Open-Source Search Engines in the Cloud

Khaled Nagi(✉)

Faculty of Engineering, Department of Computer and Systems Engineering,
Alexandria University, Alexandria, Egypt
khaled.nagi@alexu.edu.eg

Abstract. The key to the success of the analysis of petabytes of textual data available at our fingertips is to do it in the cloud. Today, several extensions exist that bring Lucene, the open-source de facto standard of textual search engine libraries, to the cloud. These extensions come in three main directions: implementing scalable distribution of the indices over the file system, storing them in NoSQL databases, and porting them to inherently distributed ecosystems. In this work, we evaluate the existing efforts in terms of distribution, high availability, fault tolerance, manageability, and high performance. We are committed to using common open-source technology only. So, we restrict our evaluation to publicly available open-source libraries and eventually fix their bugs. For each system under investigation, we build a benchmarking system by indexing the whole Wikipedia content and submitting hundreds of simultaneous search requests. By measuring the performance of both indexing and searching operations, we report of the most favorable constellation of open-source libraries that can be installed in the cloud.

Keywords: Full-text searching · Indexing · Distributed ecosystems · NoSQL · Cloud

1 Introduction

Since the early 2010s, search engines took over the job of performing intelligent textual analytics of petabytes of data. The most prominent open-source projects in this area are Solr [29] and Elasticsearch [14]. While Solr seems to be more commonly used in information retrieval domains, Elasticsearch is gaining more popularity in the area of data analysis due to its data visualization tool Kibana [9]. Both have Lucene [19] at their heart. Lucene is the de-facto standard search engine library. It is based on research dating back to 1990 [5].

The first attempt to scale Lucene beyond the classical file system is presented in [20]. Lucene storage classes are extended to store the index to and traverse the index from the relational database management systems, such as MySQL. Back then, the storage API was not standardized and undergone many changes. After the wide-spread of NoSQL database management systems and distributed BigData systems, such as Hadoop, the mean-time standardized Lucene storage API is ported to these emerging technologies in several open-source prototypes. A good overview of these efforts are found in [11].

A. Fred et al. (Eds.): IC3K 2015, CCIS 631, pp. 100–117, 2016.
DOI: 10.1007/978-3-319-52758-1_7

In this work, we investigate these open-source implementations as we believe that the key to the success of any large-scale search engine will remain *openness*. We explicitly refrain from adding any customized implementation to the off-the-shelf open-source components. *In the case of the presence of bugs in these publicly available systems, we attempt to solve them*. Other than fixing bugs, we apply only the tweaks supplied by the official performance tuning recommendations from the providers.

Our contribution is the independent evaluation of the existing approaches in terms of support for distribution. The main focus is the evaluation of the performance of both indexing and searching of these systems. Moreover, a robust distributed search engine must support *data partitioning*, *replication* and must be always consistent. So, we investigate the effect of node failures. Furthermore, we take into consideration the ease of management of the cluster.

The rest of the paper is organized as follows. In Sect. 2, we describe the properties of a distributed highly-scalable search engine. We also give a background of the technologies. In Sect. 3, we describe the architecture of each system under investigation. In Sect. 4, we present the performance evaluation based on the benchmarking scenario that we constructed. Section 5 concludes the paper.

2 Background and Related Work

The following properties must be available in any cloud-based large-scale search engine:

- **Partitioning (sharding):** It is splitting the index into several independent sections, usually called *shards*. Each shard is a separate index and is usually indexed independently. Depending on the sharding strategy, a search query is directed to the corresponding shard(s) and each result is merged and returned to the user.
- **Replication:** It means storing various copies of the same data to increase the data availability. On a system built over commodity hardware, such as Amazon EC2 [1] or Microsoft Azure [3], replication protects against the loss of data. Additionally, replication is also used to increase the throughput of index reads.
- **Consistency:** Depending on a relaxed definition of consistency, a newly indexed document is not necessarily made available to the next search request. However, the index data structure must be consistent under whatever storage model used to store it. Consistency between the internal blocks of a single index must be guaranteed all the time, whereas consistency across the independent shards is not a must.
- **Fault-Tolerance:** It means the absence of any Single Point of Failure (SPoF) in the system. In case of the failure of a node, the whole search engine should not be go offline.
- **Manageability:** The administration of the nodes of a cloud-based search engine must be made easily: either through a Command Line Interface (CLI), programmatically embeddable interface, e.g., JMX, or most preferably via web administration consoles.
- **High Performance:** It means that the response time for query processing should be under a couple of seconds under a full load of concurrent search requests. Indexing of new documents should be done in parallel.

2.1 Lucene-Based Search Engines

A full text search index is usually a variation of the inverted index structure [5]. *Indexing* begins with collecting the available set of documents by the crawler. A crawler consists in general of several hundreds of data gathering threads. The parser in these threads converts the collected documents to a stream of plain text. In the analysis phase, the stream of data is tokenized according to predefined delimiters, such as blank, tab, hyphen, etc. Then, all stop words are removed from the tokens. A stem analyzer usually reduces the tokens to their roots to enable phonetic searches. *Searching* begins with parsing the user query using the same parser used in the indexing process. This is a must or else the matching documents will not be exactly the ones needed. The tokens have to be analyzed by the same analyzer used for indexing as well. Then, the index is traversed for possible matches. The fuzzy query processor is responsible for defining the match criteria and the score of the hit according to a calculated distance vector.

