many rows. Note that each document in a collection has a unique `_id` field, which is a 12-byte field that serves as a primary key for the documents. The field is auto generated on creation of the document and is used for uniquely identifying each document.

To understand the mappings better, let us take an example of an SQL table `users` and its corresponding structure in MongoDB. As shown in Fig 1, each row in the SQL table transforms to a document and each column to a field in MongoDB.



id	user_name	email	age	city
1	Mark Hanks	mark@abc.com	25	Los Angeles
2	Richard Peter	richard@abc.com	31	Dallas

```
{
   "_id": ObjectId("5146bb52d8524270060001f3"),
   "age": 25,
   "city": "Los Angeles",
   "email": "mark@abc.com",
   "user_name": "Mark Hanks"
}
{
   "_id": ObjectId("5146bb52d8524270060001f2"),
   "age": 31,
   "city": "Dallas",
   "email": "richard@abc.com",
   "user_name": "Richard Peter"
}
```

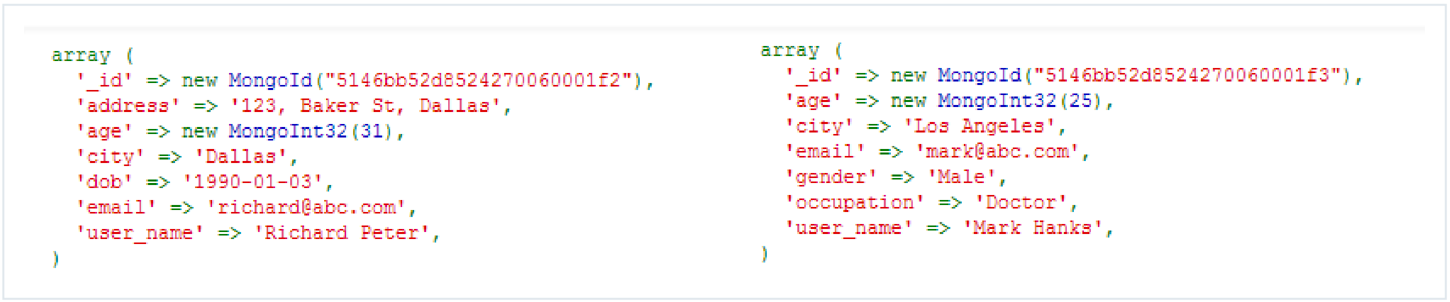
Figure 1

Dynamic Schema

One interesting thing to focus here is that different documents within a collection can have different schemas. So, it is possible in MongoDB for one document to have five fields and the other document to have seven fields. The fields can be easily added, removed and modified anytime. Also, there is no constraint on data types of the fields. Thus, at one instance a field can hold `int` type data and at the next instance it may hold an `array`.

These concepts must seem very different to the readers coming from RDBMS background where the table structures, their columns, data types and relations are pre-defined. 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):



```
array (
  '_id' => new MongoId("5146bb52d8524270060001f2"),
  'address' => '123, Baker St, Dallas',
  'age' => new MongoInt32(31),
  'city' => 'Dallas',
  'dob' => '1990-01-03',
  'email' => 'richard@abc.com',
  'user_name' => 'Richard Peter',
)
```

```
array (
  '_id' => new MongoId("5146bb52d8524270060001f3"),
  'age' => new MongoInt32(25),
  'city' => 'Los Angeles',
  'email' => 'mark@abc.com',
  'gender' => 'Male',
  'occupation' => 'Doctor',
  'user_name' => 'Mark Hanks',
)
```

Figure 2

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.

