Product Based Search Engine Microservice

Haridher Pandiyan[1], S.B. Prapulla[2], Paul Sabu Kodiyattil[3], Abdulazeem Yazari[4] and Siva Rama Krishna Kondapalli[5]

[1]Student, Computer Science and Engineering, RV College of Engineering, Bangalore, India

[2]Assistant Professor, Computer Science and Engineering, RV College of Engineering, Bangalore, India

[3,4,5]Digital Platform Manager, Mast Global, Bangalore, India

Email: [1]haridher21@gmail.com, [2]prapullasb@rvce.edu.in, [3]spaulkodiyat@mast.com, [4]ayazari@mast.com and [5]skondapalli@mast.com

***Abstract* —** A product search engine is a key element in the functioning of any e-commerce application. It indexes products in real time and produces fast results to queries entered. Currently the solution running on the organization's website uses a microservice that passes the queries entered, to a third-party service provider that does the indexing and searching. This is a paid service and hence is to be replaced by the open source search engine, Apache Solr. In this paper, we explain the microservice built, using the go-solr package along with the go-kit microservice framework in developing the microservice to replace the pre- existing paid service.

***Keywords*—** Faceting, fuzzy search, microservices, parsers, Solr, SolrCloud.

I. INTRODUCTION

Search engines are the reason the internet is so popular, that provides relevant results to the user queries within mil- seconds, searching through millions of records and returning the best matches. There are different types of search engines though, based on their use cases, The most commonly used one is the Google Search which is a web crawler based search engine, that scours the internet and indexes new websites and returns the most appropriate website to the user's search. But for e-commerce companies, most of which will be having a website of their own displaying the products that they sell and can be bought from this online portal. Here a product-based search engine will be provided that has indexed only the product related data and has been optimized with respect to the same. The most popular or largest product search engine is that of Amazon, called the A9. Any search here returns only products that are available from Amazon.

In a similar fashion, the organization' website does the same for the products it sells. Currently it is using the service of a third-party provider to do the indexing and searching of the products.

The call to this is available in their microservice named Keyword Search v6. The new service aims to remake this service as Keyword Search v97 microservice that has Apache Solr performing the indexing and searching.

Apache Solr is an open source search engine build off Lucene which is developed in Java and performs the optimal searching and indexing processes. Elastic search is another viable option, but Solr search is more suitable to the enterprise data use cases.

Section II elaborates some of the ways Solr has been used, mainly targeting web-based searches. Section III describes the architecture of the microservice developed. Section IV discusses the methodology. Section V details the results of the testing analysis and Section V1 forms the conclusion.

II. LITERATURE SURVEY

Though Solr has been being used for over 10 years, most of the papers published focus on using Solr as a web crawler-based search engine, that deals with indexing webpages. Enterprise data from an e-commerce website would deal with products, that would each have numerous attributes that apply to all, but also attributes that only apply to it.

Hence a schema can be developed to form a structure to the data enabling for better storage and retrieval. D. Yi and W. Youyu [1] tackled such a case by comparing data from a Shopping website between a regular database search versus that provided by Solr.

They highlighted Solr's Vector Space Model (VSM) that represents documents as vectors allowing for computing between similarity of terms as degree between vectors. Similar comparison with structured data was done by S. Tahiliani and A. Bansal [2] between Solr search and Hibernate search. They compared various types of searches such as wildcard search, faceted search, etc.

Their comparisons pointed which search was better for each scenario and showed that Solr search would be better suited for enterprise data for more basic and faceted searches that are the main type of search on such online stores.

Others used Solr for webpage indexing. H. Ma et al [3] explored the idea of a using Solr as a vertical search engine using Apache Nutch as the web crawler, and having a filtering system that would only extract tourism based data and index the same in Solr. A. Wang [4] explored a similar approach but with electronic product information and more expansions and optimizations on the web crawler Heritrix. A novel JE segmentation module was developed so that this would deal with Chinese character data, opening capabilities for other languages along a similar system.

L. Ma et al [5] focused on the Mongolian language with lot of homographs, hence had to transform the encoding to a common unicode format, Latin being chosen. A corpus for this and transcoded it with the Latin characters, Solr then indexed different documents and transform them with their word association model trained on the corpus to index them in Latin, and finally the values entered in the search would get converted into Latin and searched by Solr. Supun Nakandala and Sachith Withana [6] applied Solr as a backend database to store metadata information of various scientific data collected for better archiving and reuse of collected data. G. Simonini and S. Zhu [7] focused on a faceted search to retrieve the best n facets using Bayesian networks and Solr.

