SEARCH REDUCE

Sanika Paranjpe, Jagrut Chaudhari, Kshitij Wagh
sparanjp@iu.edu jagchau@iu.edu kswagh@iu.edu

Luddy School of Informatics, Computing, and Engineering,
Indiana University Bloomington, December 2022

ABSTRACT

Our application provides a search capability for finding resources (pdf, text, and word) on the S3 service offered by AWS. This functionality would help personal users of S3 and Organization users to find documents at a quick pace.

Index Terms— S3, Search, Map Reduce, Containerization, Cache

1. INTRODUCTION

Search Reduce allows users to upload documents to the S3 Bucket service provided by Amazon. The main intention of the application is to provide users with a quick and relevant search using a Machine Learning model. The application allows users to upload pdf, text, and word documents to S3.

The application uses the services like Docker and Kubernetes for creating and autoscaling the nodes in order to provide a reliable service. The application uses Redis caching mechanism to fetch the records faster. It also uses MongoDB for database operations, Flask platform for backend operations, and React for frontend operations.

2. RELATED WORK

We found out that some of the cloud applications like Google Drive, One Drive, and Dropbox use MapReduce to optimize the search functionality on the documents that exist on their cloud platform. Also, AWS has a service called Athena that uses S3 object metadata like customer identifier, category, received date, etc. to do a search based on them, but there is no single public service that provides a search on the contents of the S3 documents. Our one-stop search application would help users/organizations to manage and efficiently search relevant documents thus saving time.

3. METHODS

3.1. Fronted:
We used React JS to develop our front-end application. Please refer to figure 1 for the frontend user interface. The frontend application would perform the following functions:

1. Search for keyword/keywords: When a user types something in the text box, the React application would call an API and fetch the most relevant documents based on if the searched keyword is present in the document or if any relevant words are present in the document. Refer to figure 2.
2. Add document feature: The small plus icon on the bottom right corner of the User Interface will allow the user to upload documents (text, pdf, or word) on S3 - an object storage service provided by Amazon Web Service. Refer to figure 3.
3. Statistics: The left section in the UI would allow the user to know the statistics such as the number of documents processed and the total number of keywords processed (based on the documents). Refer to figure 4.

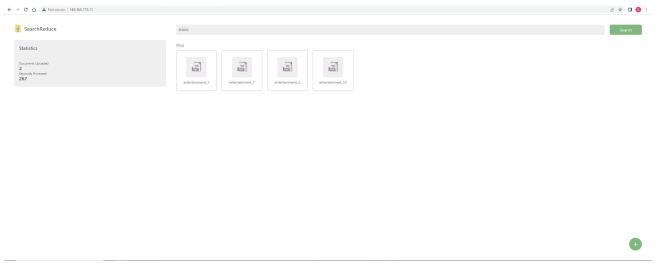

figure 1

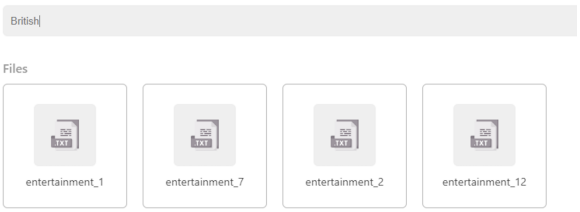

figure 2

figure 3

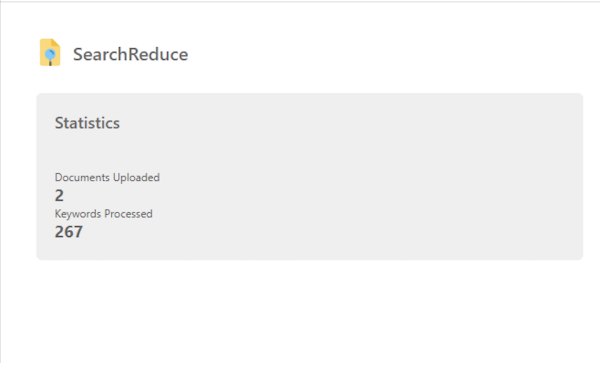

figure 4

3.2 Backend:

We used the Flask platform to write all the application program interfaces for our application. The flask application would perform the following functions/APIs:

Search: This Api would perform the following functions:

