Overview of my implementation

My implementation to the problem involved creating a Java Web app deployed into a Jetty web container instance within a Docker container on the Google Cloud Platform. The GCP dashboard shows that service was load balanced between two Docker containers. The solution was developed using Eclipse for Java Enterprise Edition; built with gradle; gradle for dependency management; deployment tested locally with a local instance of Tomcat 8.5 and API testing conducted with Postman.

My solution uses the Jersey for handling the Restful services to receive the requests. The Restful classes then invoke a StoreService to storing and retrieve the data. This class implements an interface and in a more realistic example I would use dependency injection to the implementation of this interface and other dependencies around the solutions. However, for simplicity, the dependencies in the solution are tightly coupled. If I was use dependency injection framework, I would consider Guice or Spring.

The StoreService splits the data in an array of words; sorts the array using List.Sort(); invokes a file storage utility class to save the file locally and then a Google drive storage to save the file to my personal Google drive account. Unfortunately, the application generated a Google 403 scopes permission error when I tried to inject the file into Google drive. At the time of writing this document I have not resolved this error, so although the code is written to save and retrieve from Google drive is written, it has not been fully tested. As a result, the only the files are saved to the local drive system temp folder. The code to invoke the Google drive utility class has been commented out.

The retrieval restful end point uses the same frameworks and send the result back to the client. The List.subList() method is used to return the desired number of words. Again, the code to retrieve the file from my Google Drive has been commented out, but it is all there and once the permission issue is resolved I would expect it to work fairly quickly. The HTTP POST verb is used for the ‘save’ end point and the HTTP GET verb is used for the ‘retrieve’ end point.

The code is available on my GitHub repository https://github.com/adam-birr/DeepMatter

Extension of the Solution

To meet the extended requirements, I would break up the solution up in number of ways. I would separate the saving of data from the retrieval of data in to different microservice applications. I would replace the RESTful interface with a websocket interface. This would enable the handling of data as it was being streamed into the application.

I would store the data as it was streamed it and sort it on the fly. For this I would use a binary tree. This would work by assessing if the word to the sort is greater or less than the current node. If less, then we traverse to the left node and if greater than the current node we would traverse the right node. If the node we want to traverse to is empty, then we store the word there. This storage would have a best case sorting time complexity of $O(n \log n)$ if the tree is balanced and $O(n^2)$ if the data is already sorted when it comes in based on the number of comparisons required to store each value. The space complexity required would be $O(n)$ for both the best and worse case scenarios. There are more efficient sorting algorithms (e.g. a LSD Radix sort), but I believe that they would need to work on the entire data set rather than being able to it on the fly. I would also store the reference node containing the first sorted word to save time traversing the tree when reading it back out.

I would also send the first the first 100,000 words to a microservice implementing a cache. This would sort the entire data set to sort, so more efficient sorting algorithms could be used, such as a LSD Radix sort. This data would then be stored in another database as key value pairs. The key being the filename and the value being the complete sorted list of words. In English the average word length is less than

five characters, which would make the unencrypted data around 500k in size. This might use a different database technology to the storage of the tree structure.

I would also have an in-memory cache which would store the last *n* number of retrieved lists of 100,000 words or less. This would be another microservice. This would store data in a structure like a LinkedHashmap. Once full the oldest entry would be removed to make room for the new one. The LinkedHashmap preserves the ordering of entry. If an entry is retrieved from the map, it should be removed and reinserted to avoid that entry becoming stale.

The retrieve webservice would be handled by a separate microservice so that it be could scaled individually and also for reasons of single role of responsibility. As with the save service I use a websocket interface instead of a RESTful service. This would stream the data to the client. It would first try from the in-memory cache microservice. If not found it would go to the second level key-value cache microservice. Finally, if greater than 100,000 words long it would open another websocket connection to the tree storage microservice and transverse the sorted tree to stream the result.

Supporting Infrastructure

There is a choice between static and dynamic provisioning of infrastructure. Most of my experience has been in deploy to static cloud infrastructure. This would involve knowing how environments are required up front to run all the various microservice, manually configuring load balancers/reverse proxies to direct traffic to them. Provisioning them in various data centres around the globe and setting up localised DNS entries which will direct traffic to the nearest reverse proxy. The data could be stored in a high availability distributed database architecture such as Cassandra. I don't have any experience of Cassandra, but I do have experience of CouchDB, but this would not be suitable for this level of scale and data volumes, but I believe that Cassandra would be suitable. Static provisioning does require a lot of setup and would mainly be suitable when the demand and load is fairly constant so the entire infrastructure would be fairly well utilised. The advantage is that it could be fairly decoupled from eco system of any one cloud hosting provider and if load is predictable, the costs could be lower than an on demand service. The disadvantage of this architecture would be when load on the system is unpredictable and the time taken to set up and maintain supporting infrastructure such as reverse proxies, load balancers, DNS entries, etc.

For dynamic infrastructure, the choice of PaaS would be between Azure, Amazon Web Service EC2 and Google Cloud Platform. The choice must be taken wisely as it might be hard to migrate away from the chosen eco system once development has made some progress. However, the microservice should be written in such a way that those which handle the client requests, business logic and data presentation are agnostic in regard to supporting data services. If GCP is chosen, the choice of data store is between, Cloud Datastore, Big Table and BigQuery. For this application, Big Table would appear to be more appropriate due to the it being able to scale to hundreds of petabytes and can handle millions of operations per second[1]. Although this is still an order of magnitude short of the 1 exabyte requirement. The advantage of dynamic elastic infrastructure is that scalability of infrastructure is on demand and transparent. Thus, it is easier to set up, quicker to deploy and potentially requires less maintenance.

To make this a world wide architecture, the environments would need to be provisioned in multiple regions with load balancing based on IP addresses to route the nearest region[2].

[1] https://weidongzhou.wordpress.com/2017/06/10/google-cloud-sql-vs-cloud-datastore-vs-bigtable-vs-bigquery-vs-spanner/

[2] https://cloud.google.com/load-balancing/docs/https/cross-region-example