Lucene [19] is at the heart of almost every full-text search engine. It provides several useful features, such as ranked searching, fielded searching and sorting. Searching is done through several query types including: phrase queries, wildcard queries, proximity queries, range queries. It allows for simultaneous indexing and searching by implementing a simple pessimistic locking algorithm [17].

An important internal feature of Lucene is that it uses a configurable storage engine. It comes with a codec to store the index on the disc or maintain it in-memory for smaller indices. The internal structure of the index file is public and is platform independent [16]. This ensures its portability. Back in 2007, this concept was used to store the index efficiently into Relational Database Management Systems [20]. The same technique is used today to store the index in other NoSQL databases, such as Cassandra [15] and mongoDB [23].

Apache Solr [29] is built on-top of Lucene. It is a web application that can be deployed in any servlet container. It adds the following functionality to Lucene:

- XML/HTTP/JSON APIs,
- Hit highlighting,
- Faceted search and filtering,
- Range queries,
- Geospatial search,
- Caching,
- Near Real-Time (NRT) searching of newly indexed documents, and
- A web administration interface.

SolrCloud [29] was released in 2012. It allows for both sharding and replication of the Lucene indexes. The management of this distribution is seamlessly integrated into an intuitive web administration console. Figure 1 illustrates the configuration of one of our setups in the web administration console.

Elasticsearch [14] evolved almost in parallel to Solr and SolrCloud. Both bring the same set of features. Both are very performant. Both are open-source and use a different combination of open-source libraries. At their hearts, both have Lucene. In general, Solr seems to be slightly more popular than Elasticsearch in information retrieval domains; whereas Elasticsearch is expanding more in the direction of data analytics.

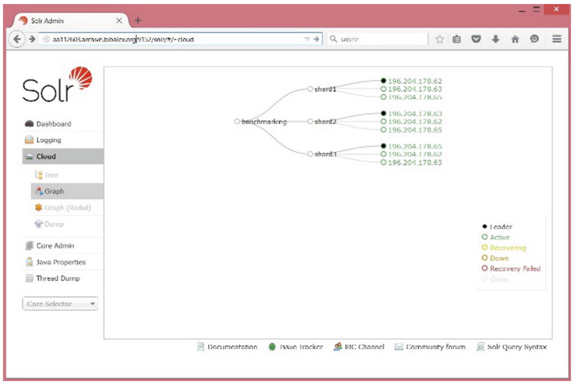

Fig. 1. Screenshot of the web administration console.

2.2 NoSQL Databases

The main strength of NoSQL databases comes from their ability to manage extremely large volumes of data. For this type of applications, ACID transaction properties are too restrictive. More relaxed models emerged such as the CAP theory or eventually consistent emerged [4]. It means that any large-scale distributed DBMS can guarantee for two of three aspects: *Consistency*, *Availability*, and *Partition* tolerance. In order to solve the conflicts of the CAP theory, the BASE consistency model (BAsically, Soft state, Eventually consistent) is defined for modern applications [4]. This principle goes well with information retrieval systems, where intelligent searching is more important than consistent ones.

A good overview of existing NoSQL database management systems can be found in [8]. Mainly, NoSQL database systems fall into four categories:

- graph databases,
- key-value systems,
- column-family systems, and
- document stores.

Graph databases concentrate on providing new algorithms for storing and processing very large and distributed graphs. They are often faster for associative data sets. They can scale more naturally to large data sets as they do not require expensive join operations. Neo4j [21] is a typical example of a graph databases.

Key-value systems use associative arrays (maps) as their fundamental data structure. More complicated data structures are often implemented on top of the maps. Redis [25] is a good example of a basic key-value systems.

The data model of *column-family* systems provides a structured key-value store where columns are added only to specified keys. Different keys can have different number of columns in any given family. A prominent member of the column family stores is Cassandra [15]. Apache Cassandra is a second generation of distributed key value stores; developed at Facebook. It is designed to handle very large amounts of data spread across many commodity servers without a single point of failure. Replication is done even across multiple data centers. Nodes can be added to cluster without downtime.

Document-oriented databases are also a subclass of key-value stores. The difference lies in the way the data is processed. A document-oriented system relies on internal structure in the document order to extract metadata that the data-base engine uses for further optimization. Document databases are schema-less and store all related information together. Documents are addressed in the database via a unique key. Typically, the database constructs an index on the key and all kinds of metadata. mongoDB [23], first developed in 2007, is considered to be the most popular NoSQL nowadays [6]. mongoDB provides high availability with replica sets.

In all attempts to store Lucene index files in NoSQL databases, the contributors take the logical index file as starting point. The set of logical files are broken into logical blocks that are stored in the database. It is therefore clear that plain key-value data stores and graph databases are not suitable for storing a Lucene index. On the other hand, document stores, such as mongoDB, are ideal stores for Lucene indices. One Lucene logical file maps easily to a mongoDB document. Similarly, the Lucene logical directory (files) is mapped to a Cassandra column family (rows), which is captured using an inherited implementation of the abstract Lucene `Directory` class. The files of the directory are broken down into blocks (whose sizes are capped). Each block is stored as the value of a column in the corresponding row.

2.3 Highly Distributed Ecosystems

After the release of [7], Doug Cutting worked on a Java-based MapReduce implementation to solve scalability issues on Nutch [13]; which is an open-source web crawler software project to feed the indexer of the search engine with textual content. This was the base for the Hadoop open source project; which became a top-level Apache Foundation project. Currently, the main Hadoop project includes four modules:

- Hadoop Common: It supports the other Hadoop modules.
- Hadoop Distributed File System (HDFS): A distributed file system.
- Hadoop YARN: A job scheduler and cluster resource management.
- Hadoop MapReduce: A YARN-based system for parallel processing of large data sets.

