

Database Systems

SOEN 363 - Winter 2020

Team GO

Project - Phase 2

Submitted to:

Dr. Essam Mansour

By:

Razine Bensari - 40029076

Marwan Ayadi - 40039895

Israt Kazi Noor - 40029299

Mohamed Farah - 27552254

Out: March 16, 2020

Due: April 15, 2020

Introduction

The aim of this report is to provide functioning queries from a NOSQL database and to explain how the dataset was constructed, loaded and queried. We used a document database MongoDB to store Github's dataset of the top 10 most starred projects. The size of the dataset is around 4.8 gb. The MongoDB dump is around 23.77 gb but once it is loaded into MongoDB and the database is initialized, the shown size of the running database is around 4.8 gb.

Toolings & Libraries

- NoSQL Booster (GUI) - **5.2.10**
- Github api: <https://api.github.com>
- Front end: ReactJs (phase 2) - if applicable
- Server side: Go (phase 2) - if applicable

Analyzing Big Data Using NoSQL Systems

Download a big real dataset; it is recommended to get a dataset of at least 0.5 GBs.

Similar to phase 1, the dataset is the results of github mining from the top 10 starred projects from 2014.

The downloaded dataset is a 2.7 gb compressed zip file. When uncompressed, it is about 23.77 gb. It contains a .JSON file for each collection and also a .BSON file for each collection. Both can be used for the NoSQL database. It also contains a system index .JSON file where custom indexes mechanisms are specified.

When the .JSON files are loaded into the database, the database memory usage is around 4.8 gb.

Provide the data model for your datasets, i.e., graph, document, key-value, or column-store.

Each collection in the database represents an entity that is persisted. We have a total of 16 collections. We will show the schema for the most important ones but not all collections since they are not all relevant for the purpose of the queries.