1. Get the request string from the frontend
2. Split the request string into words and add them to a collection in MongoDB. If the word is already present in the collection, update the count of the word searched.
3. User the threading mechanism on each word to fetch the relevant document names
4. If the word does not exist in the cache the flow will check the availability of documents listed in MongoDB.
5. If the word is found in MongoDB, it will return a list of documents
6. If the word is not found in MongoDB, Wordnet form NLTK would fetch all the relevant words(synonyms) associated with the searched word and then return the list of documents. WordNet is a pre-trained model for finding synonyms and antonyms of words. We have used WordNet to find synonyms related to the searched word
7. If the word is found in the cache, the code flow would fetch the documents list from Redis cache and return the list for documents

The response of the API would be as follows:

Request:

```
{"search_string": "Hello everyone"}
```

Response:

```
{"success": True, "data": {'_id': 1,
'stats': {'documents_uploaded': 2,
'keywords_processed': 273}, 's3_docs':
[{'name': '1670814245680_business_99',
'extension': 'txt'}, {'name': '1670814
405053_business_98', 'extension':
'txt'}, {'name':
'1670814851771_business_90',
'extension': 'txt'}]}}
```

Upload s3 Document: This API would perform the following functions:

1. Get the uploaded object name by the user from the request
2. Update the list of S3 object names with their extension in MongoDB
3. User the threading mechanism to run Map Reduce on the uploaded s3 object.
4. In the Map Reduce function the flow would fetch the object from S3, run NLTK on it to remove all the garbage words, and generate a temporary file with the output of Map Reduce
5. The code flow would then update a collection in MongoDB with the words in the document, the document name, and the number of occurrences of that word in document
6. Then a function would be called to update the number of keywords processed in the document, and the number of documents uploaded in MongoDB
7. While the thread is running, the code flow will return the statistics as a response.

The request and response of the API would be as follows:

Request:

```
{"s3_object_name": "abc.jpeg" }
```

Response:

```
{"success": True, "data": {'_id': 1, 'stats': {'documents_uploaded': 2, 'keywords_processed': 273}, 's3_docs': [{'name': '1670814245680_business_99', 'extension': 'txt'}, {'name': '1670814405053_business_98', 'extension': 'txt'}, {'name': '1670814851771_business_90', 'extension': 'txt'}]}}
```

Statistics: This api would perform the following tasks:

1. Gets the number of words processed and the number of documents uploaded, and a list of s3 objects uploaded from MongoDB

The response of the API would be as follows:

Response:

```
{"success": True, "data": {'_id': 1, 'stats': {'documents_uploaded': 2, 'keywords_processed': 273}, 's3_docs': [{'name': '1670814245680_business_99', 'extension': 'txt'}, {'name': '1670814405053_business_98', 'extension': 'txt'},{'name':'1670814851771_business_90', 'extension': 'txt'}]}}
```

.

3.3 Caching:

Caching can be used to fetch the data that is most frequently used so that it would reduce the turnaround time for sending a response. We have used the caching platform called Redis, to store the s3 object names, along with the keyword it is associated with, so that it wouldn't have to make a connection with the database to fetch the records, and instead return the data that is stored in the cache. Refer to figure 5 for the working of our caching mechanism.

Following is the kind of object that we are storing in Redis cache:

Structure:

```
keyword : ["url1", "url2" ...]
```

Example:

```
"cat" : ["example1.txt", "example2.txt" ...]
```

3.4 Database:

NoSQL databases are one of the most popular kinds of databases, as they store the data in the form of key-value pairs, and fetching of data is way faster than the traditional relational databases. We have used MongoDB as our NoSQL database. Refer to image 5 for the working of the database operations.

Following is the kind of object that we are storing in MongoDB cache:

Structure:

```
{'_id': word, 's3_docs': [{"doc_name": document_name", "total_occurences": number_of_occurences}] ,[{"doc_name": document_name, "total_occurences": number_of_occurences}]}
```

Example:

```
{'_id': "hello", 's3_docs': [{"doc_name": "hello1.txt", "total_occurences": 8}] ,[{"doc_name": "hello2.txt", "total_occurences": 5}]}
```

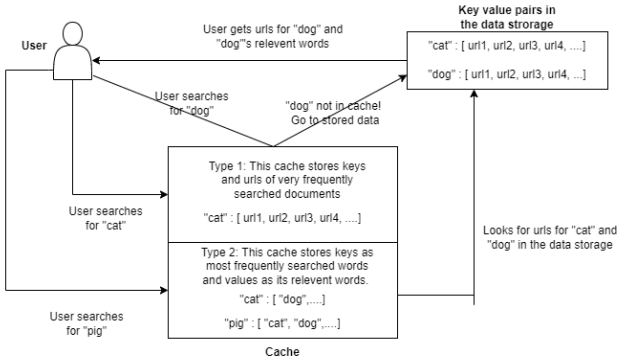

figure 5

3.5 Deployment and Hosting:

Deployment and Hosting is the most important and crucial step in terms of providing the services to the user. Following are the services that we have used and the methods we followed to deploy our project.