Each Hadoop task (Map or Reduce) works on the small subset of the data it has been assigned so that the load is spread across the cluster. The map tasks generally load, parse, transform, and filter data. Each reduce task is responsible for handling a subset of the map task output. Intermediate data is then copied from mapper tasks by the reducer tasks in order to group and aggregate the data. *It is definitely appealing to use the MapReduce framework in order to construct the Lucene index using several nodes of a Hadoop cluster*.

The input to a MapReduce job is a set of files that are spread over the Hadoop Distributed File System (HDFS). In the end of the MapReduce operations, the data is written back to HDFS. HDFS is a distributed, scalable, and portable file system. A Hadoop cluster has one *namenode* and a set of *datanodes*. Each datanode serves up blocks of data over the network using a block protocol. HDFS achieves reliability by replicating the data across multiple hosts. Hadoop recommends a replication factor of 3. Since the release of Hadoop 2.0 in 2012, several high-availability capabilities, such as providing automatic failover of the namenode, are implemented. This way, HDFS comes with no single point of failure. HDFS was designed for mostly immutable files [22] and may not be suitable for systems requiring concurrent write-operations. Since the default storage codec for Solr is append-only, it matches HDFS. With the extreme scalability, robustness and widespread of Hadoop clusters, it offers the perfect store for Solr in Cloud-based environments.

Additionally, there are *three* ecosystems that can be used in building distributed search engines: Katta, Blur and Storm.

Katta [12] brings Apache Hadoop and Solr together. It brings search across a completely distributed MapReduce-based cluster. Katta is an open-source project that uses the underlying Hadoop HDFS for storing the indices and providing access to them. Unfortunately, the development of Katta has been stopped. The main reason is the inclusion of several of the Katta features within the SolrCloud project.

Apache *Blur* [2] is a distributed search engine that can work with Apache Hadoop. It is different from the traditional big data systems in that it provides a relational data model-like storage on top of HDFS. Apache Blur does not use Apache Solr; however, it consumes Apache Lucene APIs. Blur provides data indexing using MapReduce and advanced search features; such as a faceted search, fuzzy, pagination, and a wildcard search. Blur shard server is responsible for managing shards. For Synchronization, it uses Apache ZooKeeper [32]. Blur is still in the apache incubator status. The current release version 0.2.3 works with Hadoop 1.x and is not validated using the scalability features coming with Hadoop 2.x.

The third project *Storm* [30] is also in its incubator state at Apache. Storm is a real time distributed computation framework. It processes huge data in real time. Apache Storm processes massive streams of data in a distributed manner. So, it would be a perfect candidate to build Lucene indices over large repositories of documents once it is reaches the release state. Apache Storm uses the concept of Spout and Bolts. Spouts are data inputs; this is where data arrives in the Storm cluster. Bolts process the streams that get piped into it. They can be fed data from spouts or other bolts. The bolts can form a chain of processing, with each bolt performing a unit task in a concept similar to MapReduce.

3 Systems Under Investigation

3.1 Solr on Cassandra

Solandra is an open-source project that uses Cassandra, the column-based NoSQL database, instead of the file system for storing indices in the Lucene index format [16]. The project is very stable. Unfortunately, the last commit dates back to 2010. The current Solandra version available for download uses Apache Solr 3.4 and Cassandra 0.8.6. Solandra does not use SolrCloud since it was not present at the time of the development of the open-source project. The Cassandra-based distributed data storage is implemented behind the Façade `CassandraDirectory`. Solandra uses its own index reader called `SolandraIndexReaderFactory` by overriding the default index reader.

In the Solandra project, Solr and Cassandra run both within the same JVM. However, with change in the configuration and the source code, we run a Cassandra cluster instead of the single database. In a small implementation, the Cassandra cluster spreads over 3 nodes as illustrated in Fig. 2. The larger cluster contains 7 nodes. On Cassandra, each node exchanges information across the cluster every second. This value can be change in its configuration file to match the hardware requirement. A sequentially written commit log on each node captures write activity to ensure data durability. Data is then indexed and written to an in-memory structure. Once the memory structure is full, the data is written to disk in the `SSTable` data file. All writes are automatically partitioned and replicated throughout the cluster. A cluster is arranged as a *ring* of nodes. Clients send read/write requests to any node in the ring; that takes on the role of coordinator node, and forwards the request to the node responsible for servicing it. A *partitioner* decides which nodes store which rows.

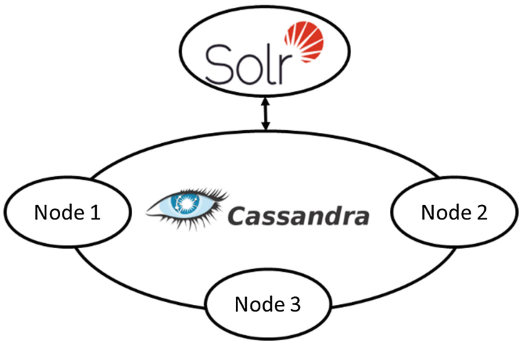

Fig. 2. Our Solandra installation.

