MODUL PRAKTIKUM

DATABASE OBJEK TERDISTRIBUSI (DOT)

OLEH

IR. NIZIRWAN ANWAR, MT

TRI ISMARDIKO WIDYAWAN, S.KOM, M.KOM

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS ESA UNGGUL
2017

DAFTAR ISI

MODUL PRAKTIKUM

MODUL 9

DATABASE OBJEK TER-DISTRIBUSI

24. MongoDB – PHP

To use MongoDB with PHP, you need to use MongoDB PHP driver. Download the driver from the url Download PHP Driver. Make sure to download the latest release of it. Now unzip the archive and put php_mongo.dll in your PHP extension directory ("ext" by default) and add the following line to your php.ini file –

```
extension = php_mongo.dll
```

Make a Connection and Select a Database

To make a connection, you need to specify the database name, if the database doesn't exist then MongoDB creates it automatically.

Following is the code snippet to connect to the database –

```
<?php
   // connect to mongodb
   $m = new MongoClient();

   echo "Connection to database successfully";
   // select a database
   $db = $m->mydb;

   echo "Database mydb selected";
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
```

Create a Collection

Following is the code snippet to create a collection –

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->createCollection("mycol");
   echo "Collection created successfully";
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection created successfully
```

Insert a Document

To insert a document into MongoDB, **insert()** method is used.

Following is the code snippet to insert a document –

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected successfully";
   $document = array(
      "title" => "MongoDB",
      "description" => "database",
      "likes" => 100,
      "url" => "http://www.tutorialspoint.com/mongodb/",
      "by", "tutorials point"
   );
   $collection->insert($document);
   echo "Document inserted successfully";
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected successfully
Document inserted successfully
```

Find All Documents

To select all documents from the collection, find() method is used.

Following is the code snippet to select all documents –

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected successfully";

   $cursor = $collection->find();
   // iterate cursor to display title of documents

   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected successfully
{
   "title": "MongoDB"
}
```

Update a Document

To update a document, you need to use the update() method.

In the following example, we will update the title of inserted document to **MongoDB Tutorial**. Following is the code snippet to update a document –

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   // now update the document
   $collection->update(array("title"=>"MongoDB"),
      array('$set'=>array("title"=>"MongoDB Tutorial")));
   echo "Document updated successfully";

   // now display the updated document
   $cursor = $collection->find();

   // iterate cursor to display title of documents
   echo "Updated document";

   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Document updated successfully
Updated document
{
   "title": "MongoDB Tutorial"
}
```

Delete a Document

To delete a document, you need to use remove() method.

In the following example, we will remove the documents that has the title **MongoDB Tutorial**. Following is the code snippet to delete a document –

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   // now remove the document
   $collection->remove(array("title"=>"MongoDB Tutorial"),false);
   echo "Documents deleted successfully";
   // now display the available documents
   $cursor = $collection->find();

   // iterate cursor to display title of documents
   echo "Updated document";
   foreach ($cursor as $document) {
      echo $document["title"] . "\n";   }
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Documents deleted successfully
```

In the above example, the second parameter is boolean type and used for **justOne** field of **remove()** method.

Remaining MongoDB methods **findOne(), save(), limit(), skip(), sort()** etc. works same as explained above.

Advanced MongoDB

MODUL 10

DATABASE OBJEK TER-DISTRIBUSI

25. MongoDB – Relationships

Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via **Embedded** and **Referenced** approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.

Let us consider the case of storing addresses for users. So, one user can have multiple addresses making this a 1:N relationship.

Following is the sample document structure of **user** document –

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "name": "Tom Hanks",
   "contact": "987654321",
   "dob": "01-01-1991"
}
```

Following is the sample document structure of **address** document –

```
{
   "_id":ObjectId("52ffc4a5d85242602e000000"),
   "building": "22 A, Indiana Apt",
   "pincode": 123456,
   "city": "Los Angeles",
   "state": "California"
}
```

Modeling Embedded Relationships

In the embedded approach, we will embed the address document inside the user document.

