Bringing Search Engines to the Cloud using Open Source Components

Khaled Nagi
Dept. of Computer and Systems Engineering, Faculty of Engineering, Alexandria University, Alexandria, Egypt

Keywords: Search Engine, Scalability, Fault Tolerance, Open-Source, Lucene, Solr, NoSQL, Hadoop.

Abstract: The usage of search engines is nowadays extended to do intelligent analytics of petabytes of data. With Lucene being at the heart of the vast majority of information retrieval systems, several attempts are made to bring it to the cloud in order to scale to big data. Efforts include implementing scalable distribution of the search indices over the file system, storing them in NoSQL databases, and porting them to inherently distributed ecosystems, such as Hadoop. We evaluate the existing efforts in terms of distribution, high availability, fault tolerance, manageability, and high performance. We believe that the key to supporting search indexing capabilities for big data can only be achieved through the use of common open-source technology to be deployed on standard cloud platforms such as Amazon EC2, Microsoft Azure, etc. For each approach, we build a benchmarking system by indexing the whole Wikipedia content and submitting hundreds of simultaneous search requests. We measure the performance of both indexing and searching operations. We stimulate node failures and monitor the recoverability of the system. We show that a system built on top of Solr and Hadoop has the best stability and manageability; while systems based on NoSQL databases present an attractive alternative in terms of performance.

1 INTRODUCTION

Since Doug Cutting originally wrote Lucene (McCandless et al., 2010) in 1999 after a long series of scientific publications dating back to 1990 (Cutting and Pedersen, 1990), it has emerged as the standard full text search engine in the open-source community. Several other open-source projects, such as Solr (Smiley et al., 2015) and Elasticsearch (Kuc and Rogozinski, 2015), are built on top of Lucene and offer extended search facilities, such as faceted navigation, hit highlighting, auto-suggest, Geospatial search.

Now, search engines are required to do intelligent analytics of petabytes of data. Back in 2007, the first attempts (Nagi, 2007) were made to provide scalable, robust and distributed search engines by porting the core of Lucene storage classes to run on relation database management systems. With the emergence of NoSQL database management systems and inherently distributed ecosystems, such as Hadoop, many open-source prototypes and implementations attempt nowadays to support the necessary features for any large-scale cloud-based implementation of a search engines (Karambelkar, 2015).

In this work, we investigate the most prominent publicly available implementations. We believe that the key to the success of any large-scale search engine will remain the same as the success of the original Lucene, which is *openness*. In our Work, we *explicitly* refrain from adding any customized implementation to the *off-the-shelf* open-source components. 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 - in which data partitioning and replication while maintaining consistency - play a major role. We always investigate the effect of node failures, since almost all popular and modern cloud providers nowadays, such as Amazon EC2 (Akioka and Muraoka, 2010) and Microsoft Azure (Bojanova and Samba, 2011), are built on commodity hardware. Furthermore, we take into consideration the ease of management of the cluster. However, our main focus is the evaluation of the performance of both indexing and searching of these systems.

The rest of the paper is organized as follows. In Section 2, the features desired in a distributed highly-scalable search engines together with a brief background of the technologies in use are presented. Section 3 brings a detailed description of the systems under investigation. In Section 4, the performance evaluation is presented. Section 5 concludes the paper and presents an outlook to our future work in this area.

2 BACKGROUND AND RELATED WORK

The following features are the key to the success of any cloud-based large-scale search engine:

- **Partitioning (Sharding):** It is splitting the index into several independent sections. Each section can be viewed as a separate index and is indexed independently. A query is answered by processing it at the shards in question before the result is consolidated and returned to the user.
- **Replication:** It provides redundancy and increases data availability. With multiple copies of data on different servers, replication protects an index from the loss of a single node. In some cases, replication can be used to increase read capacity.
- **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. Taking a deeper look into the structure of Lucene (“Lucene - Index File Formats”, n.d.), for example; the content of one internal block is dependent on the content of another. Consistency between these blocks must be guaranteed all times, whereas consistency across the independent shards is not a must.
- **Fault-tolerance:** it means the absence of any Single Point of Failure (SPoF). Most modern clouds are based on commodity hardware. The temporary absence of a node is expected to occur at any point of time. This should never lead to the failure of the whole search engine.
- **Manageability:** A cloud-based search engine is spread across several dozens of servers. The administration of these servers and the services deployed on them 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:** Cloud-based search engines should be capable of indexing the shards in parallel. They should also process hundreds of search queries in parallel with a reasonable response time (e.g., under 3 seconds).

