NoSQL Database Exploration with Github Events

**Team**:   
Junaid Abbasi  
Nikhil Mungel  
Vaishak Suresh

# Introduction

As part of this project, we explored two kinds of NoSQL database packages.

## MongoDB

MongoDB is a document based database that stores data in the form of JSON/BSON instead of traditional rows and columns. MongoDB offers a lot of flexibility for applications which naturally use data in the JSON format.

The key features in MongoDB that we used are:

* Simple Ad Hoc Queries
* Indexing
* Replication and Load Balancing
* Aggregation using MapReduce.

# Neo4j

Neo4j is an open source graph based persistent store that we used to create a recommendation engine for suggesting which users should other Github users follow.

## Part I -- MongoDB

# Implementation Details

## MongoDB

### Data

The data that we used was the events data from GitHub. Github is an online code repository that has a social aspect attached to it. Repositories on GitHub can be watched, forked, starred and followed by users. GitHub provides 18 such events. These are listed below:

* CommitCommentEvent
* CreateEvent
* DeleteEvent
* DownloadEvent
* FollowEvent
* ForkEvent
* ForkApplyEvent
* GistEvent
* GollumEvent
* IssueCommentEvent
* IssuesEvent
* MemberEvent
* PublicEvent
* PullRequestEvent
* PullRequestReviewCommentEvent
* PushEvent
* ReleaseEvent
* StatusEvent
* TeamAddEvent
* WatchEvent

We were interested in the PushEvent, WatchEvent and the IssueEvent.

GitHub does not let users download archived data and provides an API for accessing these events. The API itself is rate-limited and cannot be used to get a large number of events in the short duration of time we had for the project.

We used data from <http://www.githubarchive.org/> which has archived data in compressed JSON files by the hour. We downloaded the complete data for the year 2012. The data when uncompressed was nearly 55GB in size. This was too much data to handle on one laptop, so we decided to use 1 month’s data, which was nearly 5 GB in about 740 JSON files.

Each of these files had multiple types of events. Below is an example for push event.

{"type":"PushEvent","repo":{"id":3055800,"url":"https://api.github.dev/repos/knowledge-point/tinypm-backup","name":"knowledge-point/tinypm-backup"},"created\_at":"2012-01-01T00:00:09Z","payload":{"ref":"refs/heads/master","push\_id":55756268,"commits":[{"sha":"ad9010cbf0ecfd252c873ea7530342291f3e574b","author":{"email":"roman.apostol+backup@gmail.com","name":"Knowledge Point"},"url":"https://api.github.com/repos/knowledge-point/tinypm-backup/commits/ad9010cbf0ecfd252c873ea7530342291f3e574b","message":"backup at Sun Jan 1 00:00:02 UTC 2012"}],"head":"ad9010cbf0ecfd252c873ea7530342291f3e574b","size":1},"actor":{"login":"kp-backup","id":1287779,"url":"https://api.github.dev/users/kp-backup","avatar\_url":"https://secure.gravatar.com/avatar/b55e2cae26595d21039ad1bc05db5950?d=http://github.dev%2Fimages%2Fgravatars%2Fgravatar-user-420.png","gravatar\_id":"b55e2cae26595d21039ad1bc05db5950"},"public":true,"id":"1508512236"}

### Insertion

The data we have per event is in JSON format. Each event is unique and there can be no repetition since the events are captured by hour. We did not have a need to identify each record uniquely, since all we cared about was the rolled up statistics of the events. We used *db.collection.save()* method to insert the JSON event data into the DB. This creates a document. MongoDB has a restriction on the document size (12MB), but since the document we created was just one event, we did not have to use the **GridFS** of MongoDB.

The events that we downloaded in the form of JSON had some issues with encoding. They contained UTF-8 characters that were not encoded properly, this would cause problems when inserting into the database. We fixed most of this programmatically by encoding them when the exception happened. Things that could not be fixed were ignored since the occurrence was very rare.

