5 Tools and implementation

5.1.Tools and sharding

A combination of existing open source tools may be used to match the architectural elements proposed on Chapter 4. For this implementation, we used the following tools:

- Redis: It is an open source, BSD licensed, key-value store [18]. It is often referred to as a data structure server where keys can reference strings, hashes, lists and sets. The key-value stores sought for the implementation are a very good fit to Redis. The basic collections - DB::Set and DB::Array - map directly to existing Redis' structures of sets and hash, respectively. DB:: TwoWayArray was built using the two-way indexed collection technique described above. Redis is an in-memory datastore, associated with great performance. It has a built-in replication mechanism, based on a one master read-write node with several slaves read-only nodes copying from it, and all nodes can be setup to periodically persist the memory status to disk. It lacks server-side sharding, but the architecture designed can transfer that to the API implementation without loss of functionality;
- Resque: It is a Redis-backed for creating background jobs, placing these jobs on multiple queues and processing them later [19]. A very robust job queues and workers framework, it provides an easy scheme of job creation and queuing. It can handle job failures and retries, schedule jobs with a plugin and has the built-in characteristic of being decentralized - workers only need access to the datastore that backs its queue data to work, without needing to report to a central server. Conveniently, Resque uses Redis as its datastore, so the same network built for the indexes will be able to handle the job queues with redundancy and scalability;

- Elasticsearch: The most used open-source engine to keyword search is the Apache Lucene. However, Lucene is a very bare engine, designed to work with local documents. Some document stores are built on top of Lucene - Solr is the most famous example. Elasticsearch [20] - is a document store like this, featuring a RESTful API using JSON to store, search, and modify documents. It also has built-in mechanisms to be deployed and distributed over several nodes, and with those it meets the requirements for the node index.