2.1 Lucene-based Search Engines

A full text search index is an efficient cross-reference lookup data structure. Usually, a variation of the well-known inverted index structure is used (Cutting and Pedersen, 1990).

The *indexing* process begins with collecting the available set of documents by the data gatherer. The parser converts them to a stream of plain text. In the analysis phase, the stream of data is tokenized according to predefined delimiters and a number of operations are performed on the tokens, e.g., the removal of all stop words and the reduction of the words to their roots to enable phonetic searches.

The *searching* process begins with parsing the user query. The tokens have to be analyzed by the same analyzer used for indexing. 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.

Lucene (McCandless et al., 2010) 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 (“Lucene - Class LockFactory”, n.d.).

An important internal feature of Lucene is that it uses a configurable storage engine. In its standard release, 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 (“Lucene - Index File Formats”, n.d.). This ensures its portability. Back in 2007, this concept was used to store the index efficiently into Relational Database Management Systems (Nagi, 2007). The same technique is used today to store the index in other NoSQL databases, such as Cassandra (Lakshman and Malik, 2010) and mongoDB (Plugge et al., 2010).

Apache Solr (Smiley et al., 2015) 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 and JSON APIs
- Hit highlighting
- Faceted search and filtering
- Geospatial search
- Caching

- Near real-time searching of newly indexed documents.
- Web administration interface

SolrCloud (Smiley et al., 2015) was released in 2012. It is an extension to Solr that allows for both *sharding* and *replication*. The management of this distribution is seamlessly integrated into an intuitive web administration console. Figure 1 illustrates the configuration of one our setups in the web administration console.

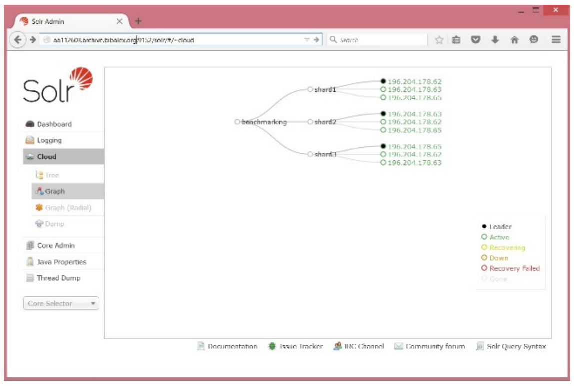

Figure 1: Screenshot of the web administration console.

Elasticsearch (Kuc and Rogozinski, 2015) 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; whereas Elasticsearch is expanding more in the direction of data analytics.

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 (Brewer, 2000). 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 (Brewer, 2000). 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 (Edlich, et al, 2010). 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 (“neo4j”, n.d.) 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 (“Redix”, n.d.) 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 (Lakshman and Malik, 2010). 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 database engine uses for further optimization. Document databases are schemaless 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 (Plugge et al., 2010), first developed in 2007, is considered to be the most popular NoSQL nowadays (“DB-Engines”, n.d.). 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 Inherently Distributed Ecosystems

After the release of (Dean and Ghemawat, 2008), Doug Cutting worked on a Java-based MapReduce implementation to solve scalability issues on Nutch (Khare et al., 2004); which is an open-source web crawler software project to feed search engines with 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 these 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 fail-over of the namenode, are implemented. This way, HDFS comes with no single point of failure. HDFS was designed for mostly immutable files (Pessach, 2013) 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 wide-spread 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 (“Katta”, n.d.) 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* (“Blur”, n.d.) 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 (“ZooKeeper”, n.d.). 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* (“Storm”, n.d.) 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 instead of the operating system file system for storing indices in the Lucene index format ("Lucene - Index File Formats", n.d.). 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. That's why any installation would use Solr and not SolrCloud. The details of the Cassandra-based distributed data storage is completely hidden behind the `CassandraDirectory` class and its associated classes. Solandra uses its own index reader called `SolandraIndexReaderFactory` by overriding the default index reader.

