Show Submission Credentials

P3. MongoDB Primer Primer for MongoDB.

44 days 1 hours left

✔ Introduction to MongoDB

Introduction to MongoDB

MongoDB Introduction

MongoDB is an open-source document store. It is built on an architecture of collections and documents, let's define them in the context of MongoDB. A document is the basic unit of data in MongoDB. It consists of sets of key-value pairs. The documents are stored in collections. MongoDB is designed with high availability and scalability, along with out-of-the-box replication and auto-sharding. Many concepts in MongoDB have close analogs compared to MySQL. The table below summarizes the analogous terms in MySQL and MongoDB. Note that these terms are not equivalent, but rather share some similarities.

MySQL	**MongoDB**
Table	Collection
Row	Document
Column	Field
Joins	Embedded documents, linking

Features of MongoDB

Dynamic Schemas

You can create records without first defining the structure or schema. You can change the structure of the documents by adding new fields or removing existing fields.

Indexing

Similar to MySQL, MongoDB supports B-Tree indexes (https://docs.mongodb.com/manual/indexes/) to perform efficient execution of queries. Without a proper index, MongoDB must perform a full-collection scan to execute a query. If an index exists and matches a query, the number of documents that must be scanned will be significantly reduced. You can tell that this mechanism is similar to what we explored with indexing in MySQL.

Replication and Eventual Consistency

Replicating data in distributed data centers can increase data locality and availability. MongoDB provides data replication with replica sets (https://docs.mongodb.com/manual/replication/). Each copy of the data is called a replica, which may serve as a primary or secondary node. **All writes are done on the primary node** and the secondary nodes maintain a copy of the data asynchronously using **eventual consistency**. When the primary node fails, **an election process** will occur to determine which secondary node will serve as the new primary one. By default, clients only read from the primary as well. However, secondary nodes can optionally serve read operations, but note that the returned data can be stale as we discussed in the CAP Theorem in the NoSQL primer.

Sharding and Load Balancing

MongoDB scales horizontally using sharding (http://docs.mongodb.org/manual/sharding/). The user chooses a shard key to determine how the data in a collection will be distributed. MongoDB supports two sharding strategies. Ranged sharding can speed up range queries, and hashed sharding can achieve more even data distribution.

With sharding, a MongoDB query router (mongos (https://docs.mongodb.com/manual/reference/program/mongos/)) can load balance the query to the right shard and thus improve the performance of the whole cluster.

Aggregation

Aggregation operations in a database returns a single computed result, by processing data records grouped on a certain condition.

MongoDB provides three approaches to perform aggregation: the aggregation pipeline, the map-reduce function, and single purpose aggregation methods.

1. Aggregation Pipeline

In MongoDB, a multi-stage pipeline can be used to transform the original records into an aggregated result.

In the following diagram, we have a collection called orders, and we want to find the sum of the order amount for each customer from all orders with status "A" (Accepted). Note that this query has been split into two pipeline stages, $match and $group.