Both sharding and replication are automatically made available by the Casandra cluster. Cassandra also guarantees the consistency of the blocks read by its various nodes. Although fault-tolerance is a strong feature of Cassandra, Solr itself is the single point of failure in this implementation, due to the absence of the integration with SolrCloud. Unfortunately, Solandra does not support the administration console of Solr. The only management option is through the Cassandra CLI.

3.2 Lucene on mongoDB

Another open-source NoSQL-based project is LuMongo [18]. LuMongo is a simple JAVA library that implements Lucene APIs and stores the index in mongoDB. All data, including indices and documents, is stored in mongoDB. mongoDB supports sharding and replication out of the box. LuMongo itself operates as another independent cluster. On error, clients can fail to another cluster node. The exchange of the cluster management information is done through an in-memory database called Hazelcast. Nodes in the cluster can be added and removed dynamically through a simple CLI command. Using the CLI, the user can perform the following operations:

- query the health status of cluster,
- list available indices, get their counts,
- submit simple queries, and
- fetch documents.

For example, the following CLI command registers a node in the cluster:

```
bash clusteradmin.sh --command registerNode --mongoConfig
mongo.properties --nodeConfig node.properties
      --address 10.0.0.10
```

The following command starts a node:

```
bash startnode.sh --mongoConfig mongo.properties --
address 10.0.0.10 --hazelcastPort 5702
```

LuMongo indices are broken down into shards called *segments*. Each segment is an independent Lucene standard index. A hash of the document's unique identifier determines which segment a document's indexed fields will be stored into. In our smaller implementation, illustrated in Fig. 3, the segments are stored in a 3 × 3 mongoDB cluster for the small setup and 7 shards and 3 replicas for the larger setup to match the number of LuMongo servers; which is 3 and 7 respectively.

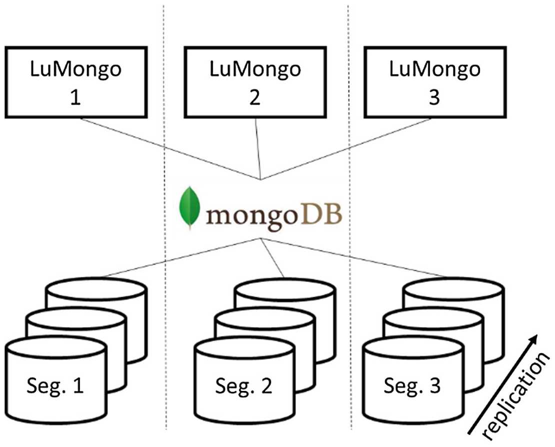

Fig. 3. Our LuMongo implementation.

In this setup, sharding is implemented in both LuMongo and mongoDB. The mongoDB takes care of partitioning seamlessly. mongoDB guarantees the consistency of the index store, while LuMongo guarantees the consistency of the search result. There is *no* single point of failure in mongoDB and LuMongo.

While running our experiments under heavy load of concurrent search requests, *we discover a memory leak within the LuMongo code*. The problem causes the LuMongo node to crash at approx. 60 concurrent search per node. On the positive side, the whole distributed system does not fail. Load is distributed evenly among the available nodes. However, the cost is degrading the performance. We track down the problem to be in fetching the content of the documents after returning the document ids from the search engine. Consequently, we fix the problem and deploy a patched LuMongo to our experiments to eliminate this malefaction.

3.3 Solrcloud

SolrCloud [29] contains a cluster of Solr nodes. Each node runs one or more collections. A collection holds one or more shards. Each shard can be replicated among the nodes. Apache ZooKeeper [32] is responsible for maintaining coordination among various nodes, similar to Hazelcast in the LuMongo project. It provides load-balancing and failover to the Solr cluster. Synchronization of status information of the nodes is done in-memory for speed and is persisted on the disk at fixed checkpoints. Additionally, the ZooKeeper maintains configuration information of the index; such as schema information and Solr configuration parameters. Together, they build a Zookeeper *ensemble*. When the cluster is started, one of the Zookeeper nodes is elected as a *leader*. Although distributed in reality, all Solr nodes retrieve their configuration parameters in a central manner through the Zookeeper *ensemble*. Usually, there are more than one Zookeeper for redundancy. All Zookeeper IPs are stored in a configuration file that is given to each Solr node at startup.

The following commands start ZooKeeper and Solr nodes respectively:

```
./bin/zkServer.sh start conf/zoo.cfg

./bin/solr restart -cloud -z
196.204.178.62:2181,196.204.178.63:2181,196.204.178.65:21
81 -p 9120 -s example/cloud/node2/solr -m 5g
```

SolrCloud distributes search across multiple shards transparently. The request gets executed on all leaders of every shard involved. Search is possible with near-real time (NRT); i.e., after a document is committed. Figure 4 illustrates our small cluster implementation. We build the cluster using a Zookeeper ensemble consisting of 3 nodes. We install 3 SolrCloud instances on three different machines, define 3 shards and replicate them 3 times. In the larger cluster, we extend the Zookeeper ensemble to spread 7 machines. We use 7 SolrCloud instance to master 7 shards while keeping the replication factor at 3.