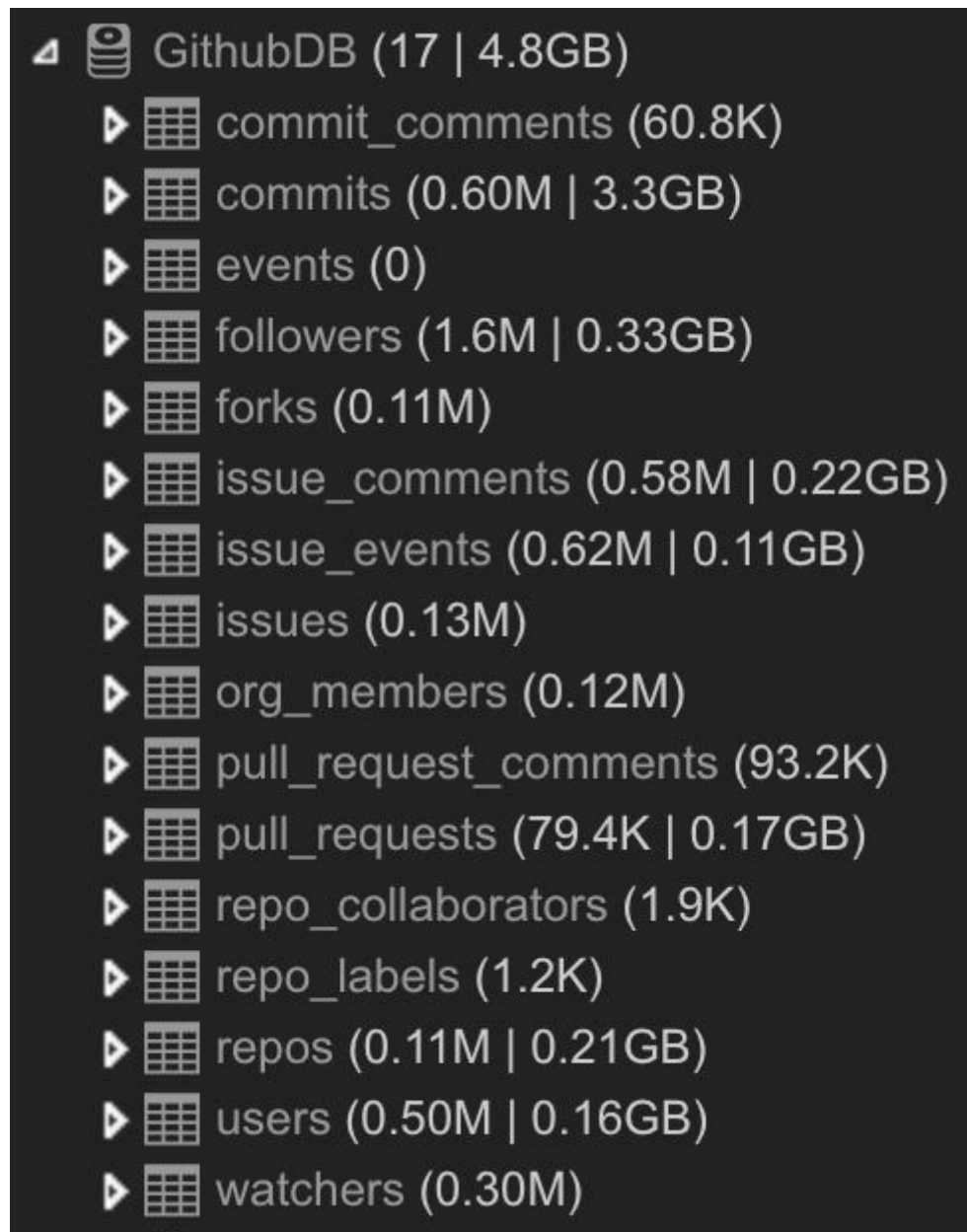


Figure 1: a list of all the collections in the document database

The following 2 figures are a JSON format of the data model of the users collection and issues collection.

```
/* 1 createdAt:9/14/2013, 6:45:02 AM*/
{
  "_id" : ObjectId("52343e2ebd3543bb7f000001"),
  "login" : "akka",
  "id" : 1496237,
  "avatar_url" : "https://2.gravatar.com/avatar/0ff72320fb1bc5db35ba4fae61614eb0?d=https%3A%2F%2Fidenticons",
  "gravatar_id" : "0ff72320fb1bc5db35ba4fae61614eb0",
  "url" : "https://api.github.com/users/akka",
  "html_url" : "https://github.com/akka",
  "followers_url" : "https://api.github.com/users/akka/followers",
  "following_url" : "https://api.github.com/users/akka/following{/other_user}",
  "gists_url" : "https://api.github.com/users/akka/gists{/gist_id}",
  "starred_url" : "https://api.github.com/users/akka/starred{/owner}/{/repo}",
  "subscriptions_url" : "https://api.github.com/users/akka/subscriptions",
  "organizations_url" : "https://api.github.com/users/akka/orgs",
  "repos_url" : "https://api.github.com/users/akka/repos",
  "events_url" : "https://api.github.com/users/akka/events{/privacy}",
  "received_events_url" : "https://api.github.com/users/akka/received_events",
  "type" : "Organization",
  "name" : "Akka Project",
  "company" : null,
  "blog" : "http://akka.io",
  "location" : "Uppsala, Sweden",
  "email" : "info@typesafe.com",
  "hireable" : null,
  "bio" : null,
  "public_repos" : 5,
  "followers" : 0,
  "following" : 0,
  "created_at" : "2012-03-03T11:44:45Z",
  "updated_at" : "2013-09-12T20:18:38Z",
  "public_gists" : 0
},
```

Figure 2: data model of the user collection for user “akka”

```
{
  "_id" : ObjectId("52343f3dbd35436de30000c6"),
  "url" : "https://api.github.com/repos/mavam/stat-cookbook/issues/2",
  "labels_url" : "https://api.github.com/repos/mavam/stat-cookbook/issues/2/labels{/name}",
  "comments_url" : "https://api.github.com/repos/mavam/stat-cookbook/issues/2/comments",
  "events_url" : "https://api.github.com/repos/mavam/stat-cookbook/issues/2/events",
  "html_url" : "https://github.com/mavam/stat-cookbook/pull/2",
  "id" : 7474201,
  "number" : 2,
  "title" : "Typo in balls and urns",
  "user" : {
    "login" : "mathbruyen",
    "id" : 449671,
    "avatar_url" : "https://1.gravatar.com/avatar/0fb99ea9d639e8224a226ab1f535df24?d=https%3A%2F%2Fidenticons",
    "gravatar_id" : "0fb99ea9d639e8224a226ab1f535df24",
    "url" : "https://api.github.com/users/mathbruyen",
    "html_url" : "https://github.com/mathbruyen",
    "followers_url" : "https://api.github.com/users/mathbruyen/followers",
    "following_url" : "https://api.github.com/users/mathbruyen/following{/other_user}",
    "gists_url" : "https://api.github.com/users/mathbruyen/gists{/gist_id}",
    "starred_url" : "https://api.github.com/users/mathbruyen/starred{/owner}/{/repo}",
    "subscriptions_url" : "https://api.github.com/users/mathbruyen/subscriptions",
    "organizations_url" : "https://api.github.com/users/mathbruyen/orgs",
    "repos_url" : "https://api.github.com/users/mathbruyen/repos",
    "events_url" : "https://api.github.com/users/mathbruyen/events{/privacy}",
    "received_events_url" : "https://api.github.com/users/mathbruyen/received_events",
    "type" : "User"
  },
  "labels" : [ ],
  "state" : "closed",
  "assignee" : null,
  "milestone" : null,
  "comments" : 0,
  "created_at" : "2012-10-10T07:53:00Z",
  "updated_at" : "2012-10-10T15:31:52Z",
  "closed_at" : "2012-10-10T15:31:52Z",
  "pull_request" : {
    "html_url" : "https://github.com/mavam/stat-cookbook/pull/2",
    "diff_url" : "https://github.com/mavam/stat-cookbook/pull/2.diff",
    "patch_url" : "https://github.com/mavam/stat-cookbook/pull/2.patch"
  },
  "body" : "Two rows in the table for combinatorics - balls and urns used to have both B distinguishable",
  "repo" : "stat-cookbook",
  "owner" : "mavam"
},
```