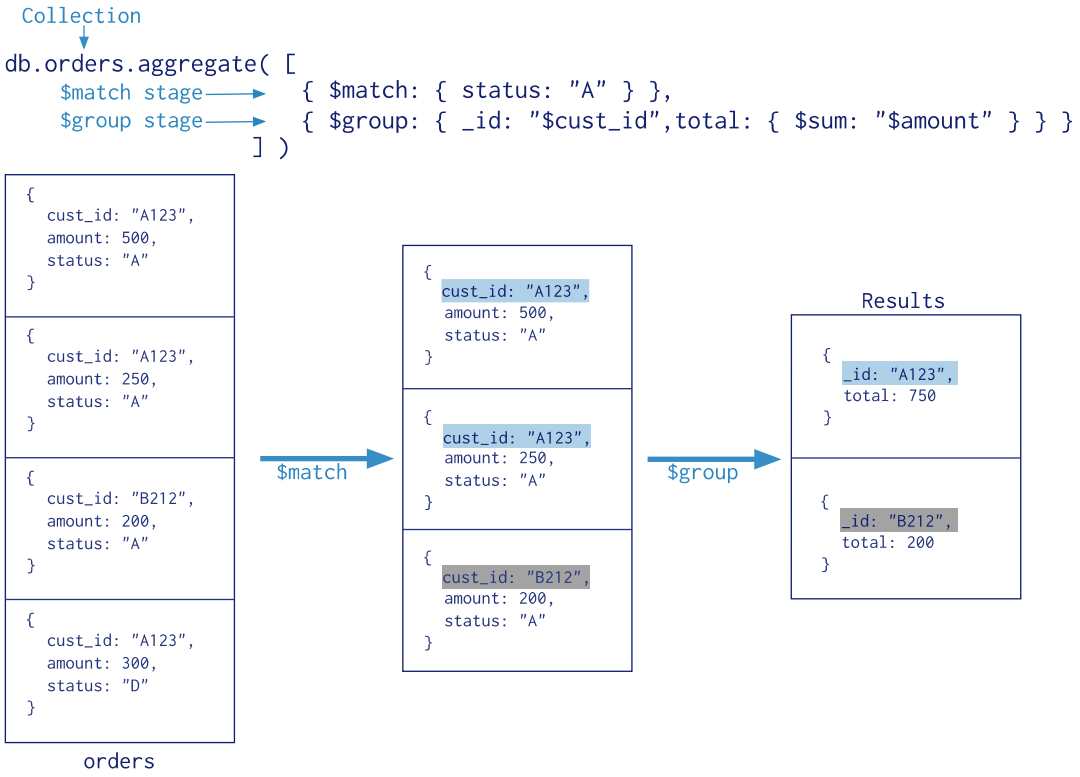

***Figure 1**: MongoDB aggregation with two stages (Reference: https://docs.mongodb.com/manual/core/aggregation-pipeline/#aggregation-pipeline-operators-and-performance)*

There are also other pipeline operators available for different aggregation operations. You can learn more about these operators here (https://docs.mongodb.com/manual/core/aggregation-pipeline/#aggregation-pipeline-operators-and-performance).

The use of pipelines is very common in the data processing field. Imagine you are performing the query above in shell script and pandas, how would you approach building this query? Despite differences in tools, the pipeline approach is widely accepted.

Last but not least, indexing can improve the performance of some aggregation operations.

2. Map-Reduce Function

MongoDB provides support for map-reduce operations. As you may recall from Hadoop MapReduce, map-reduce typically has two phases: map phase that processes entries and emits a key-value pair for each, and the reduce phase that aggregates the output of the map phase. In MongoDB, there is also an optional `finalize` phase, in which modifications can be made to the output of the reduce phase.

An example of a map-reduce operation is shown below. In this example, we are aggregating the amount of orders from every customer, and then query for the ones that have status “A”(Accepted). In the end, the result will be written to a new collection named `order_totals`. MongoDB will first filter the input documents based on the query condition, and then perform map-reduce. In the end, it will write the result to the collection specified.