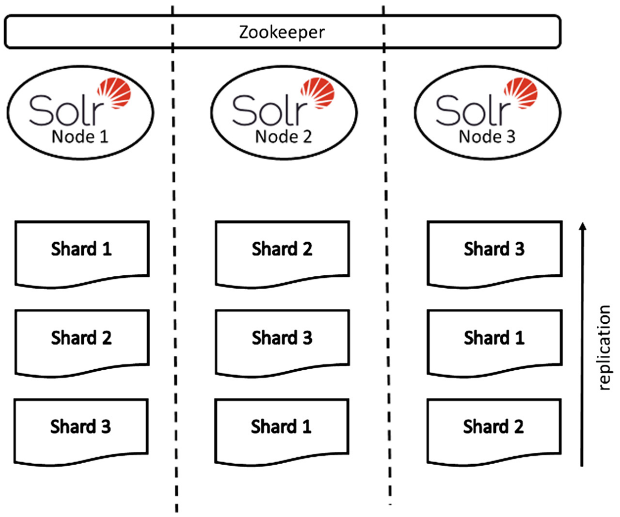

Fig. 4. Our SolrCloud implementation.

3.4 Solrcloud on Hadoop

Building SolrCloud on Hadoop is an extension to the implementation described in Sect. 3.3. The same Zookeeper ensemble and SolrCloud instances are used. Solr is then configured to read and write indices in the HDFS of Hadoop by implementing an `HdfsDirectoryFactory` and implementing a lock type based on HDFS. Both the directory factory and the lock implementation come with the current stable version of Solr [26]. The following command starts SolrCloud with Hadoop as its storage backend.

```
./bin/solr start -cloud -z 196.204.178.62:2181,
196.204.178.63:2181, 196.204.178.65:2181 -p 9152 -m 5g
-Dsolr.directoryFactory=HdfsDirectoryFactory
-Dsolr.lock.type=hdfs
-Dsolr.hdfs.home=hdfs://196.204.178.66:9161/user/nagi/62
```

Figure 5 illustrates our small cluster implementation. We leave replication to the HDFS. We set the replication factor on HDFS to 3 to be consistent with the rest of the setups. For the small cluster, we also use a 3 node Hadoop installation. For the large cluster, we use a 7 node cluster.

Solr provides indexing using MapReduce in two ways. In the first way, the indexing is done at the map side [27]. Each Apache Hadoop mapper transforms the input records into a set of (key, value) pairs, which then get transformed into `SolrInputDocument`. The Mapper task then creates an index from `SolrInputDocument`. The Reducer performs de-duplication of different indices and merges them if needed. In the second way, the indices are generated in the reduce

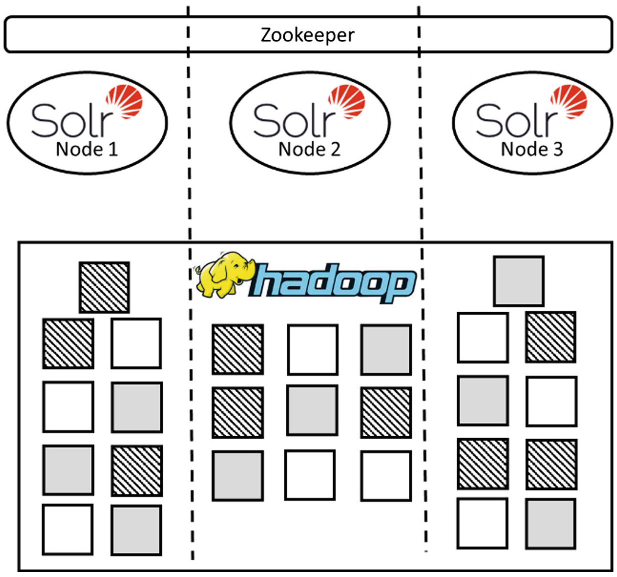

Fig. 5. Our SolrCloud implementation over Hadoop.

phase [28]. Once the indices are created using either ways, they can be loaded by SolrCloud from HDFS and used in searching. We use the first way and employ 20 nodes in the indexing process.

3.5 Functional Comparison

Table 1 summarizes the functional differences between all 4 systems under investigation.

Table 1. Functional comparison of the systems under investigation.

	Solr on Cassandra	Lucene on mongoDB	SolrCloud	SolrCloud on Hadoop
Sharding	Done by Cassandra	Done by mongoDB	Done by Solr	Done by Solr
Replication	Done by Cassandra	Done by mongoDB	Sync. on the level of the file system under to coordination of Zookeeper	Done by HDFS
Consistency	Guaranteed by Cassandra	Guaranteed by LuMungo and mongonDB	Done by Solr and managed by Zookeeper	Guaranteed by HDFS, Solr and Zookeeper
Fault-tolerance	Solr is SPoF	No SPoF	No SPoF	No SPoF
Manageability	CLI	CLI	Web	Web for Solr + web for Hadoop

4 Benchmarking

We build a full text search engine of the English Wikipedia [31] to evaluate the performance of the system described in the previous Section. The index is built over 49 GB of textual content. We develop a benchmarking platform on top of each search engine under investigation as illustrated in Fig. 6.

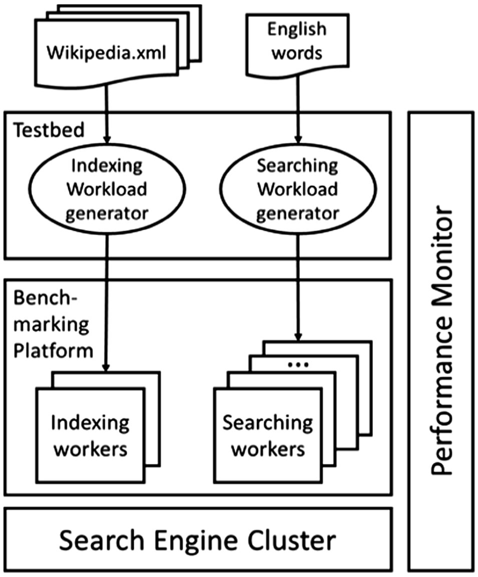

Fig. 6. Components of the benchmarking platform.