In contrast to Solr, researchers like D.F. Murad et al [8] utilized elastic search as the search tool in their application with an emphasis on the keyword matching ability while others like R. Surendran et al [9] tackled its usage for a distributed environment of a dynamic grid computation. Too further explore this idea of elastic search, a deeper study of its term matching capability was performed by [10] N. Kumar and A. Pradhan [10] in their word root finder. Overall based on different use cases studied, Solr has shown to be a better match for applications dealing with enterprise data that can be defined by a schema.

III. SYSTEM ARCHITECTURE

A. Go-Kit Microservice Architecture

Microservices developed using Go language, generally utilize the Go-kit package. This package contains various sub- modules that provide HTTP data transfer support, decoder and encoder options, utilities for logging and metrics, as well as support for consul that can be used to provide the service mesh for the microservices developed. With the support of all these tools available, a general microservice architecture is commonly followed. This involves separation into layers chiefly a transport layer, endpoint layer and service layer, with additional logging and instrumentation (for the metrics) as shown in Fig. 1.

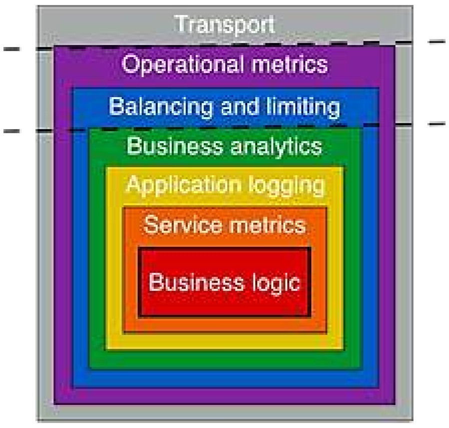


Fig 1. Go-Kit Architecture

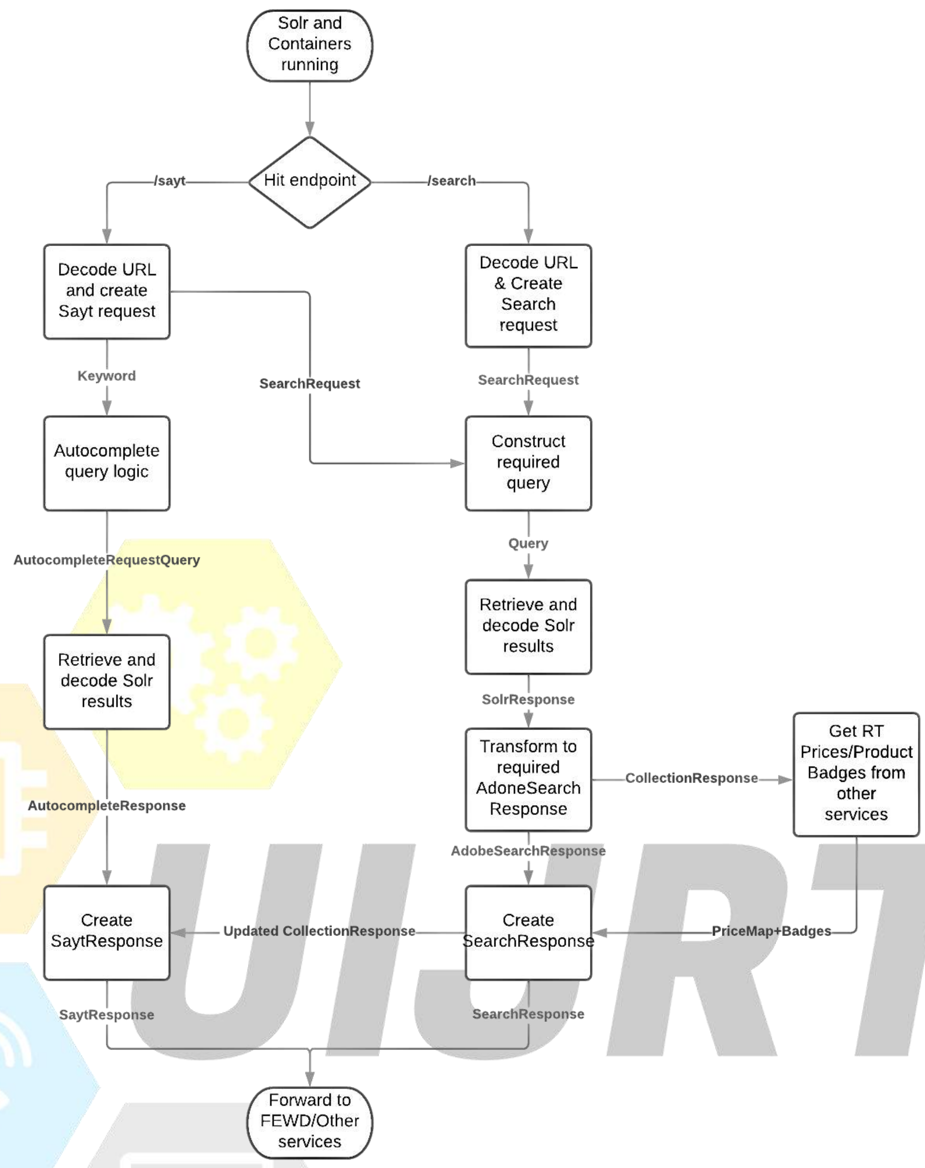
The service layer deals with the core business logic that we wish to provide in our microservice. Typically, it includes calls to business logic functions stored in an api file and makes function calls to other microservices via proxies to use their information as well. For every service function, there will be an endpoint function that provides an abstraction mapping the services to the transport layer, thereby exposing the service methods. The transport layer then manages the server logic to expose the endpoints, implement the required decoder and encoding functions for each service, depending on HTTP, gRPC or other transport being used.