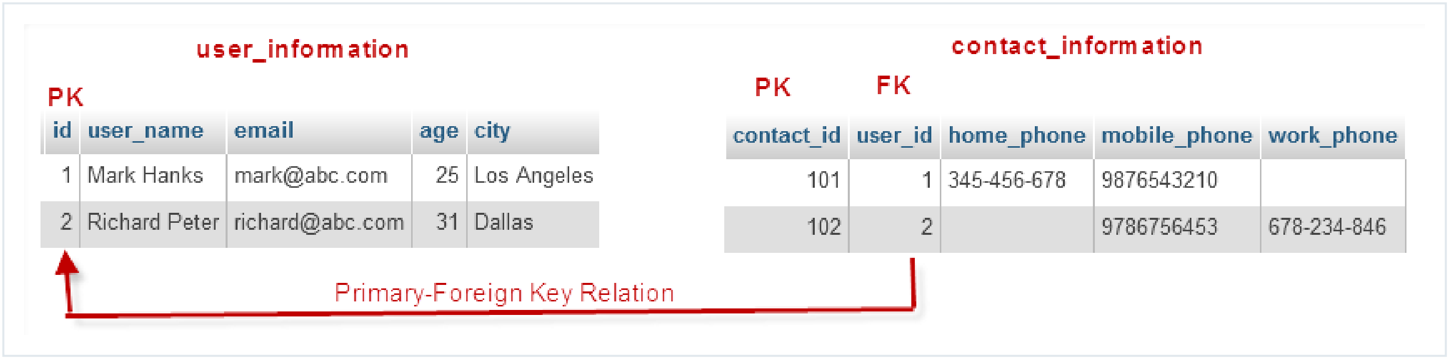
This model of dynamic schema is the reason why NosSQL databases are highly scalable in terms of design. Various complex schemas (hierarchical, tree-structured, etc) which would require number of RDBMS tables can be designed efficiently using such documents. A typical example would be to store user posts, 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.

Advertisement

Mapping Joins and Relationships

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.

Consider an example wherein we need to store user information and corresponding contact information. An ideal SQL design would have two tables, say `user_information` and `contact_information`, with primary keys `id` and `contact_id` as shown in Fig 3. The `contact_information` table would also contain a column `user_id` which would be the foreign key linking to the `id` field of the `user_information` table.


user_information
PK
contact_information
PK
FK
Primary-Foreign Key Relation

id	user_name	email	age	city
1	Mark Hanks	mark@abc.com	25	Los Angeles
2	Richard Peter	richard@abc.com	31	Dallas

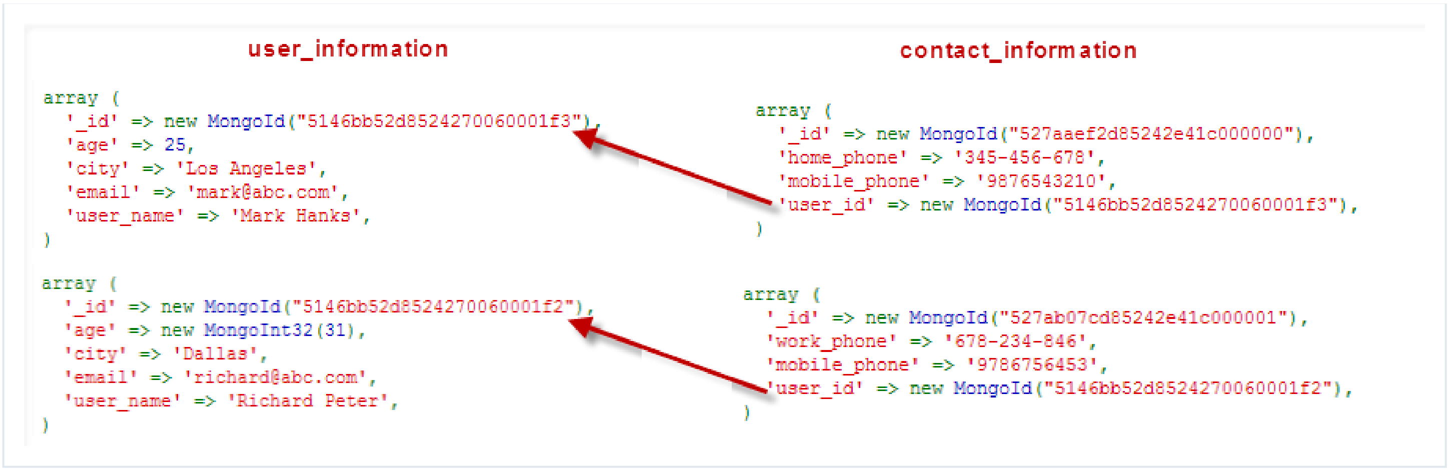
contact_id	user_id	home_phone	mobile_phone	work_phone
101	1	345-456-678	9876543210	
102	2		9786756453	678-234-846

Figure 3

Now we will see how we would design such relationships in MongoDB using approaches of Linking documents and Embedded documents. Observe that in the SQL schema, we generally add a column (like `id` and `contact_id` in our case) which acts as a primary column for that table. However, in MongoDB, we generally use the auto generated `_id` field as the primary key to uniquely identify the documents.