Docker: We created docker files for our frontend and backend applications. We are maintaining a docker repository for both of our applications. After making any changes in the code, we rebuild the docker images and push them into the repository. Following are the commands that we use to build and push the image to the Docker Repository:

Build docker image:

```
docker build --tag sanikaparanjpe/flask-repo:server-image-v4 .
```

Push docker image:

```
docker push sanikaparanjpe/flask-repo:server-image-v4
```

Kubernetes:

Figure 6 shows the structure of Kubernetes deployment done for this project. The ingress service is the entry point for our application. The ingress service is responsible for calling react and flask applications. All of the Kubernetes deployment will be inside a node. The only call going outside the node is for S3. We have created Cluster IP services for React, Flask, Redis, and MongoDB. Cluster IP Service is responsible for calling the pods which will serve the API call. A Service will be connected to a single pod or a few replicated pods. Every pod will consist of a docker image with the functionality for that service. A persistent volume is provisioned using a Storage class and has a lifecycle independent of the individual pod. So, if a pod dies, the data will still persist. Persistent volumes would be allocated to Redis and MongoDB.

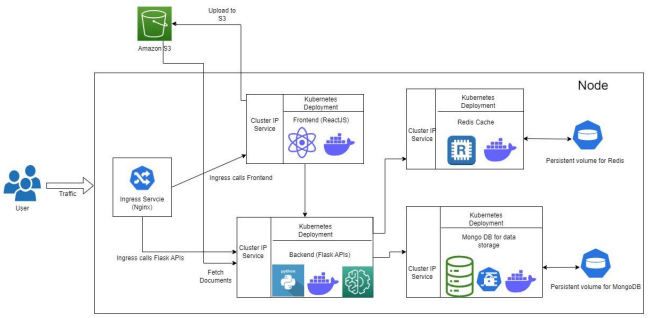

figure 6

Apache:

Apache is installed on the Linux instance so that it would forward requests to the Ingress service in Kubernetes. Apache configurations would have proxy and reverse proxy settings that would forward the request coming to the Jetstream instance IP to the IP assigned to the Ingress service. Refer to figure 7 for the working of the Apache server.

Jetstream:

We have used Jetstream to create a Linux instance. A public IP is assigned to the instance so that any services running on the instance could be accessed through that IP.

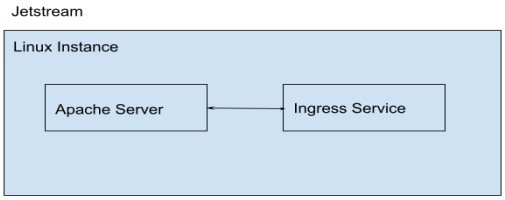

figure 7

4. RESULTS

4.1 Fast Search:

Reasons for Fast Search

1. Using Redis decreased the turnaround time of the API response in a list of documents stored in Redis.(logs presented in figure 8)
2. Using MongoDB reduced the search time by half as the data can be directly accessed via key.
3. Threading: Using threads for search keywords, reduced the time considerably, as the tasks would run in parallel.(logs demonstrating advantages of using threads figure 9)

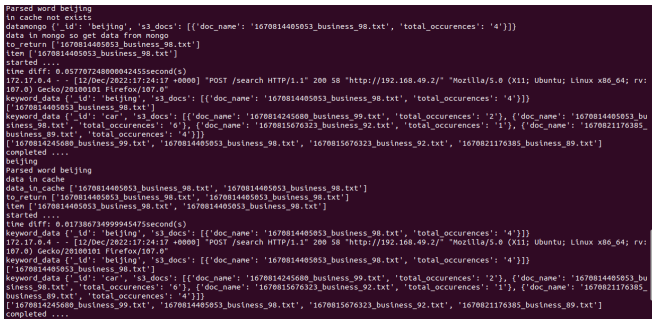

figure 8

figure 9

4.3 Experiments conducted:

Relevant Search

When a user searches for a specific word if that word is not present in MongoDB as well as cache then the Relevant Search algorithm is used. We are using Wordnet to find the synonyms of the word being searched. The below images show that the word “characteristic" is searched but was not present in the database, hence its related word “feature” was used to output documents for the search. Below 3 figures (Refer to figure 10) show logs, documents, and documents searched respectively.