B. Overall Architecture

The application is part of the microservices that run the organization's product website. The different microservices are developed in Go language, and the codebase is available across over on Bitbucket, from where a Joyent Triton container is built. These containers can communicate with other ser- vices via the service mesh provided and handled by Hashicorp Consul. For services that require tools that aren't available in Go, separate containers are hosted on Azure to handle them. As such, in the case of this implementation of Keyword Search, a SolrCloud instance is up and running on Azure, that can be hit from keyword search.

C. Execution Flow

Once the containers and SolrCloud instance is up and running, and consul agents are active and key value pairs set for authenticating the connection to the service mesh, search requests from the client user at the website will be directed to the search and the Search as you type (SAYT) endpoints. The server listening on those endpoints will decode the respective URL requests and will extract the different key value pairs and form the



Solr and Containers running
Hit endpoint
/sayt
/search
Decode URL and create Sayt request
Decode URL & Create Search request
Keyword
SearchRequest
SearchRequest
Autocomplete query logic
Construct required query
AutocompleteRequestQuery
Query
Retrieve and decode Solr results
Retrieve and decode Solr results
SolrResponse
Transform to required AdoneSearch Response
CollectionResponse
Get RT Prices/Product Badges from other services
AutocompleteResponse
AdobeSearchResponse
Create SaytResponse
Updated CollectionResponse
Create SearchResponse
PriceMap+Badges
SaytResponse
SearchResponse
Forward to FEWD/Other services

Fig 2. System Flowchart

search request. From there the service methods will call the required modules. For the search, this begins with the construction of the query to be sent to Solr, requiring the keywords and different filters applied and forming a logical congregation. After which, the required go-solr library methods are utilized to create the query instance object that will be sent to Solr for performing the required search. The results obtained will be in a nested Solr response format that will require decoding with the help of the mapstructure module before being transformed to the previous Adobe response format. Finally, calls are made to other services to get certain other real time product information (available in the Collection part of the response) such as pricing, and this final response is then passed onto the service mesh and be available to the proceeding services and front end.

The SAYT front requires 2 different calls to Solr, one being a search call itself and the other dealing with the autocomplete suggestion module, where suggested words are produced based on the current keyword entered. The two results are finally merged to form the Sayt response that's forwarded to the service mesh. This entire system flow is presented in Fig 2.

D. Functionalities

The key functionalities provided by the new keyword service shall include the following:

1) *Keyword search*: The user can pass any string in the search bar and will be returned relevant results. Searches generally are done as full text matches. To facilitate a higher level of user experience/satisfaction, Fuzzy search too is supported thus accounting for misspelt words passed in the search by the user.