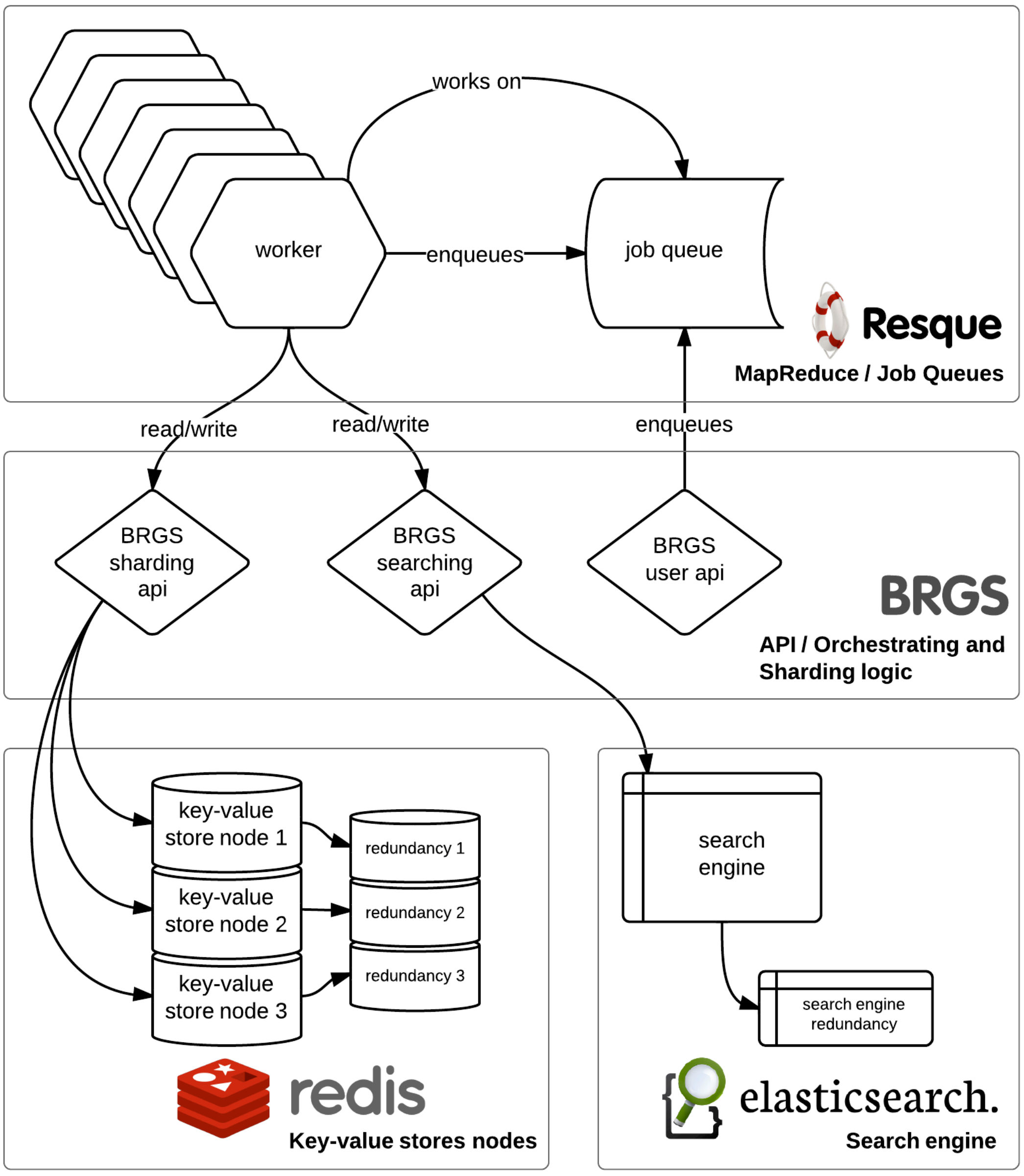

Figure 11 - Architecture with matching tools

5.2.Sharding

In the proposed architecture, the orchestrating API is responsible for implementing the sharding strategies. We combined two strategies in our implementation. The first strategy consists of splitting up the stores in half by key index. So indexes, supporting data structures, and the sparse matrix are stored on the key-value store, and keys are always composed by either a node, full-path or template index. In our implementation, the value is stored in different key-value server nodes, if the related index is odd or even. With this strategy, all nodes, full-paths, and templates with an odd index are stored on the *odd_indexes_store* key-value store, and those with an even index are stored on the *even_indexes_store*.

The functionalities listed on Chapter 4 call for two types of queries over the sparse matrix quite often. They are set to retrieve data from a single row or a single column. To expedite these types of retrieval, the implementation also uses a second sharding strategy, consisting of a trade-off. We duplicated the sparse matrix data to allow faster sequential retrieval by either row or column. In this strategy, for the first copy of the positions, all values of odd rows are stored on the *odd_rows_store* key-value store to expedite retrieval and the other rows stored on *even_rows_store.* For the second copy, the same odd/even strategy is used to store columns on the *odd_columns_store* and *even_columns_store,* respectively.

5.3.Networks

Each of the three architectural elements was chosen because they could provide a way to build a network of hosts, making as much resources available as needed. For each of the tools chosen, some of their design characteristics implied that their networks should be built somewhat differently.

Redis keeps its data in-memory. It is single-threaded and can be configured to work on a read-write master with several read-only slaves copies of it. Instances running Redis could then use a lot of memory and bandwidth, but had limited use for several cores. With sharding logic handled by the orchestrating API, each Redis host would need to store a fraction of the indexes and the sparse matrix. For this implementation, we chose to build the Redis network with seven master hosts. One of the hosts is dedicated to Resque, and the other six will store respectively the *odd_indexes_store*, *even_indexes_store*, *odd_rows_store*, *even_rows_store*, *odd_columns_store,* and *even_columns_store*. Each of those hosts is backed up by a slave host.

Resque workers are also single-threaded each, but with lower requirements for memory. Several workers can share a server with multiple cores, limited by the available bandwidth they consume. In this implementation, we used ten worker hosts, each with four workers. The workers shared the job pool on the dedicated Redis host described earlier, and a web interface was also setup on the same server running that instance.

Finally, *elasticsearch* has built-in configurations for multiple hosts. However, during this implementation, there was no need to built a network for performance, so it remained with a single master host with a slave host for backup.

AWS zone	us-east-1d
AWS EC2 AMI	ami-7539b41c (Ubuntu Server 12.10)
AWS EC2 Redis Instance type	m1.large
AWS EC2 Workers Instance type	c1.medium

Table 10 - Details of the AWS instances used

5.4.Details of the execution of RDF parsing and matrix building (steps 1 & 2)

The first step of the execution followed the proposed two MapReduce sub-steps. In the implementation, the first sub-step was called *admission* and the second sub-step *spider*. They are both exposed through REST methods.

