Comparative analysis of simple and aggregate query in MongoDB

Mukesh Babulal Arethiya
mukesh.arethiya@ves.ac.in
Vivekanand Education Society's Institute of Technology, Mumbai, Maharashtra

Meenakshi Garg
meenakshi.garg@ves.ac.in
Vivekanand Education Society's Institute of Technology, Mumbai, Maharashtra

ABSTRACT

This comparative study is not only important for Enterprises and Business but also for all those Organizations which stores and query the database frequently from a huge volume of data. Purpose of this study is to find performance effective way to query the resources from MongoDB with 2 different model patterns.

Keywords: MongoDB, DB.collection.aggregate([]), DB.collection.find()

1. INTRODUCTION

In Business and IT domains hypes oscillate frequently, and now it is developer’s responsibility to find what the best way to work with the latest technology is.

MongoDB was founded in 2007 by Dwight Merriman, Eliot Horowitz, and Kevin Ryan – the team behind DoubleClick. MongoDB is one of the trending open-source document-based database application. MongoDB stores each record as a document, and a group of the document makes a collection. Modern application require Big Data. Fast feature development and flexible deployment, these features are provided by MongoDB [1].

In comparison with RDBMS (Relational Database Management Systems), MongoDB can scale higher with limited space and gives better performance for simple queries, whereas in RDBMS as data grows performance of query get reduced. RDBMS is more focused on availability whereas MongoDB is focused on partition-tolerance. No specialized hardware or platform required to run a MongoDB instance it is built to run over commodity hardware, RDBMS databases require schema to store data in a fixed format whereas MongoDB doesn’t require any predefined schema that’s why it’s more flexible but this flexibility comes at cost that is application developer has to define constraints, referential integrity at application layer, because MongoDB accepts any kind of unstructured data, whereas in case of RDBMS constraint plays major role in validating data. Since MongoDB is distributed in nature i.e. user can use auto-sharding capabilities to evenly partition the data in a horizontal fashion across multiple replica sets. MongoDB works better with denormalized data because MongoDB isn’t designed for joins and normalized data.

With the introduction of microservices and containers applications are decomposed into several parts based on business function, each microservice has its own database so finding the best way to query database is necessary which can lead to the rational use of computing and memory.

To retrieve relevant data from the collection of documents, MongoDB provides find method [2] which comes under simple queries, whereas in order to calculate aggregate value aggregate method is used which comes under MongoDB’s Aggregation Framework [3].

The aggregate method calculates the aggregate value by executing series of pipeline operations where one operation’s output will be input for next operation. One of the aggregation operations include match operation which selects the documents that satisfy match condition(s) [4].

2. OBJECTIVE

MongoDB is preferred for the high data retrieval intensive task so it’s reasonable to squeeze more performance out of different type of queries.

In this paper we will compare performance of following methods:

a. db.collection.find()
b. db.collection.aggregate({$match})

These above methods will be executed on predefined sets of data available through “Mackaroo”, First experiment will be on smaller dataset then after using the large dataset to compare query execution speed in milliseconds then after modeling dataset to denormalized form to fully utilize a feature of aggregation framework.
3. RELATED WORKS

Addullah Talha Kabakus has written a journal article on “A performance evaluation of in-memory databases”, in which author has compared MongoDB (Document store) with other databases like H2 (RDBMS), Redis (Key-value store), Memcached (Key-value store) and Cassandra (Column store). Author has experimented with different operations like insert, update, delete, read queries on each of the above-mentioned databases and provided detail analysis of the same. [2]

Bartholomew compares SQL and NoSQL databases with providing a brief history and the use case of each one. [2]

Boicea et al. compare MongoDB and Oracle databases in order to compare NoSQL and SQL database performance through the three experiments: (1) Elapsed time to insert data, (2) elapsed time to update data, and (3) elapsed time to delete data. [7]

4. STEPS FOR COMPARISON EXPERIMENT

i. Preparation of Datasets
ii. Execution of Methods on Datasets
iii. Comparison of Execution Statistics

4.1 Preparation of Datasets
This study uses 2 kinds of datasets
i. Simple Person Collection
ii. Aggregated Collection of countries, states, and cities

4.1.1 Simple Person Collection
Each document in person collection will have following fields:

```json
{
  id: Number,
  first_name: String,
  last_name: String,
  email: String,
  gender: String(“Male”, “Female”)
}
```

Exactly 2000 JSON (JavaScript Object Notation) documents are randomly generated using “mockaroo” [5]

4.1.2 Aggregated Collection
This aggregated collection involves 3 collection creation:

i. Countries collection with following fields:
   
   ```json
   {
     _id: Number,
     sortname: String,
     name: String
   }
   ```

ii. States collection with following fields:
   
   ```json
   {
     _id: Number,
     name: String,
     country_id: Number
   }
   ```

iii. Cities collection with following fields:
   
   ```json
   {
     _id: Number,
     name: String,
     state_id: Number
   }
   ```

These above collections of data are contributed by Rohit Kumar [6].

4.2 Execution of Methods on Datasets
In this section find and aggregate, the method has executed on Person Collection and Cities Collection.