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin",
   "address": [
      {
         "building": "22 A, Indiana Apt",
         "pincode": 123456,
         "city": "Los Angeles",
         "state": "California"
      },
      {
         "building": "170 A, Acropolis Apt",
         "pincode": 456789,
         "city": "Chicago",
         "state": "Illinois"
      }
   ]
}
```

This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as –

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

Note that in the above query, **db** and **users** are the database and collection respectively.

The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

Modeling Referenced Relationships

This is the approach of designing normalized relationship. In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's **id** field.

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin",
   "address_ids": [
      ObjectId("52ffc4a5d85242602e000000"),
      ObjectId("52ffc4a5d85242602e000001")
   ]}
```

As shown above, the user document contains the array field **address_ids** which contains ObjectIds of corresponding addresses. Using these ObjectIds, we can query the address documents and get address details from there. With this approach, we will need two queries: first to fetch the **address_ids** fields from **user** document and second to fetch these addresses from **address** collection.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

26. MongoDB – Database References

As seen in the last chapter of MongoDB relationships, to implement a normalized database structure in MongoDB, we use the concept of **Referenced Relationships** also referred to as **Manual References** in which we manually store the referenced document's id inside other document. However, in cases where a document contains references from different collections, we can use **MongoDB DBRefs**.

DBRefs vs Manual References

As an example scenario, where we would use DBRefs instead of manual references, consider a database where we are storing different types of addresses (home, office, mailing, etc.) in different collections (address_home, address_office, address_mailing, etc). Now, when a **user** collection's document references an address, it also needs to specify which collection to look into based on the address type. In such scenarios where a document references documents from many collections, we should use DBRefs.

Using DBRefs

There are three fields in DBRefs:

- $**ref**: This field specifies the collection of the referenced document
- **$id**: This field specifies the _id field of the referenced document
- **$db**: This is an optional field and contains the name of the database in which the referenced document lies

Consider a sample user document having DBRef field **address** as shown in the code snippet:

```
{
   "_id":ObjectId("53402597d852426020000002"),
   "address": {
   "$ref": "address_home",
   "$id": ObjectId("534009e4d852427820000002"),
   "$db": "tutorialspoint"},
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin"
}
```

The **address** DBRef field here specifies that the referenced address document lies in **address_home** collection under **tutorialspoint** database and has an id of 534009e4d852427820000002.

The following code dynamically looks in the collection specified by **$ref**parameter (**address_home** in our case) for a document with id as specified by **$id** parameter in DBRef.

```
>var user = db.users.findOne({"name":"Tom Benzamin"})
>var dbRef = user.address
>db[dbRef.$ref].findOne({"_id":(dbRef.$id)})
```

The above code returns the following address document present in **address_home** collection:

```
{
   "_id" : ObjectId("534009e4d852427820000002"),
   "building" : "22 A, Indiana Apt",
   "pincode" : 123456,
   "city" : "Los Angeles",
   "state" : "California"
}
```

27. MongoDB – Covered Queries

In this chapter, we will learn about covered queries.

What is a Covered Query?

As per the official MongoDB documentation, a covered query is a query in which:

- All the fields in the query are part of an index.
- All the fields returned in the query are in the same index.

Since all the fields present in the query are part of an index, MongoDB matches the query conditions and returns the result using the same index without actually looking inside the documents. Since indexes are present in RAM, fetching data from indexes is much faster as compared to fetching data by scanning documents.

Using Covered Queries

To test covered queries, consider the following document in the **users** collection:

```
{
   "_id": ObjectId("53402597d852426020000002"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "gender": "M",
   "name": "Tom Benzamin",
   "user_name": "tombenzamin"
}
```

We will first create a compound index for the **users** collection on the fields **gender** and **user_name** using the following query:

```
>db.users.ensureIndex({gender:1,user_name:1})
```

Now, this index will cover the following query:

```
>db.users.find({gender:"M"},{user_name:1,_id:0})
```

That is to say that for the above query, MongoDB would not go looking into database documents. Instead it would fetch the required data from indexed data which is very fast.