Figure 3: schema of the issue collection

The rest of the data model for each collections can be found in this link: [MongoDB schema](#)

Collections in MongoDB

Here is a list of collections along with the Github API URL they cache data from. All URLs need to be prefixed with <https://api.github.com/>. In MongoDB, each entity is by default indexed by the parameter fields in each corresponding URL (see also the actual [default indexes](#)).

Collection name	Github API URL	Documentation URL
commit_comments	<code>#(user)/#{repo}/commits/#{sha}/comments</code>	commit comments
commits	<code>repos/#{user}/#{repo}/commits</code>	commits
events	<code>events</code>	events
followers	<code>users/#{user}/followers</code>	followers list
forks	<code>repos/#{user}/#{repo}/forks</code>	forks list
issues	<code>/repos/#{owner}/#{repo}/issues</code>	issues for a repo
issue_comments	<code>repos/#{owner}/#{repo}/issues/comments/#{comment_id}</code>	issue comments
issue_events	<code>repos/#{owner}/#{repo}/issues/events/#{event_id}</code>	issue events
org_members	<code>orgs/#{org}/members</code>	organization members
pull_request_comments	<code>repos/#{owner}/#{repo}/pulls/#{pullreq_id}/comments</code>	pull request review comments
pull_requests	<code>repos/#{user}/#{repo}/pulls</code>	pull requests
repo_collaborators	<code>repos/#{user}/#{repo}/collaborators</code>	repo collaborators
repo_labels	<code>repos/#{owner}/#{repo}/issues/#{issue_id}/labels</code>	issue labels
repos	<code>repos/#{user}/#{repo}</code>	repositories
users	<code>users/#{user}</code>	users
watchers	<code>repos/#{user}/#{repo}/stargazers</code>	stargazers

Figure 4: endpoints for data mining and schema

In figure 4 we can see the endpoints used for data mining to generate the datasets. Each requests was then parsed to extract relevant data to further the mining process and persisted into MongoDB.

Here is a sample of what returned response from the github api looks like

Response

```
Status: 200 OK
Link: <https://api.github.com/resource?page=2>; rel="next",
      <https://api.github.com/resource?page=5>; rel="last"

[
  {
    "login": "octocat",
    "id": 1,
    "node_id": "MDQ6VXNlcjE=",
    "avatar_url": "https://github.com/images/error/octocat_happy.gif",
    "gravatar_id": "",
    "url": "https://api.github.com/users/octocat",
    "html_url": "https://github.com/octocat",
    "followers_url": "https://api.github.com/users/octocat/followers",
    "following_url": "https://api.github.com/users/octocat/following{/other_user}",
    "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/octocat/starred{/owner}/{/repo}",
    "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
    "organizations_url": "https://api.github.com/users/octocat/orgs",
    "repos_url": "https://api.github.com/users/octocat/repos",
    "events_url": "https://api.github.com/users/octocat/events{/privacy}",
    "received_events_url": "https://api.github.com/users/octocat/received_events",
    "type": "User",
    "site_admin": false
  }
]
```

Create a NoSQL database for a real dataset of your choice.

The creation of the NoSQL database, in our case a document based database (MongoDB) is a fairly simple process.