System Configuration for running these tests is:
Processor: Intel i3 4005U @1.70Ghz
RAM: 4 GB
OS: Windows 7 Professional 64 bit

4.2.1 Db.collection.find() on Person Collection
Condition is find the document in which first name is “Saree”

Query:

```javascript
db.person.find({ first_name: "Saree" }).explain( "executionStats" )
```

Result (executionStats):

```javascript
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 1,
  "executionTimeMillis" : 4,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 2000,
  "executionStages" : {
    "stage" : "COLLSCAN",
    "filter" : {
      "first_name" : { "$eq" : "Saree" }
    },
    "nReturned" : 1,
    "executionTimeMillisEstimate" : 0,
    "works" : 2002,
    "advanced" : 1,
    "needTime" : 2000,
    "needYield" : 0,
    "saveState" : 15,
    "restoreState" : 15,
    "isEOF" : 1,
    "invalidates" : 0,
    "direction" : "forward",
    "docsExamined" : 2000
  }
}
```

4.2.2 Db.collection.aggregate([$match]) on Person Collection

Query:

```javascript
db.person.explain("executionStats")
.aggregate([$match: { first_name: "Saree" }])
```

Result (executionStats):

```javascript
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 1,
  "executionTimeMillis" : 3,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 2000,
  "executionStages" : {
    "stage" : "COLLSCAN",
    "filter" : {
      "first_name" : { "$eq" : "Saree" }
    },
    "nReturned" : 1,
    "executionTimeMillisEstimate" : 0,
    "works" : 2002,
    "advanced" : 1,
    "needTime" : 2000,
    "needYield" : 0,
    "saveState" : 15,
    "restoreState" : 15,
    "isEOF" : 1,
    "invalidates" : 0,
    "direction" : "forward",
    "docsExamined" : 2000
  }
}
```
4.2.3 Db.collection.find() on Cities Collection

This Collection has 48313 documents. Setup condition is found in a city by the name “Pelham”.

Query:

db.cities.find({name:"Pelham"}).explain("executionStats")

Result (executionStats):


4.2.4 Db.collection.aggregate({$match}) on Cities Collection

Query:

db.cities.explain("executionStats") .aggregate([{$match:{name:"Pelham"}}])

Result (executionStats):


4.3 Comparison of Execution Statistics

<table>
<thead>
<tr>
<th>Collection</th>
<th>No. of documents examined</th>
<th>Average execution time (in mills)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>2000</td>
<td>4</td>
</tr>
<tr>
<td>Cities</td>
<td>48313</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>52</td>
</tr>
</tbody>
</table>

Both of the method’s average execution time was insignificant when No. of documents was 2000 but difference became significant when no. of documents changed to 48313 and because of MongoDB Caches query result, that’s why need to restart the service to check the initial execution statistics.

5. IMPROVING PERFORMANCE OF QUERY BY CHANGING MODEL STRUCTURE

Query execution time reduces drastically when the data model is in the hierarchy.

5.1 Creating Aggregated Collection

The study has referred datasets from section 4.1.2. Derived collection location expected to have following model structure:

```
{
   _id:{
      country_id:Number,
      name: String,
      sortname: String
   },
   states:[
      {
         _id: Number,
         name: String,
         cities:
            [
               {
                  _id: Number,
                  name: String
               }
            ]
      }
   ]
}
```

In order to create such collection we have to use MongoDB’s aggregation pipeline operations.
Such as $lookup for referring another collection in the same database, Sunwind for undoing array of embedded document, a $group for grouping similar document in an array on some condition and $out to create a collection.

Following query will make aggregated collection in database:

Query:
```javascript
db.countries.aggregate([{
    $lookup: {from: 'states', localField: '_id', foreignField: 'country_id', as: 'states'}],
    $Sunwind: ['$states'],
    $lookup: {from: 'cities', localField: 'states._id', foreignField: 'state_id', as: 'states.cities'}],
    $group: {$_id: {country_id: '$_id', name: '$name', sortname: '$sortname'}, states: {$push: '$states'}},
    $out: "locations"
})
```

5.2 Execution of aggregate method to find interested document from aggregated structure

Following query has been used to find city document based on name of country as “Mexico”, state as “Michoacan” and city as “Coalcoman”.

Query:
```javascript
db.locations.aggregate([{
    $match: {'_id.name': 'Mexico'}},
    $Sunwind: '$states'},
    $match: {'states.name': 'Michoacan'}],
    $Sunwind: '$states.cities'},
    $match: {'states.cities.name': 'Coalcoman'})
})
```

Result (executionStats):
```
"executionStats": {
    "executionSuccess": true,
    "nReturned": 1,
    "executionTimeMillis": 18,
    "totalKeysExamined": 0,
    "totalDocsExamined": 246,
    "executionStages": {
        "stage": "COLLSCAN",
        "filter": {
            "_id.name": {
                "Seq": "Mexico"
            }
        }
    }
}
```

In this case execution time taken is 18 mills that’s huge improvement from previous experiment.

6. CONCLUSION

In this paper, we have seen how the aggregate method has proven better then find method in terms of performance and in addition by tweaking the model to different structure resulted in the faster query execution hence aggregate method is useful for retrieving interesting data not only from the aggregated structure but also from the simple linear structure.

7. ACKNOWLEDGMENTS

I would like to express my special thanks to my mentor Prof. Meenakshi Garg and my classmates who helped me gather the related information.

Also, I would like to thank my parents who provided me with resources that are needed for this research.

8. REFERENCES