Linking Documents

This approach will use two collections, `user_information` and `contact_information` both having their unique `_id` fields. We will have a field `user_id` in the `contact_information` document which relates to the `_id` field of the `user_information` document showing which user the contact corresponds to. (See Fig 4) Note that in MongoDB, the relations and their corresponding operations have to be taken care manually (for example, through code) as no foreign key constraints and rules apply.



```
array (
  '_id' => new MongoId("5146bb52d8524270060001f3"),
  'age' => 25,
  'city' => 'Los Angeles',
  'email' => 'mark@abc.com',
  'user_name' => 'Mark Hanks',
)

array (
  '_id' => new MongoId("5146bb52d8524270060001f2"),
  'age' => new MongoInt32(31),
  'city' => 'Dallas',
  'email' => 'richard@abc.com',
  'user_name' => 'Richard Peter',
)
```

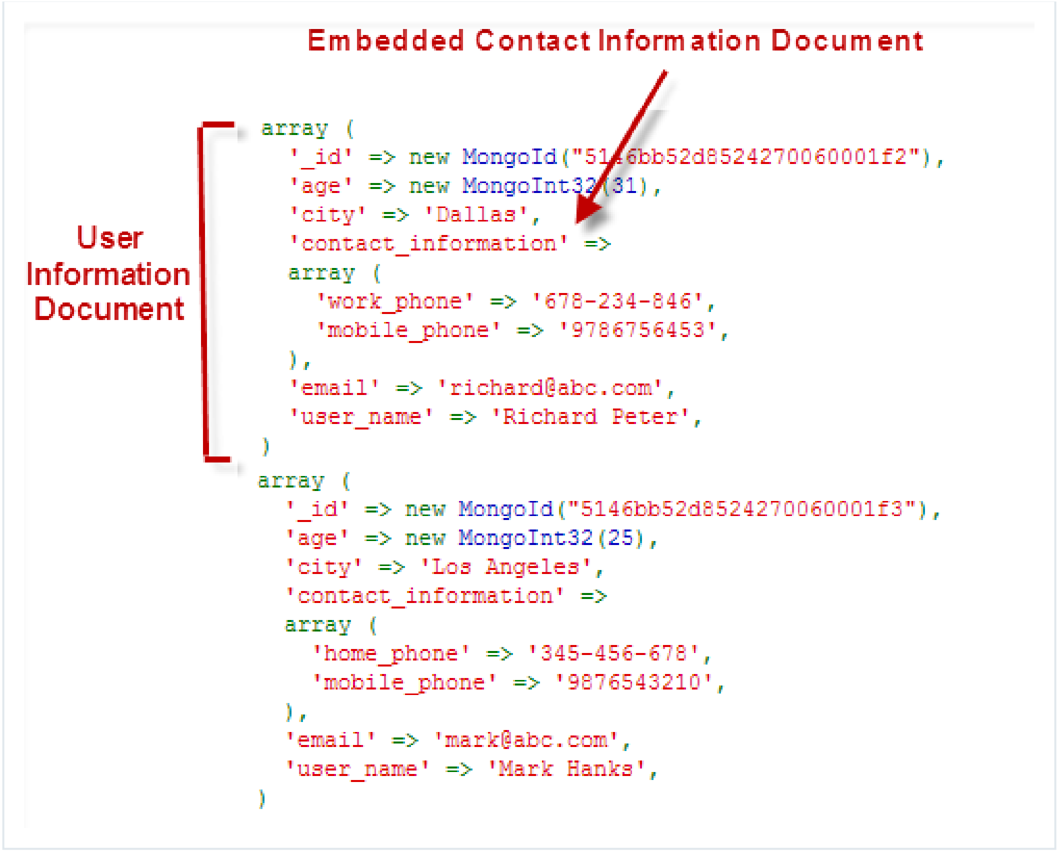
```
array (
  '_id' => new MongoId("527aaef2d85242e41c000000"),
  'home_phone' => '345-456-678',
  'mobile_phone' => '9876543210',
  'user_id' => new MongoId("5146bb52d8524270060001f3"),
)

array (
  '_id' => new MongoId("527ab07cd85242e41c000001"),
  'work_phone' => '678-234-846',
  'mobile_phone' => '9786756453',
  'user_id' => new MongoId("5146bb52d8524270060001f2"),
)
```

Figure 4

The `user_id` field in our document 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).