- Download MongoDB into the computer
- Start a MongoDB shell
- Type the following into the shell: use <DATABASE NAME>

After the command has been executed, the database has been created.

Load the dataset into your NoSQL system.

To load the dataset into the newly created database, you need to do the following.

- Run this command: **mongorestore -d <DATABASE NAME> /path/to/dataset**

Queries

#1

Get the top 10 most popular programming languages out of all the repos present in the database

```
db.repos.aggregate([
  {
    $group: {
      _id: "$language",
      Frequency: {$sum: 1}
    },
  },
  {
    $sort: {Frequency:-1}
  },
  {
    $limit: 10
  }
])
```

Key	Value	Type
▶ (1) JavaScript	{ _id : "JavaScript", Frequency : 29526 }	Document
▶ (2) Ruby	{ _id : "Ruby", Frequency : 26732 }	Document
▶ (3) PHP	{ _id : "PHP", Frequency : 12506 }	Document
▶ (4) Python	{ _id : "Python", Frequency : 10940 }	Document
▶ (5) Java	{ _id : "Java", Frequency : 9211 }	Document
▶ (6) C++	{ _id : "C++", Frequency : 6034 }	Document
▶ (7) C	{ _id : "C", Frequency : 5235 }	Document
▶ (8) C#	{ _id : "C#", Frequency : 4396 }	Document
▶ (9) Scala	{ _id : "Scala", Frequency : 2091 }	Document
▶ (10) CSS	{ _id : "CSS", Frequency : 548 }	Document

#2

Get the top 10 repos with the most issues

```
db.issues.aggregate()  
  .group({  
    _id: "$repo",  
    nb_issues:{$sum:1}  
  })  
  .sort({nb_issues:-1})  
  .limit(10)  
  .project({  
    _id:0,  
    repo_name: "$_id",  
    Number_of_issues:"$nb_issues"  
  })
```

	repo_name	Number_of_issues
1	homebrew	22,667 (22.7K)
2	rails	12,296 (12.3K)
3	symfony	9,085 (9.1K)
4	gitlabhq	5,109 (5.1K)
5	diaspora	4,538 (4.5K)
6	three.js	3,870 (3.9K)
7	elasticsearch	3,697 (3.7K)
8	xbmc	3,270 (3.3K)
9	foundation	3,262 (3.3K)
10	bitcoin	2,993 (3.0K)

#3

Get the top 10 repos with the most pull requests

```
db.pull_requests.aggregate()  
  .group({  
    _id: "$repo",  
    nb_pull_requests: {$sum:1}  
  })  
  .sort({nb_pull_requests:-1})  
  .limit(10)  
  .project({  
    _id:0,  
    repo_name: "$_id",  
    Number_of_pull_requests:"$nb_pull_requests"  
  })
```

	repo_name	Number_of_pull_requests
1	homebrew	13,213 (13.2K)
2	rails	7,825 (7.8K)
3	symfony	5,874 (5.9K)
4	zf2	4,100 (4.1K)
5	xbmc	3,130 (3.1K)
6	scala	3,002 (3.0K)
7	node	2,310 (2.3K)
8	bitcoin	1,956 (2.0K)
9	django	1,707 (1.7K)
10	cakephp	1,704 (1.7K)

#4

Get the top 10 most forked repos and their owner's name

```

db.repos.aggregate()
  .sort({forks_count:-1})
  .limit(10)
  .lookup({
    from: "users",
    localField: "owner.id",
    foreignField: "id",
    as: "User"
  })
  .unwind("$User")
  .project({
    repo_name: "$name",
    Nb_Fork: "$forks_count",
    Owner_name: "$User.name"})

```

repos 8.935 s 10 Docs				
	_id	repo_name	Nb_Fork	Owner_name
1	ObjectId("5237b4c8bd3543")	homebrew	6,911 (6.9K)	Max Howell
2	ObjectId("5237b213bd3543")	rails	6,548 (6.5K)	Ruby on Rails
3	ObjectId("5235aab8bd3543")	html5-boilerplate	5,434 (5.4K)	H5BP
4	ObjectId("52358fd1bd3543")	jquery	4,920 (4.9K)	jQuery JavaScript Library
5	ObjectId("52357c15bd3543")	node	4,736 (4.7K)	Joyent
6	ObjectId("5235d4d3bd3543")	d3	3,497 (3.5K)	Mike Bostock
7	ObjectId("5235af7abd3543")	impress.js	3,408 (3.4K)	Bartek Szopka
8	ObjectId("52367248bd3543")	CodemIgniter	2,940 (2.9K)	EllisLab, Inc.
9	ObjectId("52361bf0bd3543")	three.js	2,834 (2.8K)	Mr.doob
10	ObjectId("52362382bd3543")	foundation	2,709 (2.7K)	zurb