2) *Phrase search*: By this, we mean that higher priority is given to a specific phrase (ordering of words) occurring in the searches over ones that just contain part of the phrase in the searches.
3) *Faceted filtering*: The product website will have facets that the user can select in order to narrow down the searches. Different facet categories with different options exist, the required logical mapping of these filters must be held to return the right results.
4) *Faceted results*: Once any search is done, or facets added and a new search done, there is a requirement to state the number of products of a given facet category/type amongst the returned results. This again adds to the user experience.
5) *Sort options*: Rather than most relevant, users may want to see the results generally in the order of price or ratings or for newer products among the returned results. This same capability will be provided.
6) *Autocomplete*: When the user is typing the keyword that they want to search in the search bar, suggestions will be provided that are in the form of wildcard results/autocomplete. How it is displayed, will be handled by the front end team, but as part of the keyword search service, they must be passed as a part of the SAYT response.

IV. METHODOLOGY

The project was development was split up into two phases to separate out the concerns, allowing for focus on Solr's inclusion in the first and implementation into the former service in the second.

A. Phase 1: Wrapper Service

Solr was first setup with the creation of containers with appropriate configurations such as shards and replication factor and studied with the example data and the sample feed provided from the DataHub team. From this a field list and type were created to be supplied to Solr before ingesting the data and allowing automatic schema design.

A basic application was first built with the core microservice structure with a status check. Development was then done to provide a basic connection to Solr and retrieve example data before moving onto the sample feed. The decoding of the URL get requests via Postman was done using the go-kit url library to extract the fields and supply the request to the service. Within the service, the creation of the query is made and then the search to Solr is called before retrieving the results. The results are in a nested map interface slice that needs to be mapped to a required response struct, which is then sent back with the help of an encoder. The structs are signified with a json construct, for the response object to be easily read as a json.

Next steps included, figuring out the facet calls in Solr, the response object to be decoded and what facets are needed to be read and produced later, This also added complexity to the query creation process as the facet filters require appropriate logical mapping to filter correctly. Finally, sort option was also included.

This service was still functioning independent and requires creation of corporate key value(KV) pairs over on consul to include in the service mesh, but being a POC, the decision was to directly replace the api logic in keyword search with that of the wrapper service.

B. Phase 2: Implementation within the original microservice

As stated above, the changes were pushed onto the main keyword search service. Required adobe calls were replaced with Solr. Overall, 2 main services are to be provided, Search and Sayt. Sayt stands for search as you type and requires a slightly different request and response.

The service involves getting the product information like in search (first call to search/solr is made), getting certain facets (second call to Solr) and finally the suggestions slice (third call). This suggestions slice required a different approach as Solr only provides records as results and not word suggestions. But using grouping and the specific field only selector, satisfactory suggestions were attained. Thus, completing the second service. Unit testing and benchmark tests were followed to prove the validity and provide some performance analysis that is covered in the following section.

IV. ANALYSIS AND RESULTS

The performance analysis was carried out using benchmark tests. Repeated tests were performed, and their scores averaged out to produce the following insights.

A. Analysis of Different Query Parsers

A query parser is the component responsible for parsing the textual query and converting it into corresponding Lucene Query objects. It is generally specified via the defType parameter which stands for default type. Different query parsers are designed for different use cases depending on the data stored and the complexity of the query that will be required of.

Table 1: Nanoseconds/operation(op) for different parsers

Pass No.	Parser Type				
	Lucene	Dismax	E Dismax	Complex	Simple
1	24563145	23833456	26090622	31263145	25334215
2	23940584	23554669	25789137	28017281	25190928
3	24288350	23613452	25336907	28429611	24915621
4	23278211	23700278	25722656	29089310	24732345
Average	24267572.5	23675463.75	25734830.5	29199836.75	25043277.25

Some of the query parsers that were tested are:

1) *Lucene*: The standard default query parser. The key benefit of the standard query parser is that it supports a robust and fairly intuitive syntax allowing you to create a variety of structured queries. But on the downside, it's very intolerant of syntax errors.
2) *Dismax*: It's designed to process simple phrases (with- out complex syntax) entered by users and to search for individual terms across several fields using different weighting based on the significance of each field. The DisMax query parser supports an extremely simplified subset of the Lucene QueryParser syntax and rarely produces error messages.
3) *eDismax*: It's an improved version of the Dismax parser, supporting the full set of complex queries that Lucene parser can define, as well some additional parser specific parameters, hence why it's called extended Dismax.
4) *Simple*: This parser allows a person to type whatever they want for a query to represent. This parser will then do its best to interpret what to search for no matter how poor the composed request may be.
5) *Complex*: It permits complex query logic via potentially performing multiple passes over query text to parse for any nested logic in phrase queries. The first pass takes any phrase query content between quotes and stores for subsequent passes.