Embedding Documents

The second approach is to embed the `contact_information` document inside the `user_information` document like this (Fig 5):


Embedded Contact Information Document
User Information Document

```
array (
  '_id' => new MongoId("5146bb52d8524270060001f2"),
  'age' => new MongoInt32(31),
  'city' => 'Dallas',
  'contact_information' =>
  array (
    'work_phone' => '678-234-846',
    'mobile_phone' => '9786756453',
  ),
  'email' => 'richard@abc.com',
  'user_name' => 'Richard Peter',
)
array (
  '_id' => new MongoId("5146bb52d8524270060001f3"),
  'age' => new MongoInt32(25),
  'city' => 'Los Angeles',
  'contact_information' =>
  array (
    'home_phone' => '345-456-678',
    'mobile_phone' => '9876543210',
  ),
  'email' => 'mark@abc.com',
  'user_name' => 'Mark Hanks',
)
```

Figure 5

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.

Also, which approach to use among Linking and Embedded approach depends on the specific scenario. If the data to be embedded is expected to grow larger in size, it is better to use Linking approach rather than Embedded approach to avoid the document becoming too large. Embedded approach is generally used in cases where a limited amount of information (like address in our example) has to be embedded.

Mapping Chart

To summarize, the following chart (Fig 6) represents the common co-relations we have discussed:

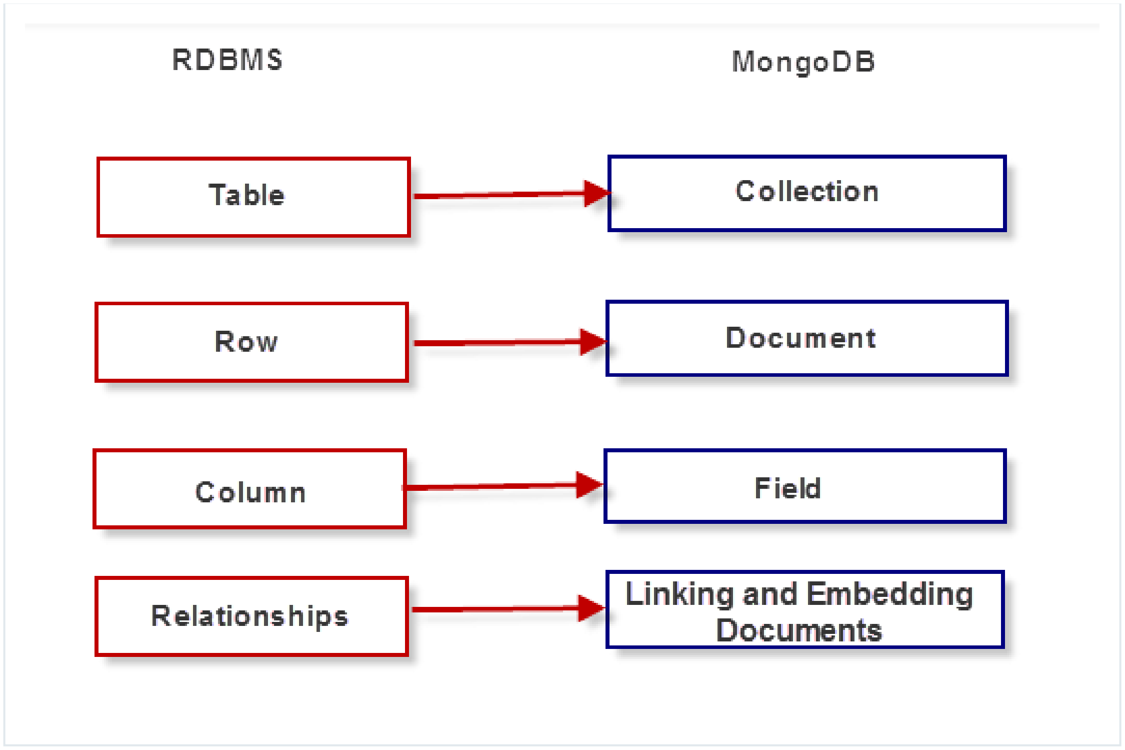


Figure 6

Mapping SQL to MongoDB Queries

Now that we are comfortable with the basic mappings between RDBMS and MongoDB, we will discuss how the query language used to interact with the database differs between them.

For MongoDB queries, let us assume a collection `users` with document structure as follows:

```
{
"_id": ObjectId("5146bb52d8524270060001f3"),
"post_text":"This is a sample post" ,
"user_name": "mark",
"post_privacy": "public",
"post_likes_count": 0
}
```

For SQL queries, we assume the table `users` having five columns with the following structure:

Figure 7

We will discuss queries related to create and alter collections (or tables), inserting, reading, updating and removing documents (or rows). There are two queries for each point, one for SQL and another for MongoDB. I will be explaining the MongoDB queries only as we are quite familiar with the SQL queries. The MongoDB queries presented here are written in the Mongo JavaScript shell while the SQL queries are written in MySQL.

Create

In MongoDB, there is no need to explicitly create the collection structure (as we do for tables using a `CREATE TABLE` query). 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.

```
SQL: CREATE TABLE `posts` (`id` int(11) NOT NULL AUTO_INCREMENT,`post_text` varchar(500)

MongoDB: db.createCollection("posts")
```

Insert

To insert a document in MongoDB, we use the `insert` method which takes an object with key value pairs as its input. The inserted document will contain the autogenerated `_id` field. However, you can also explicitly provide a 12 byte value as `_id` along with the other fields.

```
SQL: INSERT INTO `posts` (`id` ,`post_text` ,`user_name` ,`post_privacy` ,`post_likes_co

MongoDB:  db.posts.insert({user_name:"mark", post_text:"This is a sample post", post_pri
```

There is no `Alter Table` function in MongoDB to change the document structure. As the documents are dynamic in schema, the schema changes as and when any update happens on the document.

Read

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()
```

The following query does a conditional search for documents having `user_name` 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"})
```

The following query fetches specific columns, `post_text` and `post_likes_count` as specified in the second set of braces {}.

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

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

Note that MongoDB by default returns the `_id` field with each find statement. If we do not want this field in our result set, we have to specify the `_id` key with a `0` value in the list of columns to be retrieved. The `0` value of the key indicates that we want to exclude this field from the result set.

```
MongoDB: db.posts.find({},{post_text:1,post_likes_count:1,_id:0})
```

The following query fetches specific fields based on the criteria that `user_name` is `mark`.

```
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})
```

We will now add one more criteria to fetch the posts with privacy type as public. The criteria fields specified using commas represent the logical `AND` condition. Thus, this statement will look for documents having both `user_name` as `mark` and `post_privacy` as `public`.

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

MongoDB: db.posts.find({user_name:"mark",post_privacy:"public"},{post_text:1,post_likes_
```

To use logical `OR` between the criteria in the `find` method, we use the `$or` operator.

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

MongoDB: db.posts.find({$or:[{user_name:"mark"},{post_privacy:"public"}]},{post_text:1,p
```

Next, we will use the `sort` method which sorts the result in ascending order of `post_likes_count`(indicated by *1*).

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

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

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

```
SQL: SELECT *  FROM `posts` WHERE `user_name` = 'mark' order by post_likes_count DESC

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

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)
```

The way we use `offset` in SQL to skip some number of records, we use `skip` function in MongoDB. For example, the following statement would fetch ten posts skipping the first five.

```
SQL: SELECT *  FROM `posts` LIMIT 10 OFFSET  5

MongoDB: db.posts.find().limit(10).skip(5)
```

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 `user_name` as `mark` and sets their `post_privacy` 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.