The *searching workload generator* composes queries of single terms, which are randomly extracted from a long list of common *English words*. It submits them in parallel to the application. The *indexing workload generator* parses the *Wikipedia dump* and sends the page title, the timestamp, and most important the *content* to the benchmarking platform workers. They pass them to the search engine cluster to be indexed, thus simulating a web crawler. The benchmarking platform manages two connection pools of worker threads. The first pool consists of several hundreds of *searching workers* threads that process the search queries coming from the searching workload generator. The second pool consists of *indexing workers* threads that process the updated content coming from the indexing workload generator. Both worker types submit their requests over http to the search engine cluster under investigation. The performance of the system including that of the search engine cluster is monitored using the *performance monitor* unit.

4.1 Input Parameters and Performance Metrics

We choose the maximum number of fetched hits to be 50. This is a realistic assumption taking into consideration that no more than 25 hits are usually displayed on a web page.

We choose to read the content of these 50 hits and not only the title while fetching the resultset. This exaggerated implementation is intended to artificially stress test the search engines clusters under investigation. The number of search threads is varied from 32 to 320 to match the size of connection pool for the searching worker threads. By relaxing the requirement of prefetching all the pages of the resultset, the number of concurrent searching threads can be increased enormously. In case of high load, the workload generator distributes its searching search threads over 4 physical machines to avoid throttling the requests by the hosting client. Due to locking restrictions inherent in Lucene, we restrict our experiments to maximum one indexing worker per node in the search engine cluster.

In all our experiments, we monitor the response time of the search operations from the moment of submitting the request till receiving the overall result. We also monitor the system throughput in terms of:

- searches per second, and
- index inserts per hour.

Additionally, the performance monitor constantly monitors CPU and memory usages of the machines running the search engine cluster.

4.2 System Configuration

In order to neutralize the effect of using virtualized nodes in globalized data cloud centers, we conduct our experiments in an isolated cluster available at the Internet Archive of the Bibliotheca Alexandrina [10]. The Bibliotheca Alexandrina possesses a huge dedicated computer center for archiving the Internet, digitizing material at Bibliotheca Alexandrina and other digital collections.

The Internet Archive at the Bibliotheca Alexandrina has about 35 racks each rack is comprised of 30 to 40 nodes and a gigabit switch connecting them. The 35 racks are connected also with a gigabit switch. The nodes are based on commodity servers with a total capacity of 7000 TB.

The Bibliotheca Alexandrina dedicated one rack with 20 nodes to our research for approx. one month. The nodes are connected with a gigabit switch and are isolated from the activities of the Internet Archive during the period of our experiments. Each node has an Intel i5 CPU 2.6 GHz, 8 GB RAM, 4 SATA hard disks 3 TB each.

For each search engine cluster, we construct a small version and a larger one as described in Sect. 3. The small cluster consists of three nodes each containing a shard (a portion of the index) while the larger one is built over 7 nodes. In all installations have a replication factor of 3.

4.3 Indexing

Indexing speed varies largely with the number of nodes involved in the index building operation. Lucene; and hence Solr; employs a pessimistic locking mechanism while inserting data into the index. This locking mechanism is being kept for all backend implementations. From our current experiments and from previous ones [20], we conclude that there is no benefit in having more than one indexing thread per Lucene index (or Solr shard).

This means that the increase in number of shards and their dedicated indexing Lucene/Solr yields to a proportional increase in the speed of indexing. The increase is also linear for all systems under investigation. In other words, the indexing speed of a 3 nodes cluster is 3 times that's of a cluster consisting of a single node. Respectively, the indexing speed of a 7 nodes cluster is 2.3 times that's of a cluster consisting of 3 nodes. A clear winner in this contest is SolrCloud on Hadoop that employs MapReduce in indexing. Using all 20 nodes available in the MapReduce operation increases the speed by factor of 18. A minimum overhead is wasted later on in merging the indices into 3 and 7 nodes, respectively.

In order to normalize a comparison between all systems, we plot the throughput of using one indexing thread on a 3 shards, 3 replica cluster in Fig. 7. These numbers are roughly multiplied by the number of nodes involved to get the overall indexing speed.

On the normalized scale, NoSQL backends bring very different results. Casandra has by far the fastest rate of insertion (60% faster than SolrCloud). This experiment confirms the results reported by [24] proving the high throughput of Cassandra as compared to other NoSQL databases. On the other hand, mongoDB-based storage is the slowest. SolrCloud brings very good results on the file system. The overhead of storage on HDFS is about 26% which is very acceptable taking into consideration the advantages of storing data on Hadoop clusters in cloud environments and the huge speed-ups due to the use of MapReduce in indexing.

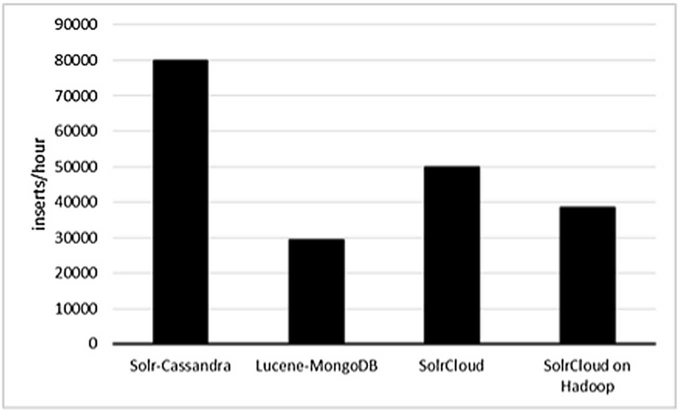

Fig. 7. Normalized indexing speed.