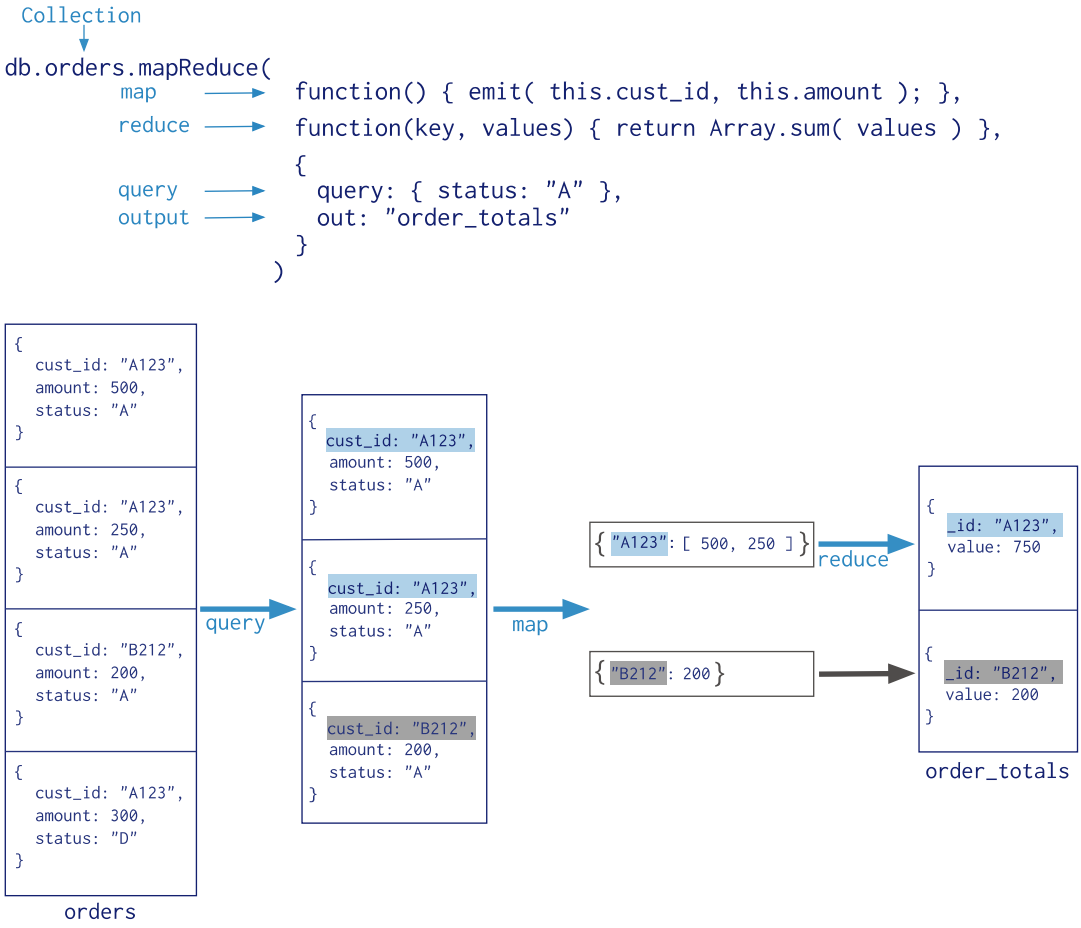

***Figure 2**: a map-reduce operation in MongoDB (Reference: https://docs.mongodb.com/manual/aggregation/#map-reduce)*

Despite the similar model map-reduce operations share between MongoDB and Hadoop, do note that map-reduce in MongoDB is not designed to process a large-scale data set. Specifically, MongoDB does not provide the functionality of distributing map-reduce operations to multiple nodes in a cluster. In contrast, Hadoop MapReduce is designed to process large-scale data sets, and is capable of distributing map-reduce jobs to multiple nodes in a hadoop cluster, which expedites the processing speed on a large-scale data set.

3. Single Purpose Aggregation Methods

There are several built-in aggregation methods to provide straightforward semantics for some of the most common data processing operations. These methods are quick and simple, but lack the flexibility and capabilities of the aggregation pipeline and map-reduce.

The following two commands are examples of single purpose aggregation methods. The first will return the number of total record in a collection, and the second will return the distinct documents in a collection.

```
db.collection.count(query, options)
db.collection.distinct(field, query, options)
```

Technicalities

Documents

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects which consist of key-value pairs. The values of fields may include other documents, arrays, and arrays of documents.

```
{
  "_id" : ObjectId("5f6ca64021ab3a0a36f22a66"),
  "address" : {
          "building" : "1007",
          "coord" : [
                  -73.856077,
                  40.848447
          ],
          "street" : "Morris Park Ave",
          "zipcode" : "10462"
  },
  "borough" : "Bronx",
  "cuisine" : "Bakery",
  "grades" : [
          {
                  "date" : ISODate("2014-03-03T00:00:00Z"),
                  "grade" : "A",
                  "score" : 2
          },
          {
                  "date" : ISODate("2013-09-11T00:00:00Z"),
                  "grade" : "A",
                  "score" : 6
          },
          {
                  "date" : ISODate("2013-01-24T00:00:00Z"),
                  "grade" : "A",
                  "score" : 10
          },
          {
                  "date" : ISODate("2011-11-23T00:00:00Z"),
                  "grade" : "A",
                  "score" : 9
          },
          {
                  "date" : ISODate("2011-03-10T00:00:00Z"),
                  "grade" : "B",
                  "score" : 14
          }
  ],
  "name" : "Morris Park Bake Shop",
  "restaurant_id" : "30075445"
}
```