#5

Repos with number of collaborators greater than 20

```
db.repo_collaborators.aggregate()
  .group({
    _id: "$repo",
    nb_collaborators: {$sum: 1}
  })
  .match({"nb_collaborators": {$gt: 20}})
  .sort({nb_collaborators: -1})
  .project({
    _id: 0,
    repo_name: "_id",
    Number_ofCollaborators: "$nb_collaborators"
  })
```

Key	Value	Type
▶ (1)	{ repo_name : "_id", Number_ofCollaborators : 313 }	Object
▶ (2)	{ repo_name : "_id", Number_ofCollaborators : 313 }	Object
▶ (3)	{ repo_name : "_id", Number_ofCollaborators : 313 }	Object
▶ (4)	{ repo_name : "_id", Number_ofCollaborators : 74 }	Object
▶ (5)	{ repo_name : "_id", Number_ofCollaborators : 70 }	Object
▶ (6)	{ repo_name : "_id", Number_ofCollaborators : 66 }	Object
▶ (7)	{ repo_name : "_id", Number_ofCollaborators : 48 }	Object
▶ (8)	{ repo_name : "_id", Number_ofCollaborators : 40 }	Object
▶ (9)	{ repo_name : "_id", Number_ofCollaborators : 39 }	Object
▶ (10)	{ repo_name : "_id", Number_ofCollaborators : 36 }	Object
▶ (11)	{ repo_name : "_id", Number_ofCollaborators : 33 }	Object
▶ (12)	{ repo_name : "_id", Number_ofCollaborators : 32 }	Object
▶ (13)	{ repo_name : "_id", Number_ofCollaborators : 30 }	Object
▶ (14)	{ repo_name : "_id", Number_ofCollaborators : 28 }	Object
▶ (15)	{ repo_name : "_id", Number_ofCollaborators : 26 }	Object
▶ (16)	{ repo_name : "_id", Number_ofCollaborators : 26 }	Object
▶ (17)	{ repo_name : "_id", Number_ofCollaborators : 25 }	Object

#6

List of Followers of each users

```
db.users.aggregate()
  .match({followers: {$gt: 0}})
  .lookup({
    from: "followers",
    localField: "login",
```

```

        foreignField: "follows",
        as: "List_of_Followers"
    })
    .project(
    {
        _id:0,
        "User": "$login",
        "List_of_Followers":1,
    })

```

Key	Value	Type
▲ (1)	{ 2 attributes }	Object
▶ List_of_Followers	Array[1019]	Array
User	hadley	String
▲ (2)	{ 2 attributes }	Object
▶ List_of_Followers	Array[429]	Array
User	johnmyleswhite	String
▲ (3)	{ 2 attributes }	Object
▶ List_of_Followers	Array[1019]	Array
User	hadley	String
▲ (4)	{ 2 attributes }	Object
▶ List_of_Followers	Array[651]	Array
User	yihui	String
▶ (5)	{ 2 attributes }	Object
▶ (6)	{ 2 attributes }	Object
▶ (7)	{ 2 attributes }	Object
▶ (8)	{ 2 attributes }	Object
▶ (9)	{ 2 attributes }	Object
▶ (10)	{ 2 attributes }	Object
▶ (11)	{ 2 attributes }	Object

#7

List of watchers of each repo

```

db.repos.aggregate()
    .match({watchers:{ $gt:0}})
    .lookup({
        from: "watchers",
        localField: "name",
        foreignField: "repo",
        as: "List_of_Watchers"
    })
    .project(
    {
        _id:0,
        "Repo": "$name",
        "List_of_Watchers":1,
    })

```

Key	Value	Type
▲ (1)	{ 2 attributes }	Object
▸ List_of_Watchers	Array[1941]	Array
Repo	akka	String
▲ (2)	{ 2 attributes }	Object
▸ List_of_Watchers	Array[677]	Array
Repo	devtools	String
▸ (3)	{ 2 attributes }	Object
▸ (4)	{ 2 attributes }	Object
▸ (5)	{ 2 attributes }	Object
▸ (6)	{ 2 attributes }	Object
▸ (7)	{ 2 attributes }	Object
▸ (8)	{ 2 attributes }	Object
▸ (9)	{ 2 attributes }	Object
▸ (10)	{ 2 attributes }	Object
▸ (11)	{ 2 attributes }	Object
▸ (12)	{ 2 attributes }	Object
▸ (13)	{ 2 attributes }	Object
▸ (14)	{ 2 attributes }	Object