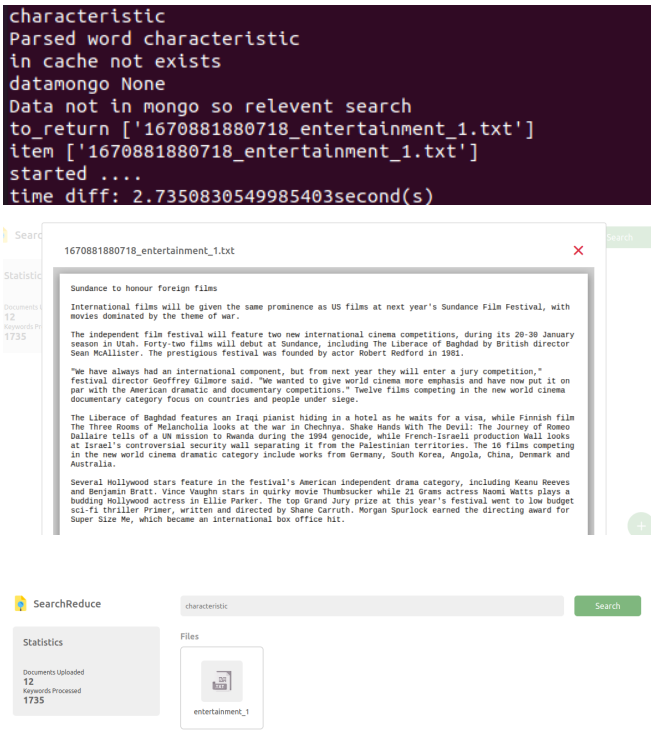

figure 10

Uploading documents

When a user searches for a specific file, a thread is created for each word in a string thus running all the thread tasks in parallel. After the threads are done executing, it would combine all the documents for the words, and send it as a response. So for example, if a user searched for a string, and it has 4 words in it and it takes 40ms to fetch the record, it would be 10ms in our case, as for each word a thread will spin up.

Scaling

Scaling allows regenerating pods in Kubernetes when they die so that the system stays more resilient. Scaling can happen in terms of many factors. The most common factors on which scaling happens are Memory and CPU usage. In our case, if the CPU usage reaches 80% of its capacity, the new pod will spin up to assure the availability of services. The following command is used to provide autoscaling in Kubernetes Architecture:

Command:

```
kubectl autoscale deployment server-deployment --cpu-percent=10 --min=1 --max=10
```

Refer to figure 11 for visualization of auto-scaling. The scaling of pods occurs when traffic is simulated on the server.

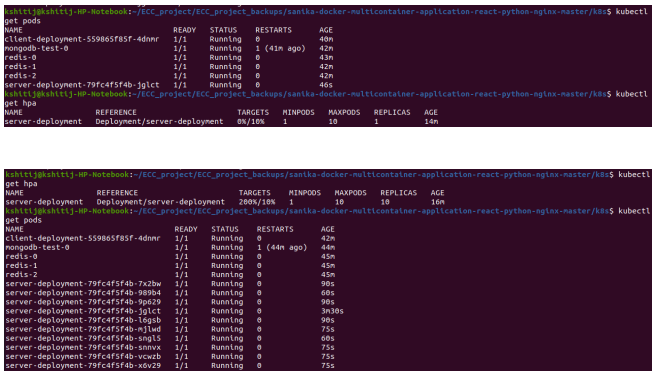

figure 11

5. CHALLENGES

Following are the challenges that we faced while working on the project:

1. Writing Kubernetes configurations to create pods, deployment, and service was the biggest challenge. It was also difficult to create different components that our system is using such as Redis, mongo, frontend(react), and backend(flask) in Kubernetes.
2. Mongo and Redis being storage components needed more configurations for Storage Class, Config Map, Persistent Volume, Persistent Volume claim, and Stateful set
3. We were facing issues in writing to the Redis cache. We had created 3 replicas of Persistent volumes in Redis; one being master and the remaining 2 workers. Our workers were read-only, hence data was not getting inserted into the cache. After changing the configuration from "read-only-replicas" to "no" we resolved the issue.
4. Ingress service is an Nginx server of Kubernetes which we are using to call our APIs. We faced issues while calling server APIs via Ingress. So we created another HTTP configuration in the ingress service.
5. When we deployed our app on Jetstream, we spent a lot of time figuring out how to use a public IP address to access our application. This was achieved using Apache.
6. One of the ongoing challenges is that we are unable to access the S3 resources from our Flask deployment on Jetstream. The behavior is such that it works sometimes but most of the time it fails while fetching the object from s3, and running the MapReduce task on the fetched object. To understand this issue, we got access to the pod where the Flask deployment was running and tried to connect to domains like “google.com” or “yahoo.com” using “ping”, but it gave us the response as “bad request”. Then we tried to ping to the google DNS servers using their IPs - “8.8.8.8” and “8.8.4.4”, and we got a proper response. We then came to the conclusion that this is a DNS