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

MongoDB: db.posts.update({user_name:"mark"},{$set:{post_privacy:"private"}},{multi:true}
```

Remove

Removing documents is quite simple and similar to SQL.

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

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

Indexing

MongoDB has a default index created on the `_id` field of each collection. To create new indexes on the fields, we use `ensureIndex` method specifying the fields and associated sort order indicated by `1` or `-1`(ascending or descending).

```
SQL: CREATE INDEX index_posts ON posts(user_name,post_likes_count DESC)

MongoDB: db.posts.ensureIndex({user_name:1,post_likes_count:-1})
```

To see all the indexes present in any collection, we use `getIndexes` method on the same lines of `SHOW INDEX` query of SQL.

```
SQL: SHOW INDEX FROM posts

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

Conclusion

In this article, we understood how the elementary concepts and terms of RDBMS/SQL relate in MongoDB. We looked upon designing relationships in MongoDB and learnt how the functionality of basic SQL queries map in MongoDB.

After getting a head start with this article, you can go ahead trying out complex queries including aggregation, map reduce and queries involving multiple collections. You can also take help of some online tools to convert SQL queries to MongoDB queries in the beginning. You can play designing a sample MongoDB database schema on your own. One of the best examples to do so would be a database to store user posts, their likes, comments and comment likes. This would give you a practical view of the flexible schema design that MongoDB offers.

Feel free to comment any suggestions, questions or ideas you would like to see further.

Ashish Trivedi

Ashish is a Computer Science Engineer and works for one of the Big Four firms as a Technology Analyst. Prior to this, he co-founded an educational start-up. In his free time, Ashish read and write blogs, watches animation cartoons, movies or discovery channel. You can find his recent blogs on http://ashishtrivediblog.wordpress.com/

Advertisement

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

29 Comments **Nettuts+** 1 **Login**

Recommend 12 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Leonardo Rothe Tagliafico • 5 years ago

We all knew that. Now tell us how to do map joins to MapReduce.

24 • Reply • Share ›

Sam Parker D Leonardo Rothe Tagliafico • 5 years ago good luck with that lol.

2 ^ | v • Reply • Share ›

expediteA • 5 years ago

On the face of it, NoSQL seems the better option, but I worry that it promotes a lack of planning when considering data schemas, where you will end up with lots of users with lots of different fields and your data gets into a massive mess.

9 ^ | v • Reply • Share ›

Michael Lawson ➦ expediteA • 5 years ago

Agreed completely. Not sure why my comment of a similar nature got downvoted >.>

2 ^ | v • Reply • Share ›

Michael Lawson • 5 years ago

"The fields can be easily added, removed and modified anytime. Also,
there is no constraint on data types of the fields. Thus, at one
instance a field can hold int type data and at the next instance it may hold an array."

That's not just different, that's horrifying. There is a reason for these sorts of restraints being in place in a regular ol RDBMS. This is absolutely no replacement for good design at the roots, not this assumed scalability by virtue of being. I think the, otherwise fabulous, tutorial should make that clear.

8 ^ | v • Reply • Share ›

gplocke • 5 years ago

In the first paragraph of Mapping Tables, Rows, and Columns, it says ". Lastly, the way we have rows in an SQL row, we have fields in MongoDB." Shouldn't that be "the way we have columns in an SQL row"?

3 ^ | v • Reply • Share ›

Raymond Camden • 5 years ago

Just FYI, the first code example seems to have been corrupted.

2 ^ | v • Reply • Share ›

arpit bhatt • a year ago

how to convert the following query
SELECT * FROM my_db.user where DATE(creation_date) between '2015-11-03' and '2015-11-04';

1 ^ | v • Reply • Share ›

ascotto • 5 years ago

Well, nice tut about basics. We are building our application for the cloud using MongoDB, what can I say is that not all that shines is gold :) You have to be careful with "transactions" in MongoDB, and how to properly use MapReduce. But anyway we are pretty satisfied with it, btw we are building this app with MongoDB https://wellioo.com/

1 ^ | v • Reply • Share ›

Anlek • 5 years ago

If you want to move your data from Sql to MongoDB, you should checkout our project at http://mongify.com. It updates the IDs and related IDs, supports embedding and much more.

