PROGRESS REPORT

DISTRIBUTED COMPUTING FINAL PROJECT

Project Title:

“Search Optimization in an Elastic Database”

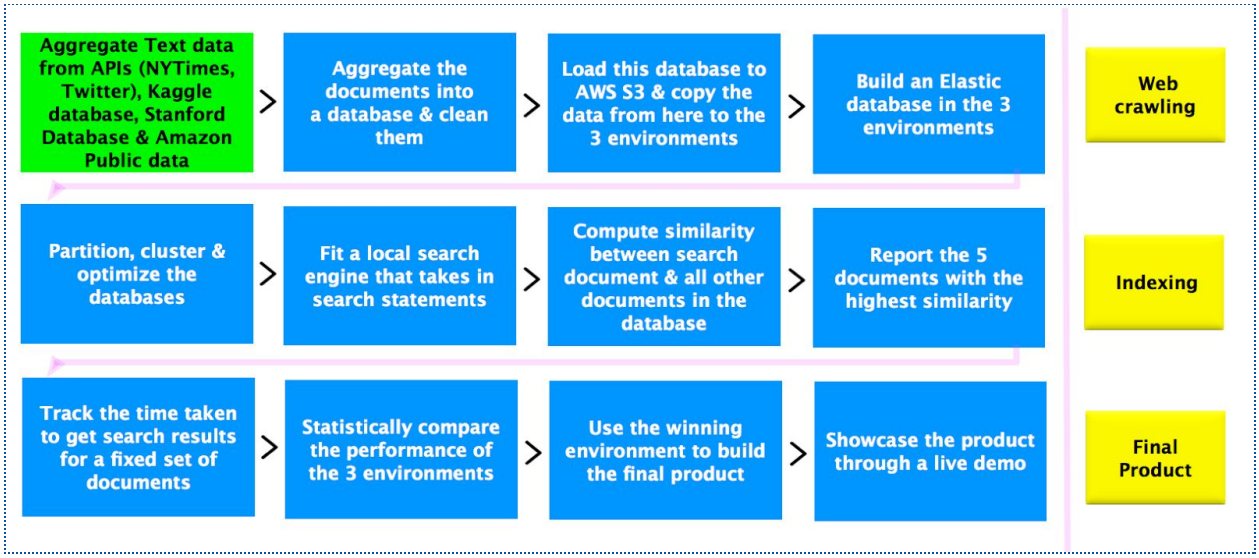
Candidates:

Swetha Reddy (20357646)
Tate Campbell (20352330)
Abhishek Singh (20363537)

1. Project Introduction

We have built a Multi-tenant Elastic Text Database that could recommend the 5 most relevant documents to the user for his given search in less than 5 second. We have successfully employed mainstream techniques such a TF-IDF, Cosine Similarity to get this application running. We have built and tested our algorithm on 3 different frameworks: Python, Spark & Hive. As Spark outperformed the other 2, we shall build our final product based on spark and shall create a Web App for the same for the final presentation.

Our Idea is to present the User with the 5 most relevant documents to his search statement. The below Flow chart walks us through the step by step execution of this project.



2. Database Creation

We used a variety of sources to create a large text database of nearly 4 million rows (1.4 GB) and stored it on AWS S3 http://bit.ly/1NkmtE3. Below is the split of our database sources:

1. Hillary Clinton's Email database (*569k rows)*, direct download using curl command
2. NYTimes database (28k rows) using its API, please see the scripts (**nyt_scrapper.py, abhi_nyt_scrapper.py**) to see the process
3. Stanford Sentiment Database (50k rows), direct download using curl command
4. Amazon Public data (3258k rows), direct download from the shared folder
5. Twitter data (4k rows) using its API, please see the script (**twitter_data.py**) to see the process

The code for database creation is in the file (**Database_creator.py**).

3. Mathematics of Document Search

A prefered way of suggestion documents to users is to identify documents that are most similar to user's current search. The notion of similarity is based on the similarity scores derived using the formula of Cosine Similarity. Now this concept becomes slightly tricky when we are applying it to text vectors instead of numerics. This tutorial (**Cosine_similarity.ipynb**) walks you through a much simple application of what we have done at scale. To further improve our Search algorithm, we shall be:

1) Getting rid of Stop words of the English language
2) Getting rid of special characters

4. Scalable applications of search databases

We are fortunate to be doing a project in a field that is already a commercial success with the help of leading High Tech pioneers. The table below shows the importance and visibility of this field..

Database	Text database size (GB)	Average search speed (secs)
Our Database	1.5	5
English Wikipedia	15	1
World Wide Web (on Google)	60,000 (Approximate)	.2

5. Implementation Framework 1: Python

Local Environment: Basic python packages such as **pandas, re, string & numpy** were used to extract the most relevant documents to the search string. The average search time using Python was **40 seconds**. Please refer to the code (**Top5_documents.py**) to see the step by step execution.

Distributed Environment: I used AWS **(16 node AWS EC2 - r3.4xlarge cluster**) to run the algorithm in a distributed setup. To enable this I had to transfer my code (copy-paste) & database (Filezilla) to the EC2 server and install python packages (Pandas, numpy & **multiprocessing**) on the instance. My search times reduced to an average of **15 seconds** on the Distributed setup, which is an improvement of nearly 300% from the **60 secs** on local setup. Please refer to the code (**Parallel_Top5_documents.py**) to see the step by step execution.

6. Implementation Framework 2: Spark

Local Environment:

The cosine similarity measure implementation in PySpark was quite similar to that in Python, the main difference being Spark makes use of RDDs which can dramatically improve speed and performance. The average search time for the local pySpark implementation was **35.6 seconds**. Please refer to the code (**pyspark_cosine.py**) to see the step by step execution in PySpark.