The *admission* sub-step was implemented to receive a URI as a POST parameter. This could be a local *file://* URI, or a public *http://* URI. The controller of this method downloads the file and creates an *rdf_admission* job to split the input file implementing the Algorithm 4. The implementation assumed the input files were in the NTriples format, and future implementations could add before this point another sub-step to convert files from other formats, or allow other forms of input, such as SPARQL end-points.

For every segment of the input file, an *rdf_parsing* job was created. This job implemented Algorithm 3 to find nodes, sources, and sinks of the segment.

The second sub-step was triggered manually, after all the input segments were processed. The *spider* sub-step was implemented as a GET method that creates a *graph_spider* job. This job implements Algorithm 6 to create a *graph_crawler* job for each source, which, in turn, implements a breadth-first search of Algorithm 5, without recursion and the map functionality of Algorithm 8.

The second step, i.e., the sparse matrix assembly, was triggered automatically by the *graph_crawler* execution. For every full-path found during its execution, the full-path was immediately stored, and a *matrix_builder* job was created with that full-path as a parameter. This job implements Algorithm 7 to store the associated full-path positions in the sparse matrix.

We faced a difficulty during these two steps. We tried to reproduce the results of the tensor-based proposal [1] with the same datasets - the DBLP and LinkedMDB as found on [7]. However, the number of full-paths found was many times larger the numbers shown in the proposal, and the execution time reached many hours instead of the minutes described in the proposal. In order to complete the experiment, we chose a smaller dataset - the STW Thesaurus for Economics [21].

We were then able to complete these two steps that had an average execution time of 2573 seconds (42.89 minutes) taken from 5 measurements, as listed on Table 11, along with how many times each job was triggered, the average time for each experiment, and the total execution time for each job time. This time is the total processing time only, excluding the manual triggering. The final dataset, after the default LZF compression reached 361.7MB, for a total of 1,334,563 full-paths found. Other details are listed on Table 12.

It is important to note that the implementation could not reproduce the results found on the tensor-based proposal that inspired it, given the excessive amount of full-paths produced by the proposal, and future work could improve on that, by pruning out less meaningful full-paths by ranking their relevance.

	execs.	t_1	t_2	t_3	t_4	t_5	Avg.	Total
rdf_admission	1	1.08	1.19	1.12	1.08	1.21	1.14	1.14
rdf_parsing	2	126.41	129.68	127.37	127.89	129.34	128.14	128.14
graph_spider	1	1.04	1.29	1.12	1.05	1.17	1.13	1.13
graph_crawler	218	147.84	133.27	141.27	143.45	145.23	142.21	775.06
matrix_builder	1334563	0.05	0.05	0.05	0.05	0.05	0.05	1,668.20
								2,573.67

Table 11 - Steps 1 and 2 jobs execution times

Nodes found	13,434	Edges found	38
Sources found	218	Full-paths found	1,334,563
Sinks found	6,589	Templates found	573

Table 12 - STW RDF graph details

5.5.Performance of the queries execution

The data retrieval queries were exposed in the implementation as RESTful actions with the same name of the query, all using the HTTP GET method. To measure the performance of the methods, we used the Apache Benchmarking tool [22]. This open-source tool allowed experimenting with concurrent access and automatic test repetition. In Table 13, we list the queries minimum, average, standard deviation, and maximum response time for 10.000 queries with 20 concurrent users. All queries were executed from other AWS EC2 instances, using the internal addresses so network trip times were minimized.

Query	ms/req			standard deviation
	min	**avg**	max	
node_query	51	**564**	2,174	86.2
path_query	11	**75**	1,639	57.6
final_node_query	8	**83**	3,097	112.0
path_intersection_query	15	**95**	1,138	60.8
path_intersection_retrieval_query	112	**871**	3,163	340.8
path_cutting_query - from start	12	**79**	1,602	57.4
path_cutting_query - to end	10	**79**	1,316	58.5

Table 13 - Queries response time - times in milliseconds per request

5.6.Summary

In this chapter, we described a possible implementation of the proposed architecture. We described how we matched each architectural element to an open source tool, and how we built the network of each element. Finally, we presented the execution times of the sparse matrix building and query executions, using the proposed distributed RDF data store. In the next chapter, we wrap up the proposal and guide the reader to related and future work.