Collections

MongoDB stores documents in collections. Collections are analogous to tables in relational databases. Unlike a table, however, a collection **does not** require its documents to have the same schema.

In MongoDB, documents stored in a collection must have a unique `_id` field that acts as a primary key (https://docs.mongodb.com/manual/reference/glossary/#term-primary-key). Furthermore, the `_id` field is reserved only for primary keys, which should be a unique value. If you don't set `_id` to be some value, MongoDB will automatically fill the field with a "MongoDB Id Object" - but you can put any unique information into that field if you wish. Visit Mongo's Documents Intro (https://docs.mongodb.com/manual/core/document/#OptimizingObjectIDs-Usethecollections%27naturalprimarykey%27intheidfield) page for more information on how Documents function in Mongo.

Provision a GCP instance using Terraform

Although you are allowed to use the Web UI to provision your resources in this primer, we strongly recommend that you use Terraform. We provide you with this prepared template that enables you to provision the required resources for this primer.

Download and unzip the terraform template with the following command:

```
wget https://clouddeveloper.blob.core.windows.net/assets/cloud-storage/primers/mongodb-terraform.tgz
tar -xvzf mongodb-terraform.tgz
```

1. Initialize Cloud SDK: `gcloud init`
2. You may create a new GCP project or use an existing project for this primer. If you decide to use an existing one, skip this step.

   ```
   gcloud projects create --name gcp-mongodb-primer
   ```

3. **Enable API for the primer**: Compute Engine API (https://console.cloud.google.com/flows/enableapi?apiid=compute)
4. Configure gcloud and set the default region, zone, and project. Replace `<Project ID>` with your unique project-id.

   ```
   gcloud config set project <Project ID>
   gcloud config set compute/region us-east1
   gcloud config set compute/zone us-east1-b
   ```

5. In the Terraform, replace the `<Project-Id>` in `main.tf` with your unique project-id. Replace the `<Tag>` in `main.tf` with the tag value being used in the **currently ongoing project**.
6. Launch Student Instance with Terraform

   ```
   terraform init
   terraform apply
   ```

After the creation, you can SSH into the VM:

```
gcloud compute --project <Project ID> ssh --zone us-east1-b clouduser@gcp-student-instance
```

Installation

For installing Mongo, we recommend that you see MongoDB's installation guides (https://docs.mongodb.com/manual/installation/). Install MongoDB Community edition for Ubuntu 18.04 (Bionic)

Documentation

For all documentation concerning MongoDB and its features, please refer to MongoDB's manual (https://docs.mongodb.com/manual/).

Tutorial

Now let's get started with some hands-on practice!

Step 1: Import data into mongoDB

Note that `root` privilege is required.

- Download the data with the following command: `wget https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/primer-dataset.json`
- Create your db path: `sudo mkdir -p /data/db && sudo chown -R clouduser /data/db`
- Start your mongod server: `sudo mongod`
- Start another terminal session and import the data: `mongoimport --db mongo_primer --collection restaurants --drop --file <PATH_TO_DOWNLOADED_JSON>`
- After you have successfully imported the restaurants collection, you will see logs as follows: `2018-02-24T16:01:48.567-0800 imported 25359 documents`

Step 2: Basic Queries with mongo Shell After we have imported data, we can perform basic queries using mongo shell.