Under Solandra, Solr and Cassandra run both within the same JVM. However, with a slight reconfiguration, we run a Cassandra cluster instead. In a small implementation, the Cassandra cluster spreads over 3 nodes and 7 nodes in the larger one as illustrated in Figure 2.

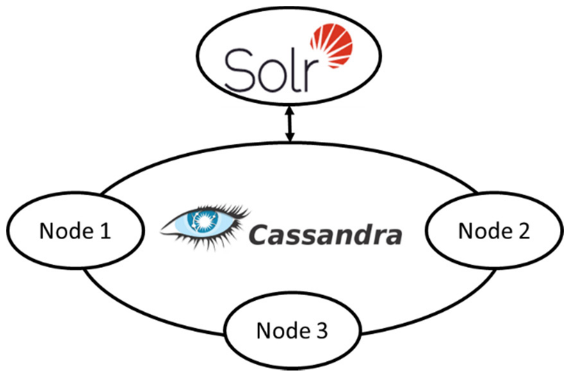

Figure 2: Our Solandra installation.

On Cassandra, each node exchanges information across the cluster every second. 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 an 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.

This way, both sharding and replication are automatically made available by Casandra. 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 ("LuMongo", n.d.). LuMongo provides the flexibility and power of Lucene queries with the scalability and ease of use of mongoDB. All data in LuMongo is stored in mongoDB including indices and documents. Inherently mongoDB can be sharded and replicated. LuMongo itself operates as a cluster. On error, clients can fail to another cluster node. Nodes in the cluster can be added and removed dynamically through a simple CLI command. The CLI offers to query the health status of cluster, list available indices, get their counts, submit simple queries, and fetch documents.

LuMongo indices are broken down into shards called *segments*. Each segment is an independent 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 Figure 3, the segments are stored in a 3x3 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.

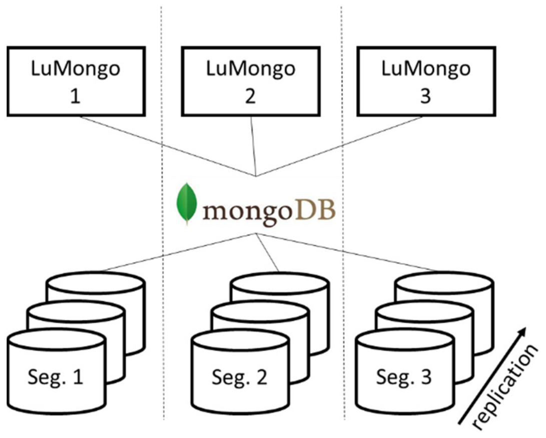

Figure 3: Our LuMongo implementation.

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.

3.3 SolrCloud

SolrCloud (Smiley et al., 2015) 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 ("ZooKeeper", n.d.) is responsible for maintaining co-ordination among various nodes. 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. Usually, there are more than one Zookeeper for redundancy. Together, they build a Zookeeper *ensemble*. When the cluster is started, one of the Zookeeper nodes is elected as a *leader*. The same occurs for Solr. There is a leader responsible for each shard.

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; 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.