#8

Average Number of Commits Per Day

```

db.commits.aggregate([
  { $group: {
    _id: {
      $add: [
        { $dayOfYear: "$commit.committer.date"},
        { $multiply:
          [400, {$year: "$commit.committer.date"}]
        }
      ]
    },
    NumberOfTimes: { $sum: 1 },
    f: {$min: "$commit.committer.date"}
  }
},
]).group({
  _id: null,
  nb_commits:{$sum:"$NumberOfTimes"},
  nb_days:{$sum: 1},
  avgCommits: { $avg: "$NumberOfTimes" }
})
.project({_id:0,
"Number of commits": "$nb_commits",
"Number of days": "$nb_days",
"Average Commits per Day":"$avgCommits"})

```

Key	Value 
 (1)	{ 3 attributes }
 Number of commits	601,080 (0.60M)
 Number of days	3,810 (3.8K)
 Average Commits per Day	157.764






#9

Average Number of Pull Requests Per Day

```

db.pull_requests.aggregate([
  { $group: {
    _id: {
      $add: [
        { $dayOfYear: "$created_at" },
        { $multiply:
          [400, { $year: "$created_at" }]
        }
      ]
    },
    NumberOfTimes: { $sum: 1 },
    f: { $min: "$created_at" }
  }
},
[]).group({
  _id: null,
  nb_pullerrequests: { $sum: "$NumberOfTimes" },
  nb_days: { $sum: 1 },
  avgPullRequest: { $avg: "$NumberOfTimes" }
})
.project({ _id: 0,
  "Number of Pull Requests": "$nb_pullerrequests",
  "Number of days": "$nb_days",
  "Average Pull Requests per Day": "$avgPullRequest" })

```

Key	Value 
 (1)	{ 3 attributes }
 Number of Pull Requests	79,359 (79.4K)
 Number of days	1,133 (1.1K)
 Average Pull Requests per Day	70.043


#10

Average Number of Issues Per Day

```

db.issues.aggregate([
  { $group: {
    _id: {
      $add: [
        { $dayOfYear: "$created_at"},
        { $multiply:
          [400, { $year: "$created_at"}]
        }
      ]},
    NumberOfTimes: { $sum: 1 },
    f: { $min: "$created_at" }
  }},
  { $group: {
    _id: null,
    nb_issues: { $sum: "$NumberOfTimes" },
    nb_days: { $sum: 1 },
    avgIssues: { $avg: "$NumberOfTimes" }
  }},
  { $project: {
    _id: 0,
    "Number of Issues": "$nb_issues",
    "Number of days": "$nb_days",
    "Average Issues per Day": "$avgIssues"
  }}
])

```

Key	Value 
  (1)	{ "Number of Issues" : 126308 }
 Number of Issues	126,308 (0.13M)
 Number of days	1,600 (1.6K)
 Average Issues per Day	78.943

Consistency & Availability

Consistency

Since we opted for MongoDB to complete our project, we noticed that MongoDB is strongly consistent by default - if you do a write and then do a read, assuming the write was successful you will always be able to read the result of the write you just performed. This is because MongoDB is a single-master system and all reads go to the primary by default.

If you optionally enable reading from the secondaries then MongoDB becomes eventually consistent where it's possible to read out-of-date results. MongoDB uses a readers-writer lock that allows concurrent reads access to a database but gives exclusive access to a single write operation. It allows clients to read documents inserted or modified before it commits these modifications to disk, regardless of write concern level or journaling configuration.

For systems with multiple concurrent readers and writers, MongoDB will allow clients to read the results of a write operation before the write operation returns. If the mongod terminates before the journal commits, even if a write returns successfully, queries may have read data that will not exist after the mongod restarts. In our case, since we did not have a distributed system, we couldn't explore the option where multiple concurrent readers and writers had access to a shared DB.

Critical Trade-offs between Consistency and Availability in MongoDB When Partitioned

Basically, whenever there is a partition and a crisis happens and MongoDB needs to decide what to do, it will choose Consistency over Availability. It will stop accepting writes to the system until it believes that it can safely complete those writes.