The different parsers were tested via the benchmarks and the following datapoints were obtained as shown in Table 1. A graph for the same has been plotted as shown in Fig 1.

Based on the results returned, it is clear that Dismax has performed the best. But Dismax is optimized to handle simple queries and does not support complex queries, which will be the case for the keyword search application and hence Lucene will be considered better. The results are actually very consistent with other studies too returning similar results as eDismax is slower than Lucene and complex involves multiple passes.

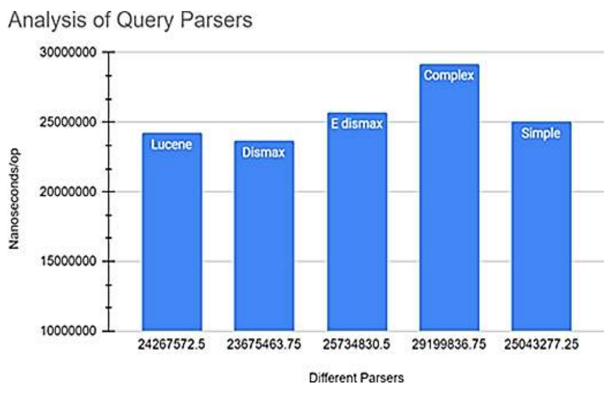


Fig 3. Plot of Analysis of Different Query Parsers

B. Analysis of Different Number of Filters

Filters refer to the facets applied by the user to narrow down their search results. Thus, a comparison was done to check how the added filters would affect the speed of the overall search. The following tests were performed using the Standard Lucene parser and the results obtained in Table 2 and plotted as a graph in Fig 2.

Table 2: Nanoseconds/operation Vs Number of Filters

Pass No.	No. of Filters			
	0 filters	1 filter	2 filters	4 filters
1	31152491	31480012	12951704	11991156
2	31480012	14426219	11040314	10553007
3	28239739	12689602	10754049	9237753
Average	30290747	14165108	11582022	10593972

One would expect the added filters to delay the search further as more checks will have to be done, but due to in built optimizations in Lucene's implementation, all comparisons are made in a single pass with very little difference and hence the time comes more down to the number of records that were hit as a match and returned.

Thus, with more filters, lesser hits will be made and this the decrease in the response time.

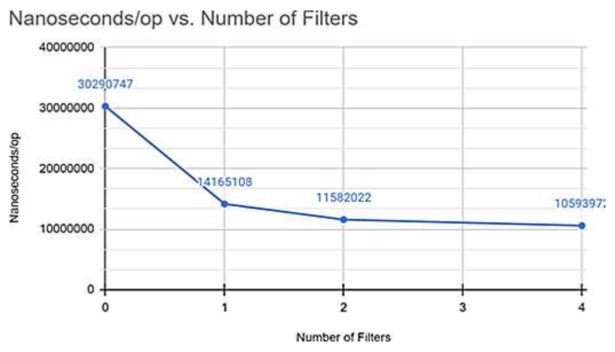


Fig 4. Plot of Analysis of Different Number of Filters

C. Analysis of Different Number of Different Fuzzy Levels

Fuzzy search was also added. To have Solr perform it, a simple "~" followed by a number specifying the order of fuzzy search is added to the q parameter of the query. Though with phrase searches, this refers to the slope parameter which checks the distance between words in a phrase. So, a comparison was done between the direct full text match and other fuzzy levels, and wildcard search. The following results were obtained as shown in Table 3. A graph was plotted using the same as shown in Fig 3.

Table 3: Nanoseconds/operation Vs Fuzzy Levels

Pass No.	**Fuzzy Levels**			
	Level 0	**Level 1**	**Level 2**	**Level 4**
1	21843327	26824816	31152491	26824816
2	24875931	29085295	31480012	35912582
3	22103520	27363918	28239739	29550603
Average	22940926	27758009	30290747	30768226