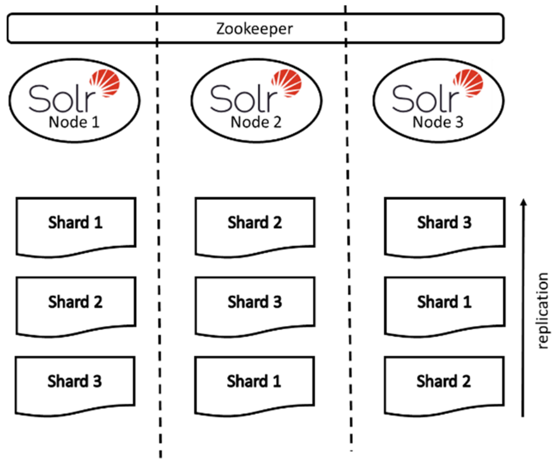

Figure 4: Our SolrCloud Implementation.

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.

3.4 SolrCloud on Hadoop

Building SolrCloud on Hadoop is an extension to the implementation described in Section 3.3. The same Zookeeper ensemble and SolrCloud instances are used. Solr is then configured to read and write indices in the HDFS by implementing an `HdfsDirectoryFactory` and implementing a lock type based on HDFS. Both come with the current stable version of Solr ("Solr", n.d.), version 5.2.1. 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.

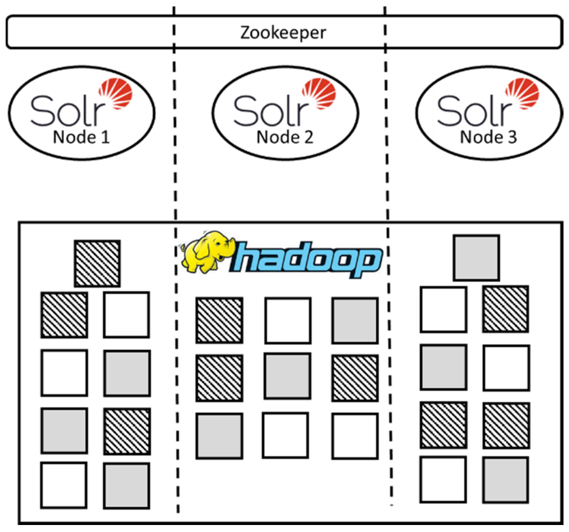

Figure 5: Our SolrCloud implementation over Hadoop.

Solr provides indexing using MapReduce in two ways. In the first way, the indexing is done at the map side ("Solr-1045", n.d.). 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 phase ("Solr-1301", n.d.). 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 Lu-Mungo 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

In our order to evaluate the performance of the various search engine clusters under investigation, we build a full text search engine of the English Wikipedia ("Wikipedia-dumps", n.d.). 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 Figure 6.

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 content, and other attributes such as timestamp and revision numbers to the benchmarking platform workers, which in turn pass them to the search engine cluster be indexed. 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 index *inserting 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.

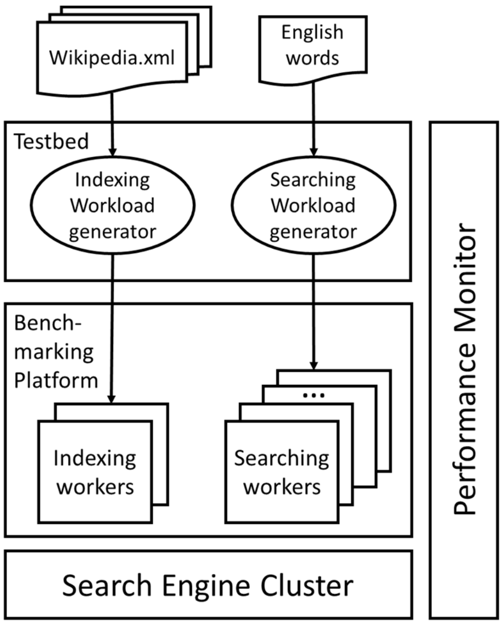

Figure 6: Components of the benchmarking platform.

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 result-set. 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. 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; such as Amazon EC2 or Microsoft Azure, we conduct our experiments in an isolated cluster available at the Internet Archive of the Bibliotheca Alexandrina ("Internet Archive BA", n.d.). 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 Section 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 (Nagi, 2007), 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 Figure 7. These numbers are roughly multiplied by the number of nodes involved to get the overall indexing speed.