Each individual event was unique in the dataset and had a natural key (the create\_date). We did not need this key specifically because we were rolling up the data from the entire collection. For this reason, we let MongoDB generate the “\_id” on its own.

### Querying

We used Bottle.py to provide a restful interface for querying the DB. We were interested in the following queries.

* Get all Push events
* Get all Watch events
* Get all Follow events
* Get all Issues events
* Get top repository
* Get top user
* Get repos with most following
* Get repos with most issues

Since we were using bottle.py, we used MongoPy library to query the database.

Below is a sample query, which gets the top repository with most commits.

db['push\_events']**.**aggregate([{"$group": {"\_id": "$repo.name", "count": {"$sum": 1}}},

         {"$sort": SON ([("count", **-**1), ("\_id", **-**1)])},

         {"$limit": limit}])

The query uses the aggregate function to get the repos that have the most commits. We initially used the ***group***function, but then realized that it has a limit of 10,000. We were not using a sharded database, so we used ***aggregate***function instead of ***map-reduce.***

The above query groups the push\_events collection by repo name and sums up the number of times the repo occurs. We are also sorting in descending order based on the count. The limit function limits the number of records fetched. This is a configurable parameter through the code.

### Sharding

We implemented sharding to understand the process of setting it up. In the end we did not use the shards since all the shards were running on the same laptop; they did not give us a significant performance improvement.

In the process of setting up the shards, we learnt the following things:

* *Mongos:* This is the controller for all the shards. All requests to the DB are handled by this and appropriately forwarded to the correct shad. The information is obtained from the config server.
* *Shards* These are the instances of *mongod*
* *ShardKey* The unique identifier that identifies unique records. We just used an autogenerated ID since the value of the key itself did not matter to us.
* *Config Server:* This is the central repository that holds the location of each shard.

Below are the observations:

The sharding doesn’t really help on the single machine because the seeking still competes for the same resource and seek is sequential. We did not get an opportunity to try this across different machines.

|  |  |  |
| --- | --- | --- |
| Operation | No Shard | 2 Shards |
| Insert | 474 | 350 |
| Select | 5 | 4.5 |

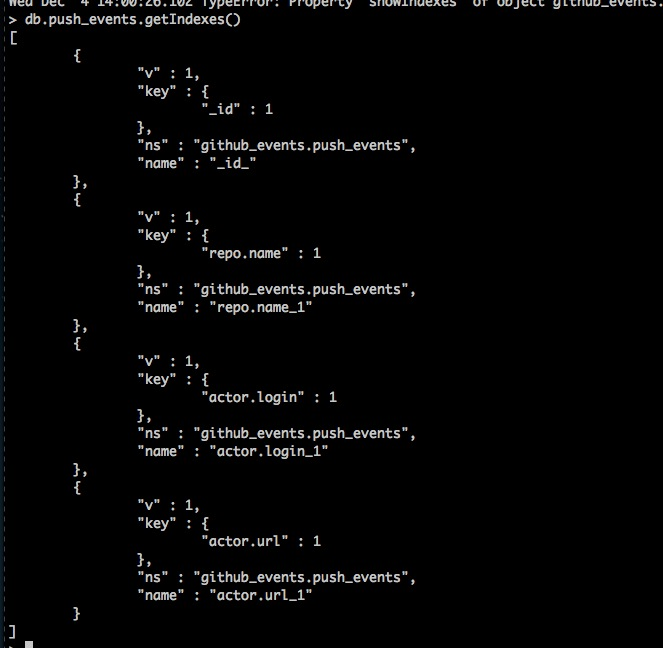
### Indexes

Indexes in Mongo are similar to indexes in RDBMS. They reside in memory where it is easy to search. By default, for each collection ‘\_id’ is indexed. This in our case was the auto generated ID.