4.4 Searching

Searching is more important than indexing. We repeat the search experiments with the number of search threads varying from 32 to 320. The duration of each experiment is set to 15 min to eliminate any transient effect.

The set of experiments is repeated for both the small cluster and the large cluster. The response time for the small cluster is illustrated in Fig. 8 and the large cluster in Fig. 9. The throughput in terms of number of searches per second versus the number of searching threads is plotted in Fig. 10 for the small cluster and in Fig. 11 for the larger one.

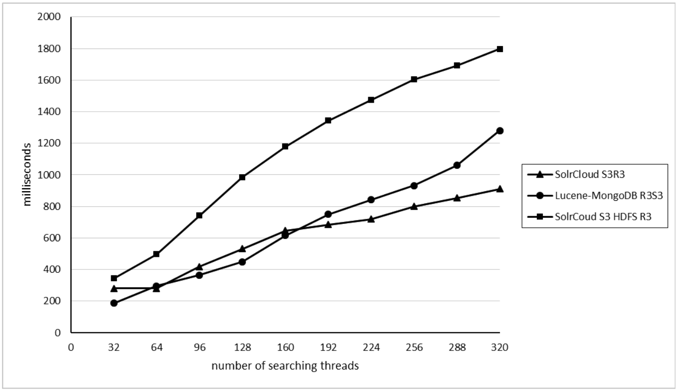

Fig. 8. Search time on the small cluster.

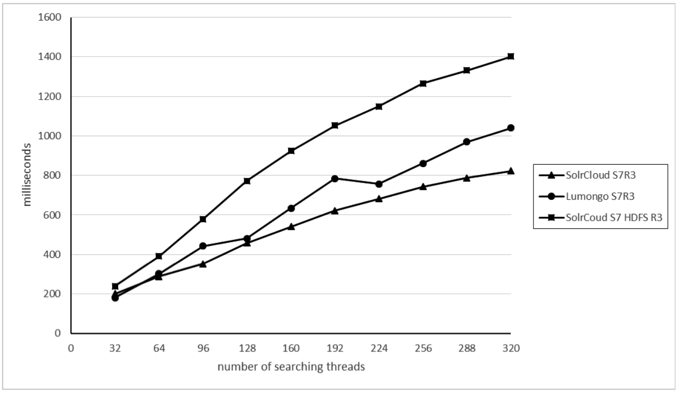

Fig. 9. Search time on the large cluster.

The bad news is that the response time of the single Solr on the Cassandra cluster is far higher than the other systems (>10 s). So, we dropped plotting its values for both clusters. The same applies to the throughput, which was much lower than its counterparts (<50 searches/second). Again this matches the findings in [24], where the high throughput of Cassandra comes at the cost of read latency.

The good news is that the response time for the other systems is very much below the usual 3 s threshold tolerated by a searching user. The maximum search time measured on the small cluster is below 1.8 s and 1.4 s for the larger cluster. The curves

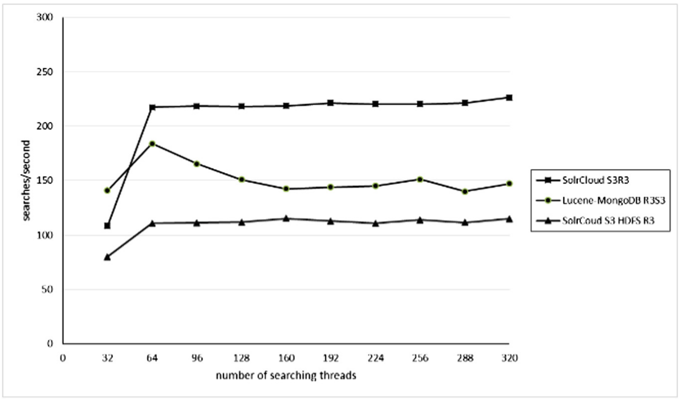

Fig. 10. Throughput of the small cluster.

also show that the response time of the larger cluster is better than the smaller cluster under all settings. This means that the performance of the system is enhanced by the increase of the number of nodes. The system did not achieve its saturation yet.

The figures also illustrate the impact of HDFS on the response time and the overall throughput of the search. Although the search time is increased by almost 40% and the throughput is almost halved, the absolute values remain far below the user threshold of 3 s by retrieving the hits and the contents of each hit for a result-set size of 50 in less than 2 s.

Another important remark is that the performance of all systems degrade gracefully including LuMongo *after fixing the memory leak problem found in the original implementation*.

The throughput curves, Figs. 10 and 11, illustrate that the throughput saturates after a certain number of concurrent search threads. In the small cluster, Fig. 10, the three setups saturate at 64 concurrent threads. On the large cluster, Fig. 11, this number increases to 128.