Availability

MongoDB gets high availability through Replica-Sets. A replica set gathers different MongoDB servers into a cluster with different types of members. A primary instance which receives all the write operations. It also receives all the read operations by default. One or multiple secondary nodes which own a copy of the data. These nodes can receive the read operations by modifying the read preference in the MongoDB driver. These secondary servers receive data modification in real time when the primary receives write operations.

As soon as the primary goes down or becomes unavailable else, then the secondaries will determine a new primary to become available again. There is a disadvantage to this: Every write that was performed by the old primary, but not synchronized to the secondaries will be rolled back and saved to a rollback-file, as soon as it reconnects to the set (the old primary is a secondary now). So in this case some consistency is sacrificed for the sake of availability.

Indexes & Performance

One of the strong suits of MongoDB is how fast it operates given a good indexing strategy. This NoSQL system offers a great range of indexing strategies. We can create **indexes to support our queries**, use **indexes to support the sorting of the query results**, or simply use **an indexing strategy that serves the purpose of fast read/write operations**.

In our dataset, the default indexing strategy is to use the URL of the entity that we request through the github API and then use the path of that URL and index it. For example, when we mine github API in the following link:

Collection name	Github API URL
commit_comments	<code>#{user}/#{repo}/commits/#{sha}/comments</code>

The github API **URL** becomes the indexed value. This allows to provide a symbolic unique key (index) for each document. This type of indexing is efficient for read/write operations. Whenever a user wants to obtain a specific resource or simply persist a new object. Since the URL itself is used to identify a unique resource (hence the name Unique/Uniform Resource Locator).

However, most use cases are not simple read/write. Sometimes, the business logic requires you to perform complex queries to your database. This is where a good indexing strategy come into place.

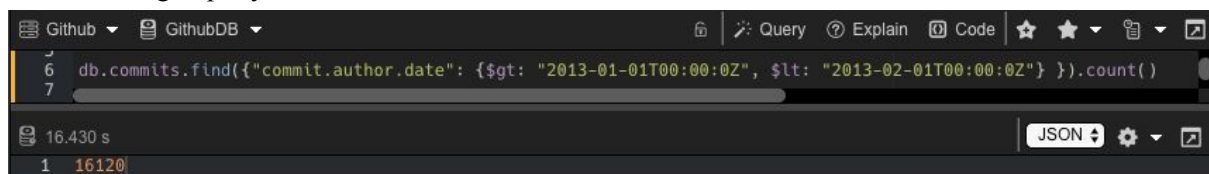
What about queries?

The efficiency of indexing is brought to life with the different indexing schemes. When it comes to querying your database (MongoDB), we are dealing with a list of collections. Each collection is a set of documents. Documents can be seen as your persisted entities, objects from the server side, data from the clients...

Consider the following example to appreciate the efficiency that indexes in MongoDB offer. Suppose you are working at Github and the year 2013 is coming to an end. Analytics team asks your developers to provide a report of the insights of the year 2013 (in our case we will use the github dataset at hand). You know that many of your queries will include a range to incorporate only the data flow that happened during the year 2013. Let's say you want to see how many commits have been made this year, how many pull requests, how many contributions from users or maintainers. All of these data needs to be filtered by date in order to satisfy the year constraint.

Let us consider the example of counting all commits made during the first day of the month of January (2013-01-01T00:00:0Z) until the first day of February (2013-02-01T00:00:0Z). Essentially we want to get all commits made during January.

The following a query to obtain such a count:

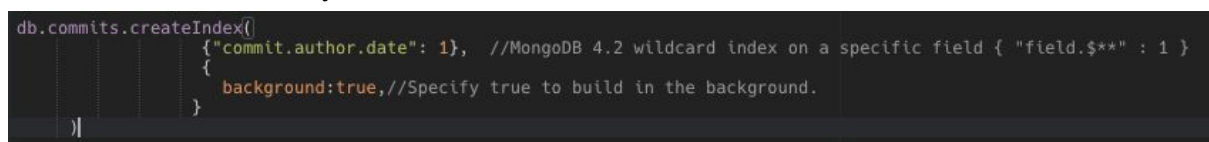


```

6 db.commits.find({'commit.author.date': {'$gt': '2013-01-01T00:00:0Z', '$lt': '2013-02-01T00:00:0Z'}}).count()
7
16.430 s
1 16120
  
```

Notice that it took 16.430 seconds to get a count of 16120 commits for the month of January.