Since our index does not include **_id** field, we have explicitly excluded it from result set of our query, as MongoDB by default returns _id field in every query. So the following query would not have been covered inside the index created above:

```
>db.users.find({gender:"M"},{user_name:1})
```

Lastly, remember that an index cannot cover a query if:

- Any of the indexed fields is an array
- Any of the indexed fields is a subdocument

MODUL 11

DATABASE OBJEK TER-DISTRIBUSI

28. MongoDB – Analyzing Queries

Analyzing queries is a very important aspect of measuring how effective the database and indexing design is. We will learn about the frequently used **$explain** and **$hint** queries.

Using $explain

The **$explain** operator provides information on the query, indexes used in a query and other statistics. It is very useful when analyzing how well your indexes are optimized.

In the last chapter, we had already created an index for the **users** collection on fields **gender** and **user_name** using the following query:

```
>db.users.ensureIndex({gender:1,user_name:1})
```

We will now use **$explain** on the following query:

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).explain()
```

The above explain() query returns the following analyzed result:

```
{
   "cursor" : "BtreeCursor gender_1_user_name_1",
   "isMultiKey" : false,
   "n" : 1,
   "nscannedObjects" : 0,
   "nscanned" : 1,
   "nscannedObjectsAllPlans" : 0,
   "nscannedAllPlans" : 1,
   "scanAndOrder" : false,
   "indexOnly" : true,
   "nYields" : 0,
   "nChunkSkips" : 0,
   "millis" : 0,
   "indexBounds" : {
      "gender" : [
         [
            "M",
            "M"
         ]
      ],
      "user_name" : [
            [
                {
                    "$minElement" : 1
                },
                {
                    "$maxElement" : 1
                }
            ]
        ]
    }
}
```

We will now look at the fields in this result set:

- The true value of indexOnly indicates that this query has used indexing.
- The cursor field specifies the type of cursor used. BTreeCursor type indicates that an index was used and also gives the name of the index used. BasicCursor indicates that a full scan was made without using any indexes.
- **n** indicates the number of documents matching returned.
- **nscannedObjects** indicates the total number of documents scanned.
- **nscanned** indicates the total number of documents or index entries scanned.

Using $hint

The **$hint** operator forces the query optimizer to use the specified index to run a query. This is particularly useful when you want to test performance of a query with different indexes. For example, the following query specifies the index on fields **gender** and **user_name** to be used for this query:

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1})
```

To analyze the above query using $explain:

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1}).e
xplain()
```

29. MongoDB – Atomic Operations

MongoDB does not support **multi-document atomic transactions**. However, it does provide atomic operations on a single document. So if a document has hundred fields the update statement will either update all the fields or none, hence maintaining atomicity at the document-level.

Model Data for Atomic Operations

The recommended approach to maintain atomicity would be to keep all the related information, which is frequently updated together in a single document using **embedded documents**. This would make sure that all the updates for a single document are atomic.

Consider the following products document:

```
{
   "_id":1,
   "product_name": "Samsung S3",
   "category": "mobiles",
   "product_total": 5,
   "product_available": 3,
   "product_bought_by": [
      {
         "customer": "john",
         "date": "7-Jan-2014"
      },
      {
         "customer": "mark",
         "date": "8-Jan-2014"
      }
   ]
}
```

In this document, we have embedded the information of the customer who buys the product in the **product_bought_by** field. Now, whenever a new customer buys the product, we will first check if the product is still available using **product_available** field. If available, we will reduce the value of product_available field as well as insert the new customer's embedded document in the product_bought_by field. We will use **findAndModify** command for this functionality because it searches and updates the document in the same go.

```
>db.products.findAndModify({
   query:{_id:2,product_available:{$gt:0}},
   update:{
      $inc:{product_available:-1},
      $push:{product_bought_by:{customer:"rob",date:"9-Jan-2014"}}
   }
})
```

Our approach of embedded document and using findAndModify query makes sure that the product purchase information is updated only if it the product is available. And the whole of this transaction being in the same query, is atomic.

In contrast to this, consider the scenario where we may have kept the product availability and the information on who has bought the product, separately. In this case, we will first check if the product is available using the first query. Then in the second query we will update the purchase information. However, it is possible that between the executions of these two queries, some other user has purchased the product and it is no more available. Without knowing this, our second query will update the purchase information based on the result of our first query. This will make the database inconsistent because we have sold a product which is not available.

30. MongoDB – Advanced Indexing

Consider the following document of the **users** collection:

```
{
   "address": {
      "city": "Los Angeles",
      "state": "California",
      "pincode": "123"
   },
   "tags": [
      "music",
      "cricket",
      "blogs"
   ],
   "name": "Tom Benzamin"
}
```

The above document contains an **address sub-document** and a **tags array**.

Indexing Array Fields

Suppose we want to search user documents based on the user's tags. For this, we will create an index on tags array in the collection.

Creating an index on array in turn creates separate index entries for each of its fields. So in our case when we create an index on tags array, separate indexes will be created for its values music, cricket and blogs.

To create an index on tags array, use the following code:

```
>db.users.ensureIndex({"tags":1})
```

After creating the index, we can search on the tags field of the collection like this:

```
>db.users.find({tags:"cricket"})
```

To verify that proper indexing is used, use the following **explain** command:

```
>db.users.find({tags:"cricket"}).explain()
```

The above command resulted in "cursor" : "BtreeCursor tags_1" which confirms that proper indexing is used.

Indexing Sub-Document Fields

Suppose that we want to search documents based on city, state and pincode fields. Since all these fields are part of address sub-document field, we will create an index on all the fields of the sub-document.

For creating an index on all the three fields of the sub-document, use the following code:

```
>db.users.ensureIndex({"address.city":1,"address.state":1,"address.pincode":1})
```

Once the index is created, we can search for any of the sub-document fields utilizing this index as follows:

```
>db.users.find({"address.city":"Los Angeles"})
```

Remember that the query expression has to follow the order of the index specified. So the index created above would support the following queries:

```
>db.users.find({"address.city":"Los Angeles","address.state":"California"})
```

It will also support the following query:

```
>db.users.find({"address.city":"LosAngeles","address.state":"California","addre
ss.pincode":
```

31. MongoDB – Indexing Limitations

In this chapter, we will learn about Indexing Limitations and its other components.

Extra Overhead

Every index occupies some space as well as causes an overhead on each insert, update and delete. So if you rarely use your collection for read operations, it makes sense not to use indexes.

RAM Usage

Since indexes are stored in RAM, you should make sure that the total size of the index does not exceed the RAM limit. If the total size increases the RAM size, it will start deleting some indexes, causing performance loss.

Query Limitations

Indexing can't be used in queries which use:

- Regular expressions or negation operators like $nin, $not, etc.
- Arithmetic operators like $mod, etc.
- $where clause

Hence, it is always advisable to check the index usage for your queries.

Index Key Limits

Starting from version 2.6, MongoDB will not create an index if the value of existing index field exceeds the index key limit.

Inserting Documents Exceeding Index Key Limit

MongoDB will not insert any document into an indexed collection if the indexed field value of this document exceeds the index key limit. Same is the case with mongorestore and mongoimport utilities.

Maximum Ranges

- A collection cannot have more than 64 indexes.
- The length of the index name cannot be longer than 125 characters.
- A compound index can have maximum 31 fields indexed.

32. MongoDB – ObjectId

We have been using MongoDB Object Id in all the previous chapters. In this chapter, we will understand the structure of ObjectId.

An **ObjectId** is a 12-byte BSON type having the following structure:

- The first 4 bytes representing the seconds since the unix epoch
- The next 3 bytes are the machine identifier
- The next 2 bytes consists of process id
- The last 3 bytes are a random counter value

MongoDB uses ObjectIds as the default value of **_id** field of each document, which is generated while the creation of any document. The complex combination of ObjectId makes all the _id fields unique.

Creating New ObjectId

To generate a new ObjectId use the following code:

```
>newObjectId = ObjectId()
```

The above statement returned the following uniquely generated id:

```
ObjectId("5349b4ddd2781d08c09890f3")
```

Instead of MongoDB generating the ObjectId, you can also provide a 12-byte id:

```
>myObjectId = ObjectId("5349b4ddd2781d08c09890f4")
```

Creating Timestamp of a Document

Since the _id ObjectId by default stores the 4-byte timestamp, in most cases you do not need to store the creation time of any document. You can fetch the creation time of a document using getTimestamp method:

```
>ObjectId("5349b4ddd2781d08c09890f4").getTimestamp()
```

This will return the creation time of this document in ISO date format:

```
ISODate("2014-04-12T21:49:17Z")
```

Converting ObjectId to String

In some cases, you may need the value of ObjectId in a string format. To convert the ObjectId in string, use the following code:

```
>newObjectId.str
```

The above code will return the string format of the Guid:

```
5349b4ddd2781d08c09890f3
```

MODUL 12

DATABASE OBJEK TER-DISTRIBUSI

33. MongoDB – MapReduce

As per the MongoDB documentation, **MapReduce** is a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses **mapReduce** command for map-reduce operations. MapReduce is generally used for processing large data sets.

MapReduce Command

Following is the syntax of the basic mapReduce command –

```
>db.collection.mapReduce(
   function() {emit(key,value);},  //map function
   function(key,values) {return reduceFunction}, {   //reduce function
      out: collection,
      query: document,
      sort: document,
      limit: number
   }
)
```

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs, which is then reduced based on the keys that have multiple values.

In the above syntax -

- **map** is a javascript function that maps a value with a key and emits a key-value pair
- **reduce** is a javascript function that reduces or groups all the documents having the same key
- **out** specifies the location of the map-reduce query result
- **query** specifies the optional selection criteria for selecting documents
- **sort** specifies the optional sort criteria
- **limit** specifies the optional maximum number of documents to be returned

Using MapReduce

Consider the following document structure storing user posts. The document stores user_name of the user and the status of post.

```
{
   "post_text": "tutorialspoint is an awesome website for tutorials",
   "user_name": "mark",
   "status":"active"
}
```

Now, we will use a mapReduce function on our **posts** collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code –

```
>db.posts.mapReduce(
   function() { emit(this.user_id,1); },
   function(key, values) {return Array.sum(values)}, {
      query:{status:"active"},
      out:"post_total"
   }
)
```

The above mapReduce query outputs the following result –

```
{
   "result" : "post_total",
   "timeMillis" : 9,
   "counts" : {
      "input" : 4,
      "emit" : 4,
      "reduce" : 2,
      "output" : 2
   },
   "ok" : 1,
}
```

The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.

To see the result of this mapReduce query, use the find operator –

```
>db.posts.mapReduce(
   function() { emit(this.user_id,1); },
   function(key, values) {return Array.sum(values)}, {
      query:{status:"active"},
      out:"post_total"
   }
).find()
```

The above query gives the following result which indicates that both users **tom** and **mark** have two posts in active states –

```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

In a similar manner, MapReduce queries can be used to construct large complex aggregation queries. The use of custom Javascript functions make use of MapReduce which is very flexible and powerful.

34. MongoDB – Text Search

Starting from version 2.4, MongoDB started supporting text indexes to search inside string content. The **Text Search** uses stemming techniques to look for specified words in the string fields by dropping stemming stop words like **a, an, the**, etc. At present, MongoDB supports around 15 languages.

Enabling Text Search

Initially, Text Search was an experimental feature but starting from version 2.6, the configuration is enabled by default. But if you are using the previous version of MongoDB, you have to enable text search with the following code:

```
>db.adminCommand({setParameter:true,textSearchEnabled:true})
```

Creating Text Index

Consider the following document under **posts** collection containing the post text and its tags:

```
{
   "post_text": "enjoy the mongodb articles on tutorialspoint",
   "tags": [
      "mongodb",
      "tutorialspoint"
   ]
}
```

We will create a text index on post_text field so that we can search inside our posts' text:

```
>db.posts.ensureIndex({post_text:"text"})
```

Using Text Index

Now that we have created the text index on post_text field, we will search for all the posts having the word **tutorialspoint** in their text.

```
>db.posts.find({$text:{$search:"tutorialspoint"}})
```

The above command returned the following result documents having the word **tutorialspoint** in their post text:

```
{
   "_id" : ObjectId("53493d14d852429c10000002"),
   "post_text" : "enjoy the mongodb articles on tutorialspoint",
   "tags" : [ "mongodb", "tutorialspoint" ]
}
{
   "_id" : ObjectId("53493d1fd852429c10000003"),
   "post_text" : "writing tutorials on mongodb",
   "tags" : [ "mongodb", "tutorial" ]
}
```

If you are using old versions of MongoDB, you have to use the following command:

```
>db.posts.runCommand("text",{search:" tutorialspoint "})
```

Using Text Search highly improves the search efficiency as compared to normal search.