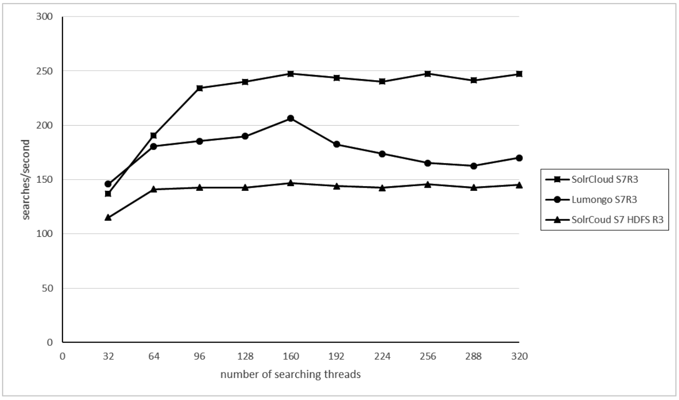

Fig. 11. Throughput of the large cluster.

5 Conclusion

In this paper, we investigate the available options for building large-scale search engines that we deploy in a private Cloud. We restrict ourselves to open-source libraries and do not add extra implementation other that publicly available. Nevertheless, we allow ourselves to fix bugs that we encounter in the available code. We investigate each variation, in terms of scalability through data partitioning, redundancy through replication, consistency, and the ease of management. We build a benchmarking platform on top of the systems under investigation. For each variation, we construct a small and a large cluster. The results of the experiments show that the combination of Solr and Hadoop provide the best tradeoff in terms of scalability, stability and manageability. Search engines based on NoSQL databases offer either a superior indexing speed, or fast searching times, which seem to be mutually exclusive in our settings.

Acknowledgements. We Would like to Thank the Bibliotheca Alexandrina for Providing Us with the Necessary Hardware for Conducting the Benchmarking Experiments.

References

1. Akioka, S., Muraoka, Y.: HPC Benchmarks on Amazon EC2. In: IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (2010)
2. Blur (Incubating) Home. https://incubator.apache.org/blur/. Accessed Jan (2016)
3. Bojanova, I., Samba, A.: Analysis of cloud computing delivery architecture models. In: IEEE Workshops of International Conference on Advanced Information Networking and Applications (2011)
4. Brewer, E.: Towards robust distributed systems. In: ACM Symposium on Principles of Distributed Computing (2000)
5. Cutting, D., Pedersen, J.: Optimizations for dynamic inverted index maintenance. In: SIGIR 1990 (1990)
6. DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems. http://db-engines.com/en/ranking. Accessed Jan (2016)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. Edlich, S., Friedland, A., Hampe, J., Brauer, B.: NoSQL: Introduction to the World of Non-relational Web 2.0 Databases (In German) NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. Hanser Verlag, Munich (2010)
9. Gupta, Y.: Kibana Essentials. Packt Publishing, Birmingham (2015)
10. Internet Archive at Bibliotheca Alexandrina. http://www.bibalex.org/en/project/details?documetid=283. Accessed Jan (2016)
11. Karambelkar, H.V.: Scaling Big Data with Hadoop and Solr, 2nd edn. Packt Publishing, Birmingham (2015)
12. Katta. http://katta.sourceforge.net/. Accessed Jan (2016)
13. Khare, R., et al.: Nutch: a flexible and scalable open-source web search engine. Technical report. Oregon State University, pp. 32–32 (2004)

14. Kuc, R., Rogozinski, M.: Mastering Elasticsearch, 2nd edn. Packt Publishing, Birmingham (2015)
15. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
16. Lucene - Index File Formats. https://lucene.apache.org/core/3_0_3/fileformats.html. Accessed Jan (2016)
17. Lucene – LockFactory. http://lucene.apache.org/core/4_8_0/core/org/apache/lucene/store/LockFactory.html. Accessed Jan (2016)
18. LuMongo Realtime Time Distributed Search. http://lumongo.org/. Accessed Jan (2016)
19. McCandless, M., Hatcher, E., Gospodnetiæ, O.: Lucene in Action, 2nd edn. Manning, Greenwich (2010)
20. Nagi, K.: Bringing information retrieval back to database management systems. In: International Conference on Information and Knowledge Engineering, IKE 2007 (2007)
21. Neo4j. http://www.neo4j.org. Accessed Jan (2016)
22. Pessach, Y.: Distributed Storage: Concepts, Algorithms, and Implementations. CreateSpace Independent Publishing Platform (2013)
23. Plugge, E., Hawkins, D., Membrey, P.: The Definitive Guide to mongoDB: The NoSQL Database for Cloud and Desktop Computing. Apress, Berkeley (2010)
24. Rabl, T., et al.: Solving big data challenges for enterprise application performance management. VLDB Endow. **5**(12), 1724–1735 (2012)
25. Redis. http://redis.io/. Accessed Jan (2016)
26. Solr - Apache Lucene - The Apache Software Foundation! http://lucene.apache.org/solr/. Accessed Jan (2016)
27. Solr-1045, Build Solr index using Hadoop MapReduce. https://issues.apache.org/jira/browse/SOLR-1045. Accessed Jan (2016)
28. Solr-1301, Add a Solr contrib that allows for building Solr indices via Hadoop's Map-Reduce. https://issues.apache.org/jira/browse/SOLR-1301. Accessed Jan (2016)
29. Smiley, D., Pugh, E., Parisa, K., Mitchell, M.: Apache Solr Enterprise Search Server, 3rd edn. Packt Publishing, Birmingham (2015)
30. Storm - The Apache Software Foundation. https://storm.apache.org/. Accessed Jan (2016)
31. Wikipedia article dump. https://dumps.wikimedia.org/enwiki/. Accessed Jan (2016)
32. Zookeeper. https://zookeeper.apache.org/. Accessed Jan (2016)