We know that most of our queries to generate that report will use a date constraint. So let us index this field from the commit entity as follow:



```

db.commits.createIndex(
  {"commit.author.date": 1}, //MongoDB 4.2 wildcard index on a specific field { "field.$*" : 1 }
  {
    background:true,//Specify true to build in the background.
  }
)
  
```

Using this command to create an index

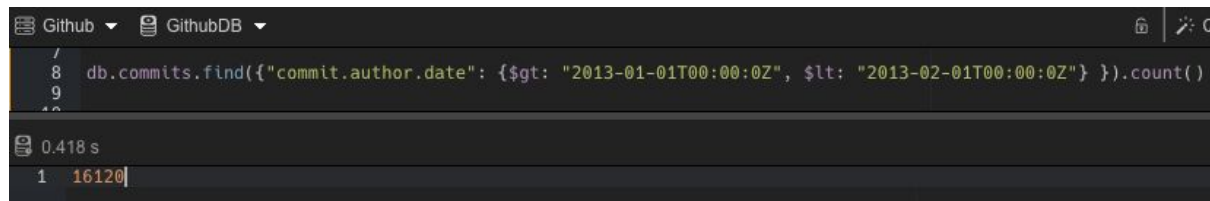
```

db.commits.createIndex(
    {"commit.author.date": 1}, //MongoDB 4.2 wildcard
index on a specific field { "field.$*" : 1 }
    {
        background:true,//Specify true to build in the
background.
    }
)
  
```

We can then run our query once again and see an improved execution time

```
db.commits.find({"commit.author.date": {$gt: "2013-01-01T00:00:0Z", $lt: "2013-02-01T00:00:0Z"} }).count()
```

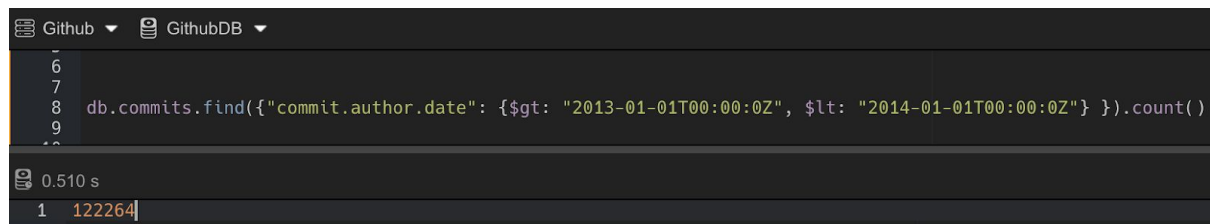
Here you can see how long it took



```
Github ▾ GithubDB ▾
8 db.commits.find({"commit.author.date": {$gt: "2013-01-01T00:00:0Z", $lt: "2013-02-01T00:00:0Z"} }).count()
9
0.418 s
1 16120
```

The execution time went from 16.430 seconds to 0.418 seconds. This corresponds to a 39 times faster execution time. Now, we can generate this report by making similar queries and indexing the “date” field in all the necessary documents with much less time.

Here is the execution time if we want the count of all commits made during the year 2013 in order for us to put it in the yearly report for the analytics team.



```
Github ▾ GithubDB ▾
6
7
8 db.commits.find({"commit.author.date": {$gt: "2013-01-01T00:00:0Z", $lt: "2014-01-01T00:00:0Z"} }).count()
9
0.510 s
1 122264
```

It took around 0.5 seconds to get the count of all commits made during the year of 2013 whereas before the indexing, the system had to go through all the fields of the commit document, throughout the whole collection, then it had to locate the attribute, then do a string comparison to verify if that commit is within the date range. By supplying an index, our execution time decreased greatly.

The indexing can also be applied to sorting the result set in a similar fashion as indexing a specific entity attribute as above.

The only thing to watch out for is to be smart when it comes to indexing. We need to know what part of our queries slow down the execution and how indexing accelerates the execution time. We also need to understand that indexing a single attribute may not be enough, sometimes compound indexes are necessary to completely benefit from the efficiency it brings.

Problem Encountered

The main issue encountered is that when mining github api for data about projects, repositories, commits, pull requests and many more, we had a limit of 5000 authenticated requests and 60 unauthenticated requests per hour. At this rate, it would be tedious and long to accumulate enough

variety and build a big dataset. This is why we also incorporated a MongoDB dump of data from 2014 that contains around 20 gb worth of data.

With this extra amount of data, our queries held much more insights and were closer to the actual state/representation of the github databases for repositories.