Description of System Architecture

data-source.py

The system architecture of this project is very similar to the ninemultiples lab. The project starts with the data-source.py file in which we fetch 3 API calls to github's service (via HTTP requests). Each call returns a dataset of 50 repositories of a given programming language placed within the url parameters. In our case, Python, CSharp, Java were chosen. The data-source.py file takes these 3 JSON responses and combines them into a line string structure, e.g:

```
progLanguage + "\t:" + item['full_name'] + "\t:" + dateformatted[0] + " " +
dateformatted[1] + "\t:" + str(item['stargazers_count']) + "\t:" + descriptionRe
```

Where progLanguage contains the programming language (e.g "Python"); repositories full name, the date in UTC format when it was pushed at, number of favourites, and the description of the repository (regex'd to keep letters). And between each item a token is used to separate the values, '\t:'.

As each repository goes into that format, it gets appended to the data in which it will be sent to the Apache spark cluster (spark_app.py) over TCP connection.

spark_app.py

The spark_app.py contains the majority of the computation. First and foremost, data from the data-source.py is batched every 60 seconds. Each batch run will first split the data. Earlier it was mentioned data has a token in which it separates each attribute, so the data gets mapped based on splitting each line on "\t:". The next step is where (i) (iii) (iv) gets processed.

The splitted data is now mapped again in a key,value relationship, in the format of:
Key: repo full name , **Value:** (pushed at datetime, programming language, # of stars, desc)
updateStatebyKey will be run on this using the method aggregate_repos.
This is essentially a map reduce algorithm in which the keys (i.e name of repo) will be reduced based on the pushed at datetime, it will compare the datetime's and choose the min; min essentially returns the earliest date (doesn't choose newest date) so we keep old repos and ignore new incoming duplicate repo data.

Then for each rdd, it will be processed via process_rdd function. This is where data get's filtered flatmap reduced etc. Steps for (i) (iii) and (iv) get processed. Then create an sql context on that chosen rdd, in which SQL queries will be used to retrieve the data.

E.x (i) uses that rdd defined from above, since the rdd as of this moment contains all UNIQUE repos (due to the map reduce mentioned above), we can simply use 3 SQL count queries based on where = language, i.e where = "PYTHON" etc.

E.x (iii) as mentioned above, all repos in the rdd are unique and based off the 'oldest' version of the repo (if there was a duplicate). So, simple SQL avg query on star_gazers

E.x (iv) This one was slightly different as I've done it on 3 sql tables, for simplicity as each table contains different language's words. This was possible as it used the initial rdd from

above, and filtered on language, and then flatmapped via split on each whitespace to separate each word and attach a value of 1 (filter again to remove empty spaces). This will be then applied in a mapreduce where the value keeps adding so we can get a frequency.

Once all of the above steps are done it will be combined into a json file in which it will be sent over to the dashboard, flask web app.

Only step (ii) doesn't go through updateStatebyKey instead it uses regular reduceByKey, since we're only doing based on 60 seconds it's not required to check previous repos. First it gets filtered based on current time when spark is doing that function subtracted by the pushed at time of the repo. Since we're not allowed to contain duplicate repos in that 60 seconds they will be filtered out using the same aggregate_repos reducer function.

Once this step is done it will also be sent over to the flask application.

flask_app.py

The flask application takes in the data via json dump and then gets stored into redis to store values. These values are then taken and mapped on the html file in each try clause statement. **NOTE*** This may be a bug or an error in my code, the line graph does not render until at least 2 values are stored (i.e 2+ batch runs)** . I'm very sorry about that.