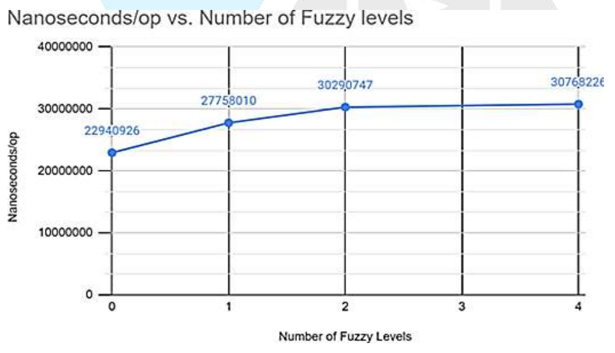


Fig 3. Plot of Analysis of Different Fuzzy Levels

Based on the graph, it is clear that with increase in fuzzy levels, more words and hence records will be matched, this leading to an increase in the response time. But the line tends to flatten out beyond a point. This is owing to the fact that beyond a certain degree, all permutations and swapped combinations have already been matched and no more will be caught on increasing the degree. Thus, the results may vary depending on the word searched. But Fuzzy level 2 is finally chosen for any search, allowing the user to enter spelling mistakes and still find the results adds to the user experience. Finally, not mentioned on the graph, but wildcard search did also perform well, roughly the same as fuzzy level 2, but it doesn't apply well to our application and hence not considered.

VI. CONCLUSION

Thus, the new microservice developed performs a slightly better search service than the former service. It has taken into account some of the missing capabilities of the current Adobe search and provided them via Solr. Being open source, there is quite the cost cutdown, as well as more control over the search, allowing for a deeper development for future search optimization and modification/addition of existing services. Solr has served as an ideal search engine for this enterprise use case, providing all the required functions, and many more to be included. The Go-Solr interface by developed by vang822 provides a sufficient interface to talk to Solr using Go, with all its functionalities. Overall, with proper development and review, this microservice should be bound to replace the former service soon. Performance analysis was carried out using benchmark tests. Repeated tests were performed, and their scores averaged out to produce the following insights.

REFERENCES

[1] D. Yi and W. Youyu, Shopping Website Search System Based on Solr, 2019 11th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), Qiqihar, China, 2019, pp. 708- 711, doi: 10.1109/ICMTMA.2019.00162.

[2] S. Tahiliani and A. Bansal, Comparative Analysis on Big Data Tools: Apache Solr Search and Hibernate Search, 2018 3rd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT), Bangalore, India, 2018, pp. 164-170, doi: 10.1109/RTEICT42901.2018.9012331.

[3] H. Ma, W. Du, S. Xu and W. Li, Searching Tourism Information by Using Vertical Search Engine Based on Nutch and Solr, 2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA), Honolulu, HI, USA, 2019, pp. 128-132, doi: 10.1109/SERA.2019.8886775.

[4] A. Wang, Design and Implementation of Vertical Search Platform for Electronic Product Information, 2017 International Conference on Robots Intelligent System (ICRIS), Huai'an, 2017, pp. 101-104, doi: 10.1109/ICRIS.2017.32.

[5] L. Ma, W. Bao, W. Bao, W. Yuan, T. Huang and X. Zhao, A Mongolian Information Retrieval System Based on Solr, 2017 9th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), Changsha, China, 2017, pp. 335-338, doi: 10.1109/ICMTMA.2017.0087.

[6] S. Nakandala et al., Schema-independent scientific data cataloging framework, 2015 Moratuwa Engineering Research Conference (MER- Con), Moratuwa, Sri Lanka, 2015, pp. 289-294, doi: 10.1109/MER- Con.2015.7112361.

[7] G. Simonini and S. Zhu, Big data exploration with faceted browsing, 2015 International Conference on High Performance Computing Simulation (HPCS), Amsterdam, Netherlands, 2015, pp. 541-544, doi: 10.1109/HPC- Sim.2015.7237087.

[8] D. F. Murad, T. Darwis, M. Z. Achsani and F. C. Utami, "Elasticsearch Analyzer In Broad Match Advertising System," 2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI), 2018, pp. 415-420, doi: 10.1109/ISRITI.2018.8864308.

[9] R. Surendran and B. P. Varthini, "Integration based large scale broker's resource management on friendly shopping application in Dynamic Grid computing," 2012 Fourth International Conference on Advanced Computing (ICoAC), 2012, pp. 1-6, doi: 10.1109/ICoAC.2012.6416830.

[10] N. Kumar and A. Pradhan, "Personalized Terms Derivative: Semi-supervised Word Root Finder," 2016 International Conference on Information Technology (ICIT), 2016, pp. 260-264, doi: 10.1109/ICIT.2016.059.G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.