Distributed Environment:

A 2-core r3.4xlarge cluster EMR cluster on AWS with equivalent tech specs to those described in section 5 was used to compute cosine similarities between the query and all documents in the database using pySpark in parallel. The average search time for the distributed PySpark implementation was **6.40 secs**.

7. Implementation Framework 3: HIVE (In process)

Local Environment:

We ran a AWS cluster (**m3.xlarge 2 Master nodes**) without any bucketing to check for performance. We loaded the data from **S3** to **EMR hdfs** and loaded the hive table with this hdfs data. We have done basic searching to get 5 most relevant documents for our search statements. Find the code in local_hive.sql.

Distributed Environment:

We ran a 320 Gibs AWS cluster (**r3.4xlarge 2 Master nodes**) with 32 buckets in the table to check for performance. We loaded the data from **S3** to **EMR hdfs** and loaded the hive table with this hdfs data. We have done basic searching on the search statements to list the 5 most relevant documents to our search statements in the distributed environment. Please refer to the file (distributed_hive.sql) for the code.

8. Presentation Deliverables

We plan on presenting these 2 deliverables on our presentation day, to showcase our work to audience:

1. Slide Deck, highlighting Concept, implementation, use case & results
2. A Spark based Web App that would demonstrate a live execution of our algorithm

9. Final Deliverables

For the final deliverables after the presentation, we shall be submitting the following 2 things:

1. A detailed final report that shall cover all our work in entirety with conclusions
2. All updated codes of what we have used in the process with comments

10. Implementing Parallelization

We have implemented parallelization across all the 3 frameworks to enable effective distributed computing. For any major computing job, parallel processing substantially outpaces serial processing of code. This is due to higher RAM usage in parallel computing as compared to serial computing.

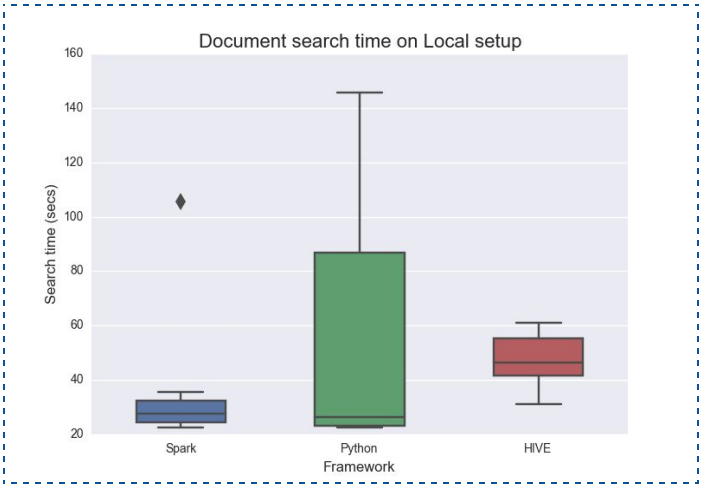
We have benefitted in all the 3 frameworks with much higher search speed for our input string. However parallelization worked best in spark due to its inherent efficient build structure that has very little start time. Python helped with parallelization to but the improvement was somewhat limited in search speed. Also we had to use an external library "**multiprocessing**" and write additional functions to enable parallel processing in python. This goes on to prove the supremacy of spark for this application and hence why we chose it to be engine of our final product.

11. Split of Responsibilities

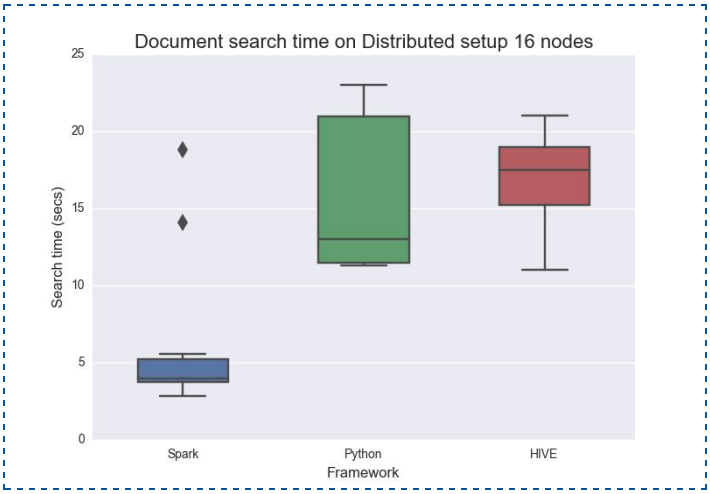
Responsibility	Primary	Secondary	Completed
Project proposal	Abhishek	Tate, Swetha	Yes
Database creation & Web scrapping	Abhishek	Tate	Yes
Spark Implementation	Tate	-	Yes
Python Implementation	Abhishek	-	Yes
Hive Implementation	Swetha	Abhishek	No
Progress Report	Abhishek, Tate	Swetha	Yes
Presentation & Web App	Abhishek, Tate	Swetha	No
Final Report	Swetha	Tate, Abhishek	No

12. Plots based on our search results

Local Setup: We searched for 10 popular strings on our search database in the 3 different environments and recorded the time taken to obtain the most similar strings .



Distributed Setup: We searched the above 10 strings on our search database in the distributed setup as well and recorded the time taken to obtain the most similar strings.



2 observations from this experiment:

1. Parallelization boosted search speed in all 3 environments (Look at the Y-axis of the 2 charts)
2. Spark benefited with nearly 450% improvement in search speed or 450% reduction in average search time