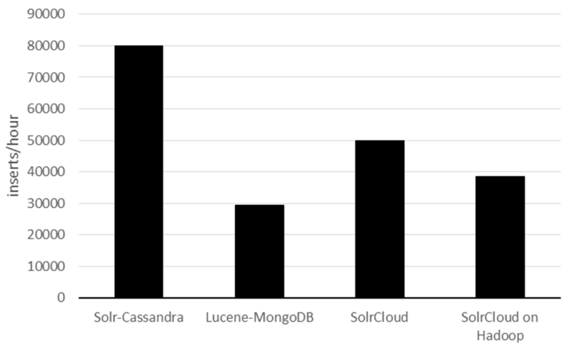

Figure 7: Normalized indexing speed.

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 (Rabl et al., 2012) 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.

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 minutes 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 Figure 8 and the large cluster in Figure 9. The throughput in terms of number of searches per second versus the number of searching threads is plotted in Figure 10 for the small cluster and in Figure 11 for the larger one.

The bad news is that the response time of the single Solr on the Cassandra cluster is far higher than the other systems (>10 seconds). So, we dropped plotting its values for both clusters. The same applies to the throughput, which was much lower than its counterparts (< 50 searhes/second). Again this matches the findings in (Rabl et al., 2012), 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 seconds threshold tolerated by a searching user. The maximum search time measured on the small cluster is below 1.8 seconds and 1.4 seconds for the larger cluster. The curves 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 seconds by retrieving the hits *and* the contents of each hit for a result-set size of 50 in less than 2 seconds.

Another important remark is that the performance of all systems degrade gracefully with the increase of workload except for LuMongo. Under heavy workloads, (192 for the small cluster and 288 for the large cluster) LuMongo runs out of heap memory. We track down the problem to be in fetching the content of the documents after returning the document ids from the search engine. There is a small memory leakage in LuMongo that causes the abortion of the searches under heavy loads. Havingthis solved in future releases on LuMongo, Lu

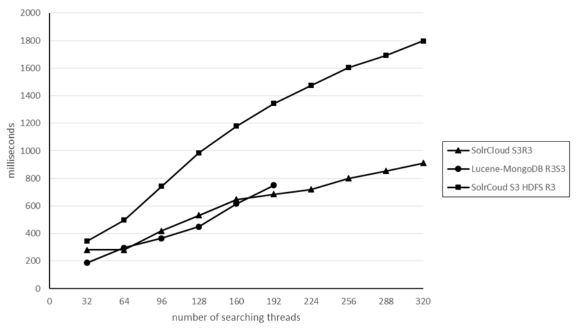

Figure 8: Search time on the small cluster.

Mongo will be a very important choice regarding its superior response time illustrated in Figure 8 and Figure 9.

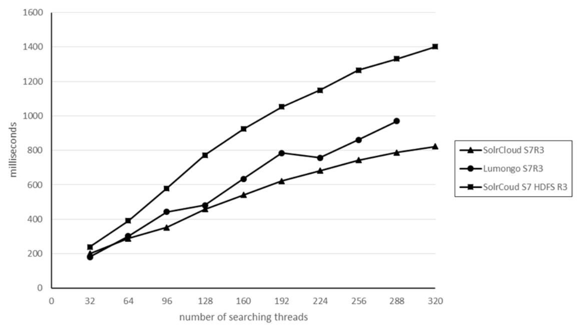

Figure 9: Search time on the large cluster.

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

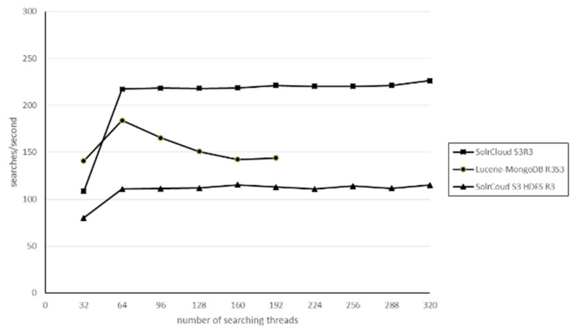

Figure 10: Throughput of the small cluster.