New indexes are added using the *ensureIndex()* method. This takes some time to build if a lot of data has already been inserted. We query the repository name, the URL and the actor to find the top repo and users. We noticed that the queries were slow initially, so we created index on the fields and there was a significant increase in the speed.

|  |  |  |
| --- | --- | --- |
| Query | No Index | With Index |
| Select Top Repo | 5 | 3.2 |

Below are the screen shots of the indexes we created.







## Part II -- Neo4j

Neo4j is a persistent graph based storage solution, which is released under an open source license. We used Neo4j to store ‘events’ that we collected from Github’s public feed.

### Domain Context

Github is a ‘social coding’ website that allows developers to host their source code and other developers to interact with it.

Github publishes 11 events as a part of its public event feed. These events describe everything that happens on github.com including creation of code repositories, pushing code, making pull requests, forking repositories, watching repositories, etc.

We filter out 4 such events --

WatchEvent -- created when a user starts ‘watching’ a repository. Watching a repository is a low involvement action that enables the user to receive updates about a repository.

ForkEvent -- created when a user ‘forks’ a repository. Forking a repository gives the user their own copy of the repository that they can then modify. This is a more involved action than watching.

PullRequestEvent -- a pull request is when a user asks the owner of the original repository to accept some code that the user wrote in their forked repository (or branch). This is the first action that demonstrates that the user has actually written code as opposed to just reading it.

PushEvent -- a push event is generated when a user pushes out code to a repository that they have commit rights to. This demonstrates the highest level of involvement with a given repository.

### Objective

In brief, our recommendation engine suggests users to follow by analyzing these events on repositories that one follows.

We rank other users to follow based on what kind of actions they take on repositories that you are interested in. In order of importance, the events we rank are --

Push -- most important, gets the most points while ranking.

PullRequest -- gets the second most points

Fork -- gets fewer points because user hasn’t contributed yet

Watch -- gets the minimum points

Apart from analyzing repositories that you are directly interested in, we can also analyze repositories that the users we recommended you follow are interested in; lending this recommendation engine a multi-level exploration side.

### Technical Implementation Details

The Ruby programming language was used along with the excellent ‘neography’ gem package that allows Ruby to have an object oriented interface to Neo4j’s API.

Users and Repositories were both stored as ‘Nodes’ in Neo4j and the events themselves were stored as ‘Relationships’.

In the first cut, we stored the relationship weights as attributes on the Neo4j Relationship object themselves, but soon discovered that changing the weights to fine tune the recommendation engine is extremely tedious since thousands of nodes need to be modified.

We then moved to store the EventTypes in Neo4j and handled weights from the Ruby environment at run time where they could be easily tuned.

### What did not work and how it was solved

1. Storing weights for recommendations as attributes on Neo4j relationships. Wanting to fine tune the recommendation engine by changing weights is expensive because millions of relationships need to be then updated. What works is storing the Event types in Neo4j and calculating weights and aggregating things at run time.
2. Py2neo is a python library for integrating with Neo4j but at this point is rather premature and has an inconsistent API. Moving to Neography which is a Ruby library yielded fast results because of its elegant API.
3. Enforcing uniqueness from the Application layer turned out to be extremely slow when dealing with records more than a few thousand. Using Neo4j indexes to index all nodes and relationships to enforce uniqueness at the data store layer.