Deleting Text Index

To delete an existing text index, first find the name of index using the following query:

```
>db.posts.getIndexes()
```

After getting the name of your index from above query, run the following command. Here, **post_text_text** is the name of the index.

```
>db.posts.dropIndex("post_text_text")
```

35. MongoDB – Regular Expression

Regular Expressions are frequently used in all languages to search for a pattern or word in any string. MongoDB also provides functionality of regular expression for string pattern matching using the **$regex** operator. MongoDB uses PCRE (Perl Compatible Regular Expression) as regular expression language.

Unlike text search, we do not need to do any configuration or command to use regular expressions.

Consider the following document structure under **posts** collection containing the post text and its tags:

```
{
   "post_text": "enjoy the mongodb articles on tutorialspoint",
   "tags": [
      "mongodb",
      "tutorialspoint"
   ]
}
```

Using regex Expression

The following regex query searches for all the posts containing string **tutorialspoint** in it:

```
>db.posts.find({post_text:{$regex:"tutorialspoint"}})
```

The same query can also be written as:

```
>db.posts.find({post_text:/tutorialspoint/})
```

Using regex Expression with Case Insensitive

To make the search case insensitive, we use the **$options** parameter with value **$i**. The following command will look for strings having the word **tutorialspoint**, irrespective of smaller or capital case:

```
>db.posts.find({post_text:{$regex:"tutorialspoint",$options:"$i"}})
```

One of the results returned from this query is the following document which contains the word **tutorialspoint** in different cases:

```
{
   "_id" : ObjectId("53493d37d852429c10000004"),
   "post_text" : "hey! this is my post on TutorialsPoint",
   "tags" : [ "tutorialspoint" ]
}
```

Using regex for Array Elements

We can also use the concept of regex on array field. This is particularly very important when we implement the functionality of tags. So, if you want to search for all the posts having tags beginning from the word tutorial (either tutorial or tutorials or tutorialpoint or tutorialphp), you can use the following code:

```
>db.posts.find({tags:{$regex:"tutorial"}})
```

Optimizing Regular Expression Queries

- If the document fields are **indexed**, the query will use make use of indexed values to match the regular expression. This makes the search very fast as compared to the regular expression scanning the whole collection.

- If the regular expression is a **prefix expression**, all the matches are meant to start with a certain string characters. For e.g., if the regex expression is **^tut**, then the query has to search for only those strings that begin with **tut**.

MODUL 13

DATABASE OBJEK TER-DISTRIBUSI

36. MongoDB – RockMongo

RockMongo is a MongoDB administration tool using which you can manage your server, databases, collections, documents, indexes, and a lot more. It provides a very user-friendly way for reading, writing, and creating documents. It is similar to PHPMyAdmin tool for PHP and MySQL.

Downloading RockMongo

You can download the latest version of RockMongo from here: http://rockmongo.com/downloads

Installing RockMongo

Once downloaded, you can unzip the package in your server root folder and rename the extracted folder to **rockmongo**. Open any web browser and access the **index.php** page from the folder rockmongo. Enter admin/admin as username/password respectively.

Working with RockMongo

We will now be looking at some basic operations that you can perform with RockMongo.

Creating New Database

To create a new database, click **Databases** tab. Click **Create New Database**. On the next screen, provide the name of the new database and click on **Create**. You will see a new database getting added in the left panel.

Creating New Collection

To create a new collection inside a database, click on that database from the left panel. Click on the **New Collection** link on top. Provide the required name of the collection. Do not worry about the other fields of Is Capped, Size and Max. Click on **Create**. A new collection will be created and you will be able to see it in the left panel.