- Connect to your mongod server: To connect to the mongod server using mongo shell, simply type the following command in your terminal: `mongo`
- We then need to choose the database that contains our restaurants collection: `use mongo_primer`
- Retrieve a restaurant document: To check the fields inside a restaurant document, simply use the following command: `db.restaurants.findOne()`

You will then retrieve a document shown below:

```
{
  "_id" : ObjectId("5a91fceb436c0d52e4aa03fb"),
  "address" : {
    "building" : "469",
    "coord" : [
      -73.961704,
      40.662942
    ],
    "street" : "Flatbush Avenue",
    "zipcode" : "11225"
  },
  "borough" : "Brooklyn",
  "cuisine" : "Hamburgers",
  "grades" : [
    {
      "date" : ISODate("2014-12-30T00:00:00Z"),
      "grade" : "A",
      "score" : 8
    },
    {
      "date" : ISODate("2014-07-01T00:00:00Z"),
      "grade" : "B",
      "score" : 23
    },
    {
      "date" : ISODate("2013-04-30T00:00:00Z"),
      "grade" : "A",
      "score" : 12
    },
    {
      "date" : ISODate("2012-05-08T00:00:00Z"),
      "grade" : "A",
      "score" : 12
    }
  ],
  "name" : "Wendy'S",
  "restaurant_id" : "30112340"
}
```


Query for documents in the restaurants collection:

MongoDB supports a multi-level query in its query language. The user can query by a top-level field, a field in an embedded document, and even a field in an array. For instance, if the user wants to query for all restaurants on `Flatbush Avenue`, the query will look like below:
`db.restaurants.find({"address.street" : "Flatbush Avenue"})`

To query for restaurants that contain grade “A” in its grades array field, simply use the following command: `db.restaurants.find({"grades.grade" : "A"})`

It is also possible to query for documents based on conditions. MongoDB provides comparison operators (https://docs.mongodb.com/manual/reference/operator/query-comparison/) to facilitate this functionality. Say we want to find restaurants whose grades contain a score higher than 50, the query will look like the following:
`db.restaurants.find({"grades.score" : { $gt: 50}})`

Moreover, we can combine conditions. To query for documents that satisfy multiple conditions, simply use a query with all these conditions. For instance:
`db.restaurants.find({"grades.score" : { $gt: 50}, "borough": "Manhattan"})`

This will query for documents with `Manhattan` in their `borough` field, and contain a score higher than 50 in the grades array.

You can also take advantage of `$or` operator to query for documents that satisfy any of the conditions listed. `db.restaurants.find({$or: [{"grades.score" : {$gt: 50}}, {"borough": "Manhattan"}]})` This query will return documents that match any of these two conditions.

To limit the number of documents to return, simply add `.limit(<number_of_documents_expected>)` at the end of your query.

To practice your query skills, try to come up with a query that finds 10 restaurants on `Flatbush Avenue` with a score above 30 in all of its grades.