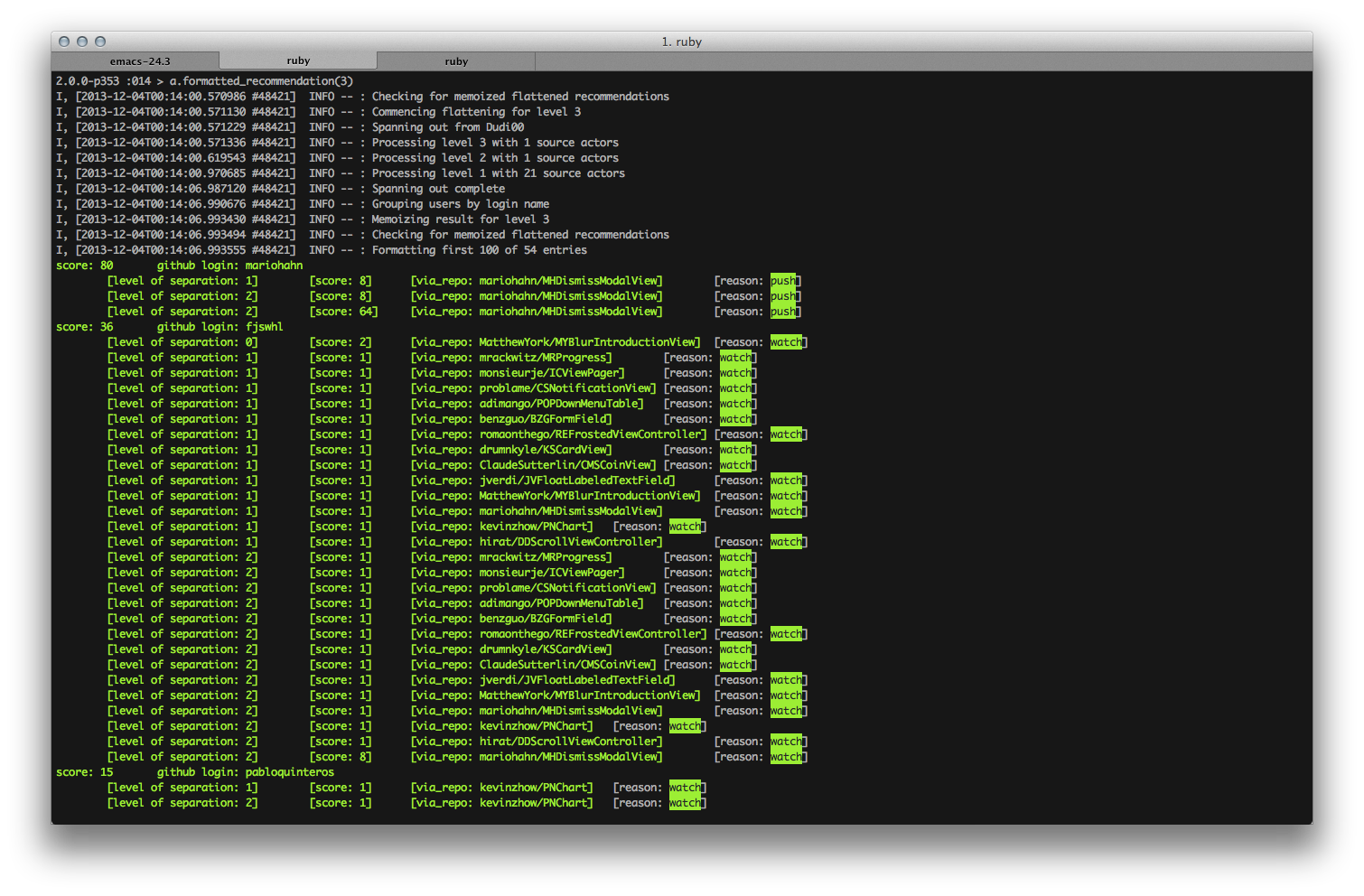
### Software Used

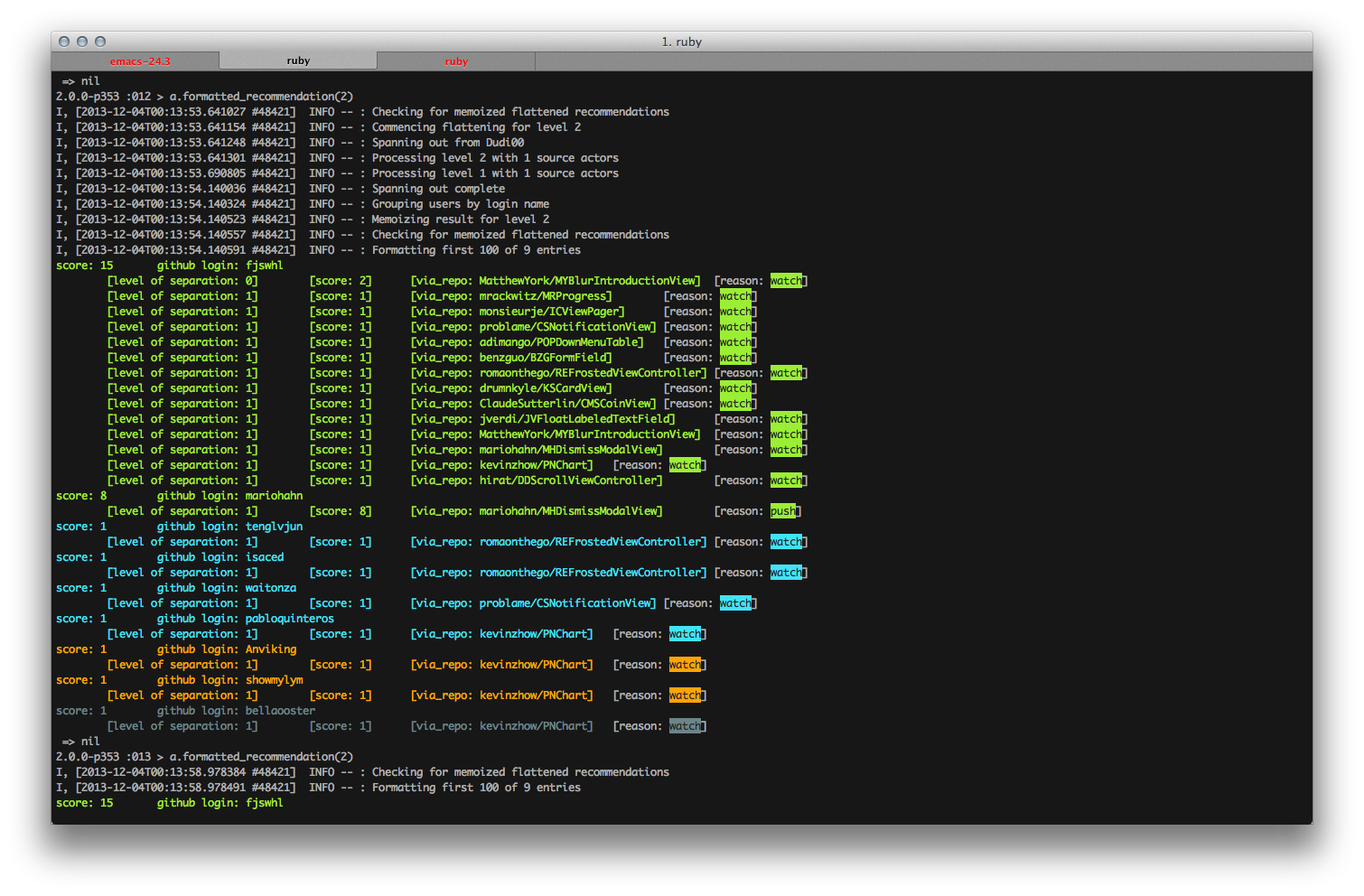
Neo4j v2.0.0-RC1 (Linux Mint 15 Olivia)

Ruby v2.0.0p353 (Mac OS X Mountain Lion)

Neography v1.2.3 (Mac OS X Mountain Lion)

Screenshots of the Recommendation Engine





## Visualizations for top-ten repositories with most IssueEvent activity:

We analyzed the data for one year from GitHub for IssueEvent. The goal was to find top ten repositories with most open and closed issues.

# Below are some of the charts that we generated using [DSPL: Dataset Publishing Language](https://developers.google.com/public-data/)