Creating New Document

To create a new document, click on the collection under which you want to add documents. When you click on a collection, you will be able to see all the documents within that collection listed there. To create a new document, click on the **Insert** link at the top. You can enter the document's data either in JSON or array format and click on **Save**.

Export/Import Data

To import/export data of any collection, click on that collection and then click on **Export/Import** link on the top panel. Follow the next instructions to export your data in a zip format and then import the same zip file to import back data.

37. MongoDB – GridFS

GridFS is the MongoDB specification for storing and retrieving large files such as images, audio files, video files, etc. It is kind of a file system to store files but its data is stored within MongoDB collections. GridFS has the capability to store files even greater than its document size limit of 16MB.

GridFS divides a file into chunks and stores each chunk of data in a separate document, each of maximum size 255k.

GridFS by default uses two collections **fs.files** and **fs.chunks** to store the file's metadata and the chunks. Each chunk is identified by its unique _id ObjectId field. The fs.files severs as a parent document. The **files_id** field in the fs.chunks document links the chunk to its parent.

Following is a sample document of fs.files collection:

```
{
   "filename": "test.txt",
   "chunkSize": NumberInt(261120),
   "uploadDate": ISODate("2014-04-13T11:32:33.557Z"),
   "md5": "7b762939321e146569b07f72c62cca4f",
   "length": NumberInt(646)
}
```

The document specifies the file name, chunk size, uploaded date, and length.

Following is a sample document of fs.chunks document:

```
{
   "files_id": ObjectId("534a75d19f54bfec8a2fe44b"),
   "n": NumberInt(0),
   "data": "Mongo Binary Data"
}
```

Adding Files to GridFS

Now, we will store an mp3 file using GridFS using the **put** command. For this, we will use the **mongofiles.exe** utility present in the bin folder of the MongoDB installation folder.

Open your command prompt, navigate to the mongofiles.exe in the bin folder of MongoDB installation folder and type the following code:

```
>mongofiles.exe -d gridfs put song.mp3
```

Here, **gridfs** is the name of the database in which the file will be stored. If the database is not present, MongoDB will automatically create a new document on the fly. Song.mp3 is the name of the file uploaded. To see the file's document in database, you can use find query:

```
>db.fs.files.find()
```

The above command returned the following document:

```
{
   _id: ObjectId('534a811bf8b4aa4d33fdf94d'),
   filename: "song.mp3",
   chunkSize: 261120,
   uploadDate: new Date(1397391643474), md5:
"e4f53379c909f7bed2e9d631e15c1c41",
   length: 10401959
}
```

We can also see all the chunks present in fs.chunks collection related to the stored file with the following code, using the document id returned in the previous query:

```
>db.fs.chunks.find({files_id:ObjectId('534a811bf8b4aa4d33fdf94d')})
```

In my case, the query returned 40 documents meaning that the whole mp3 document was divided in 40 chunks of data.

38. MongoDB – Capped Collections

Capped collections are fixed-size circular collections that follow the insertion order to support high performance for create, read, and delete operations. By circular, it means that when the fixed size allocated to the collection is exhausted, it will start deleting the oldest document in the collection without providing any explicit commands.

Capped collections restrict updates to the documents if the update results in increased document size. Since capped collections store documents in the order of the disk storage, it ensures that the document size does not increase the size allocated on the disk. Capped collections are best for storing log information, cache data, or any other high volume data.

Creating Capped Collection

To create a capped collection, we use the normal createCollection command but with **capped** option as **true** and specifying the maximum size of collection in bytes.

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000})
```

In addition to collection size, we can also limit the number of documents in the collection using the **max** parameter:

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000,max:1000})
```

If you want to check whether a collection is capped or not, use the following **isCapped** command:

```
>db.cappedLogCollection.isCapped()
```

If there is an existing collection which you are planning to convert to capped, you can do it with the following code:

```
>db.runCommand({"convertToCapped":"posts",size:10000})
```

This code would convert our existing collection **posts** to a capped collection.