resolution issue. After some more research, we found out that the Alpine Linux image (python:alpine3.10) that we are using for the Flask deployment has DNS resolution issues, and a lot of people have faced this same problem with the Alpine image. We tried applying a few solutions like restarting the core DNS deployment provided by Kubernetes, adding a ConfigMap with Google's DNS servers, and changing the ConfigMap of CoreDNS deployment, but nothing worked in our favor. We even thought of a solution to get all the deployments on the host network, but the way we have set up Redis (With 1 Master and 2 Worker odes) we were only able to get the Master node on the host network. Another solution is to deploy Redis on a single pod, but it would totally defeat the purpose of our Redis architecture (one Master and 2 Worker nodes). Changing the Alpine Image to a full-fledged Ubuntu image would also be one of the solutions, but doing this would subsequently increase the size of the docker image, which in real-world scenarios is a big "No" because of the number of resources it will consume. Doing this will also increase the costs of the deployment. Also, we started facing this issue after we deployed our project on Jetstream. All the deployments and functionalities work perfectly fine on the local machine. Our current goal is to keep working on this problem until we find a solution for it.

6. CONCLUSION

Working on this project was truly a learning experience. This project enabled us to learn and work hands-on on varied technologies and tools such as Docker, Kubernetes, AWS S3, Redis, Mongo, and Apache. The WordNet model from the NTLK library helped us search for words relevant to words in the document, and assign document names to those keywords. In this case, even if the user searches for words that are not in the document, they will get some meaningful results. Also, the removal of all the scrap words from the documents would also contribute towards a meaningful search.

The usage of Redis as a cache and the usage of threads helped in a faster response time thus not keeping the user idle for a long time. The usage of Kubernetes and Docker helped in the easy and fast deployment of React, Flask, Redis, and MongoDB applications. Because of the debugging capabilities in Kubernetes, finding and fixing the issues in the deployment was hassle-free. Managing the container images with Docker Hub also made deployments very easy on Kubernetes, as it picks up the image that we provide in the deployment configuration. So after a minor code change, the developers won't have to make any changes in the deployment. Different versions of the applications can be saved in the Docker Hub Repository with different tags as well.

Also, Flask being a lightweight framework, was perfect for our applications' needs, and writing and managing Apis in Flask was hassle-free. For the frontend part, React being a very dynamic framework, building all the UI components was effortless. Overall, the technologies that we used were a perfect blend for our application.

7. RESOURCES

Github Link (follow the readme file to replicate the working of the project):
https://github.iu.edu/sparanjp/ECC_project

Jetstream deployment:
http://149.165.173.11/

7. REFERENCES

[1] Deploying Redis Cluster on Kubernetes:
https://www.containiq.com/post/deploy-redis-cluster-on-kubernetes
[2] Deploying MongoDB on Kubernetes:
https://devopscube.com/deploy-mongodb-kubernetes/
[3] Dockerize a Flask- React project:
https://blog.miguelgrinberg.com/post/how-to-dockerize-a-react-flask-project
[4] Docker Documentation: https://docs.docker.com/
[5] Kubernetes Docs: https://kubernetes.io/docs/home/
[6] Horizontal Pod Scaling:
https://medium.com/@myte/kubernetes-autoscaling-abf11314346e
[7] Wordnet (Relevant Search reference) :
https://www.tutorialspoint.com/how-to-get-synonyms-antonyms-from-nltk-wordnet-in-python
[8] Apache Installation:
https://www.mirantis.com/blog/quick-tip-use-apache-as-a-proxy-server-to-access-internal-ips-from-an-external-machine/
[9] Links we followed related to the DNS issue:
https://discuss.kubernetes.io/t/coredns-and-problem-with-resolving-hostnames/12593
https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/use-the-host-network
https://stackoverflow.com/questions/65181012/does-alpine-have-known-dns-issue-within-kubernetes
https://github.com/kubernetes-sigs/kind/issues/442
https://blog.yaakov.online/kubernetes-getting-pods-to-talk-to-the-internet/
https://discuss.kubernetes.io/t/kubernetes-pods-do-not-have-internet-access/5252/13
https://stackoverflow.com/questions/53313967/kubernetes-container-not-able-to-ping-www-google-com