The query result will look like the following:


```
    { "_id" : ObjectId("5a91fceb436c0d52e4aa04af"), "address" : { "building" :
"2301", "coord" : [ -73.9270926, 40.6142428 ], "street" : "Flatbush Avenue", "zi
pcode" : "11234" }, "borough" : "Brooklyn", "cuisine" : "American", "grades" : [
{ "date" : ISODate("2014-02-20T00:00:00Z"), "grade" : "A", "score" : 10 }, { "da
te" : ISODate("2013-07-01T00:00:00Z"), "grade" : "B", "score" : 15 }, { "date" :
ISODate("2013-01-10T00:00:00Z"), "grade" : "B", "score" : 15 }, { "date" : ISODa
te("2012-07-23T00:00:00Z"), "grade" : "A", "score" : 11 }, { "date" : ISODate("2
012-01-04T00:00:00Z"), "grade" : "A", "score" : 7 }, { "date" : ISODate("2011-09
-19T00:00:00Z"), "grade" : "C", "score" : 34 }, { "date" : ISODate("2011-04-11T0
0:00:00Z"), "grade" : "B", "score" : 16 } ], "name" : "New Floridian Diner", "re
staurant_id" : "40367164" }
    { "_id" : ObjectId("5a91fceb436c0d52e4aa0a5d"), "address" : { "building" :
"318", "coord" : [ -73.9720554, 40.6765935 ], "street" : "Flatbush Avenue", "zip
code" : "11238" }, "borough" : "Brooklyn", "cuisine" : "Pizza", "grades" : [ {
"date" : ISODate("2014-11-20T00:00:00Z"), "grade" : "A", "score" : 11 }, { "dat
e" : ISODate("2013-12-05T00:00:00Z"), "grade" : "A", "score" : 11 }, { "date" :
ISODate("2013-07-08T00:00:00Z"), "grade" : "A", "score" : 10 }, { "date" : ISODa
te("2013-01-04T00:00:00Z"), "grade" : "A", "score" : 13 }, { "date" : ISODate("2
012-05-24T00:00:00Z"), "grade" : "A", "score" : 11 }, { "date" : ISODate("2012-0
2-06T00:00:00Z"), "grade" : "C", "score" : 32 } ], "name" : "Antonio'S Pizza",
"restaurant_id" : "40535659" }
    { "_id" : ObjectId("5a91fceb436c0d52e4aa0add"), "address" : { "building" :
"218", "coord" : [ -73.9754742, 40.6809974 ], "street" : "Flatbush Avenue", "zip
code" : "11217" }, "borough" : "Brooklyn", "cuisine" : "Pizza", "grades" : [ {
"date" : ISODate("2014-09-30T00:00:00Z"), "grade" : "A", "score" : 2 }, { "date"
: ISODate("2013-10-08T00:00:00Z"), "grade" : "A", "score" : 11 }, { "date" : ISO
Date("2013-05-21T00:00:00Z"), "grade" : "A", "score" : 9 }, { "date" : ISODate
("2012-09-01T00:00:00Z"), "grade" : "A", "score" : 8 }, { "date" : ISODate("2012
-03-31T00:00:00Z"), "grade" : "C", "score" : 31 } ], "name" : "Gino'S Pizza", "r
estaurant_id" : "40551093" }
    ...
```


Step 3: Build index to speed up your query

Just like traditional RDBMS, the user can build index on certain field to speed up queries using that field as a condition. For the restaurants collection, say we frequently query based on the field `borough`, we may want to build index on this field. `db.restaurants.createIndex( { "borough": 1 } )`

You will see the following response:

```
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

You might wonder why there are 2 indexes now. This is because mongodb automatically indexes on the `_id` field.

Now, we can check how fast is a query based on `borough` with this index. Let's test this using the following query: `db.restaurants.find( { "borough": "San Francsico" }).explain("executionStats");`

You will see the following info in the console:

```
  ...
  executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 0,
      "executionTimeMillis" : 6,
      "totalKeysExamined" : 0,
      "totalDocsExamined" : 0,
  ...
```

Let's check the query execution time without an index: `db.restaurants.dropIndex( { "borough": 1 } )` `db.restaurants.find( { "borough": "San Francsico" }).explain("executionStats");`

```
  ...
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 0,
    "executionTimeMillis" : 16,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 25359,
  ...
```

Wow, the query with index was 10 ms faster! We were also able to skip examining 25359 docs with an index!

That's it for the tutorial! We encourage you to explore and learn more. With the knowledge you learned in this tutorial, you should be able to cope with the challenges in later projects.

To explore Mongo's Java API offering and other open source libraries, refer to their GitHub page docs here (http://mongodb.github.io/mongo-java-driver/3.6/) and Mongo's GitHub org (https://github.com/mongodb) for tons of cool tools and projects to explore.

Other NoSQL Solutions

To explore more NoSQL alternatives, http://nosql-database.org (http://nosql-database.org) has tons of information on all variations of NoSQL data stores.

Additionally, GitHub has a NoSQL Showcase (https://github.com/showcases/nosql-databases) that is incredibly relevant regarding real-time open-source efforts and the happenings therein.

Delete Cloud Resources

If you created a new project, delete the GCP project:

```
$ gcloud projects delete gcp-mongodb-primer
```

If you use an existing project, make sure to terminate only resources that you have created in this primer.