Querying Capped Collection

By default, a find query on a capped collection will display results in insertion order. But if you want the documents to be retrieved in reverse order, use the **sort** command as shown in the following code:

```
>db.cappedLogCollection.find().sort({$natural:-1})
```

There are few other important points regarding capped collections worth knowing:

- We cannot delete documents from a capped collection.
- There are no default indexes present in a capped collection, not even on _id field.
- While inserting a new document, MongoDB does not have to actually look for a place to accommodate new document on the disk. It can blindly insert the new document at the tail of the collection. This makes insert operations in capped collections very fast.
- Similarly, while reading documents MongoDB returns the documents in the same order as present on disk. This makes the read operation very fast.

MODUL 14

DATABASE OBJEK TER-DISTRIBUSI

39. MongoDB – Auto-Increment Sequence

MongoDB does not have out-of-the-box auto-increment functionality, like SQL databases. By default, it uses the 12-byte ObjectId for the **_id** field as the primary key to uniquely identify the documents. However, there may be scenarios where we may want the _id field to have some auto-incremented value other than the ObjectId.

Since this is not a default feature in MongoDB, we will programmatically achieve this functionality by using a **counters** collection as suggested by the MongoDB documentation.

Using Counter Collection

Consider the following **products** document. We want the _id field to be an **auto-incremented integer sequence** starting from 1,2,3,4 upto n.

```
{
  "_id":1,
  "product_name": "Apple iPhone",
  "category": "mobiles"
}
```

For this, create a **counters** collection, which will keep track of the last sequence value for all the sequence fields.

```
>db.createCollection("counters")
```

Now, we will insert the following document in the counters collection with **productid** as its key –

```
{
  "_id":"productid",
  "sequence_value": 0
}
```

The field **sequence_value** keeps track of the last value of the sequence.

Use the following code to insert this sequence document in the counters collection –

```
>db.counters.insert({_id:"productid",sequence_value:0})
```

Creating Javascript Function

Now, we will create a function **getNextSequenceValue** which will take the sequence name as its input, increment the sequence number by 1 and return the updated sequence number. In our case, the sequence name is **productid**.

```
>function getNextSequenceValue(sequenceName){

   var sequenceDocument = db.counters.findAndModify({
      query:{_id: sequenceName },
      update: {$inc:{sequence_value:1}},
      new:true
   });

   return sequenceDocument.sequence_value;
}
```

Using the Javascript Function

We will now use the function getNextSequenceValue while creating a new document and assigning the returned sequence value as document's _id field.

Insert two sample documents using the following code –

```
>db.products.insert({
   "_id":getNextSequenceValue("productid"),
   "product_name":"Apple iPhone",
   "category":"mobiles"
})

>db.products.insert({
   "_id":getNextSequenceValue("productid"),
   "product_name":"Samsung S3",
   "category":"mobiles"
})
```

As you can see, we have used the getNextSequenceValue function to set value for the _id field.

To verify the functionality, let us fetch the documents using find command –

```
>db.prodcuts.find()
```

The above query returned the following documents having the auto-incremented _id field

```
{ "_id" : 1, "product_name" : "Apple iPhone", "category" : "mobiles"}

{ "_id" : 2, "product_name" : "Samsung S3", "category" : "mobiles" }
```

DAFTAR PUSTAKA

MongoDB Tutorial https://www.tutorialspoint.com/mongodb/ di-akses tanggal 31 Agustus 2017 di Jakarta

P M. Tamer Özsu and Patrick Valduriez, (2011), Principles of Distributed Database Systems, Third Edition, Springer Publishing ISBN 978-1-4419-8833-1

Ajay D. Kshemkalyani and Mukesh Singhal, (2008), Distributed Computing - Principles, Algorithms, and Systems, Cambridge University Press, ISBN-13 978-0-511-39341-9

DAFTAR PUSTAKA

MongoDB Tutorial https://www.tutorialspoint.com/mongodb/ di-akses tanggal 31 Agustus 2017 di Jakarta

P M. Tamer Özsu and Patrick Valduriez, (2011), Principles of Distributed Database Systems, Third Edition, Springer Publishing ISBN 978-1-4419-8833-1

Ajay D. Kshemkalyani and Mukesh Singhal, (2008), Distributed Computing - Principles, Algorithms, and Systems, Cambridge University Press, ISBN-13 978-0-511-39341-9