1 • Reply • Share ›

Dagg • 9 months ago

Very clean and clear explanation. Thank you for avoiding using pompous 'technical' jargon only to appear smart (which happens too often in IT)

• Reply • Share ›

kamala illangovan • a year ago

Hi,.. I am new to mongodb and node js. How to query a collection, with the list of a column values retrieved from another collection?
Thanks!!

• Reply • Share ›

muhammad zubair • 2 years ago

MongoDB Relationship step by step guideline its new ans simple detail
http://programmershelper.co...

• Reply • Share ›

niharika • 3 years ago

can you please tell me how can we insert multiple records not just one record in one go

• Reply • Share ›

Arfan Ali niharika • 2 years ago

```
db.post.insert([
{ title: 'MongoDB Overview', likes:100},
{ title: 'MongoDB Overview1', likes:200},
{ title: 'MongoDB Overview2', likes:300}
]);
```

• Reply • Share ›

arpit bhatt Arfan Ali • a year ago

how to convert the following query

```
SELECT * FROM my_db.user where DATE(creation_date) between '2015-11-03' and '2015-11-04';
```

• Reply • Share ›

Farashida Nazari • 3 years ago

I have some problem to filter a certain date range in MongoDB using SQL, anybody assist on that? thanks~

• Reply • Share ›

Vikram Pawar • 3 years ago

what are the tools available in market to convert SQL to MongoDB ?

^ | ˅ • Reply • Share ›

siddhesh Pujare • 3 years ago

can we perform join in mongodb???

^ | ˅ • Reply • Share ›

joshbeam ➦ siddhesh Pujare • 3 years ago

Yes now you can: https://docs.mongodb.org/ma...

^ | ˅ • Reply • Share ›

Roberto Andrew • 4 years ago

Diagrams for MongoDB:

MongoDB stores information based on JSON documents. The complexity of the documents increase if the number of sub-documents increase. Mastering such a documents using diagrams may make work much easier.
This will be represented in DbSchema as three entities, one for each sub-document.
DbSchema does discover the schema by scanning the data.

^ | ˅ • Reply • Share ›

Roberto Andrew • 4 years ago

Recently I discovered a tool simply called DbSchema (http://www.dbschema.com). First of all I was impressed because they do diagrams for MongoDB.
Second I found great an data explorer from them, where you can explore data from each collection and sub-documents in a separate window.Now I am dealing with virtual foreign key from them, to explore data from two collections bind referencing one the other via ObjectId's.

Look for DbSchema tool, is great for the diagrams they do for MongoDB, query builder and data explorer.Some features you may discover inside like virtual foreign keys makes the interaction really similar with relational databases, where you can place data in multiple collections and join it with ObjectId's.

I was surprised to see is possible to have diagrams for MongoDB as well, as for any relational database.Go for the tool DbSchema. Have a look on relational data browse and the virtual foreign keys there, they are a step forward in designing a database with data over multiple collections and references between them via ObjectId's.

^ | ˅ • Reply • Share ›

chitopolo • 4 years ago

This is a very useful tutorial, thank you very much! ;)

^ | ˅ • Reply • Share ›

teknobabble • 4 years ago

Every time an image is used for a code snippet, a kitten dies. Also, the 3rd one refers to some MongoId class, instead of ObjectId. I assume that this is referencing the mongoDB PHP extension? You should mention that.

^ | ˅ • Reply • Share ›

Don Bosco • 5 years ago

Thank you for writing this tutorial. I think that this is in general a good tutorial if you want to introduce the MongoDB Terms to SQL users.

But i expected something different when reading the headline. The main challenging tasks are: How should i design my data structures. There are a lot of barriers. It is not only about mapping or converting the schema but you have to think in a different way. Besides that you also have business critical criterias like "how do i handle transactions" (ACID). I also miss all the benefits of constraints and referential integrity. These are the topics i really would like to read when i saw the headline. Anyway it is a well written good introduction but imho any developer will figure that out in an hour.

^ | ˅ • Reply • Share ›

Ramesh Reddy • 5 years ago

Good tutorial, I read with great interest. You all need to check out http://teiid.org where I have written a translator for MongoDB which shows all above in practice, using it you can do all the above and more. Yes you can even do JOINS. If you know SQL you can use it, you do not need to learn any MongoDB query syntax.

^ | ˅ • Reply • Share ›

Advertisement