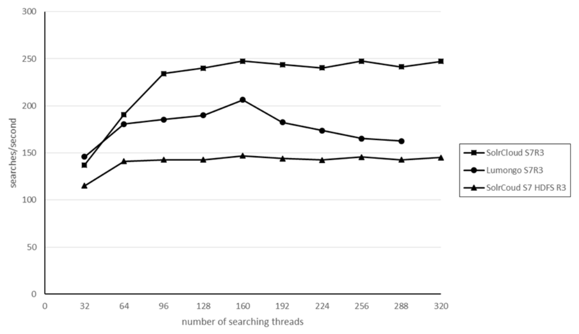

Figure 11: Throughput of the large cluster.

5 CONCLUSION AND FUTURE WORK

In this paper, we investigate the available options for building large-scale search engines that are capable of running in the Cloud. We restrict ourselves to open-source libraries, including Lucene, Solr, mongoDB, Cassandra, and Hadoop. We explicitly do not add extra implementation other that publicly available components. We investigate each variation, in terms of scalability through data partitioning, redundancy through replication, consistency either through the NoSQL databases or through open-source synchronization libraries, such as Zookeeper. The ease of management of the multi-node cluster is also an important issue in our evaluation. Performance plays a major part in our analysis. We build a benchmarking platform on top of the systems under investigation. For each variation, we construct a small and a large cluster. In our experiments, we measure both the speed of indexing as well as the search time and the throughput of the searching threads. The results of the experiments show that 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. Unfortunately, they suffer from stability in their integration implementations.

In the future, we plan to contribute to LuMongo by fixing its memory leakage problem. A good contribution would also be the extension of Solandra to support SolrCloud instead of a single Solr instance. Having done this, the owner of the large-scale search engine would have the choice between either using the Hadoop infrastructure or a NoSQL cluster installation depending on availability in his/her environment and his/her knowledge.

ACKNOWLEDGEMENTS

We would like to thank the Bibliotheca Alexandrina for providing us with the necessary hard-ware for conducting the benchmarking experiments.

REFERENCES

Akioka, S. and Muraoka, Y., 2010. HPC Benchmarks on Amazon EC2, *Proceedings of the IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA).*

Bojanova, I. and Samba, A., 2011. Analysis of Cloud Computing Delivery Architecture Models, *IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA).*

Blur, n.d., Apache Blur (Incubating) Home, *https://incubator.apache.org/blur/*, retrieved July 2015.

Brewer, E., 2000. Towards Robust Distributed Systems. *ACM Symposium on Principles of Distributed Computing.*

Cutting, D. and Pedersen, J., 1990. Optimizations for Dynamic Inverted Index Maintenance, *Proceedings of SIGIR '90.*

DB-Engines, n.d., Knowledge Base of Relational and NoSQL Database Management Systems, *http://db-engines.com/en/ranking*, retrieved July 2015.

Dean, J. and Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM. 51, 1, 107–113.*

Edlich, S., Friedland, A., Hampe, J., Brauer, B., 2010. *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.

Internet Archive BA, n.d., Internet Archive at Bibliotheca Alexandrina, *http://www.bibalex.org/en/project/details?documentid=283*, retrieved July 2015.

Karambelkar, H.V., 2015. *Scaling Big Data with Hadoop and Solr*, Packt Publishing, 2nd Edition.

Katta, n.d., *http://katta.sourceforge.net/*, retrieved July 2015.

Khare, R. et al., 2004: Nutch: A flexible and scalable open-source web search engine. *Technical Report Oregon State University. 1, 32–32.*

Kuc, R. and Rogozinski, M., 2015. *Mastering Elasticsearch,* Packt Publishing, 2nd Edition.

Lakshman, A. and Malik, P., 2010. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40.

Lucene - Index File Formats, n.d. *https://lucene.apache.org/core/3_0_3/fileformats.html*, retrieved July 2015.

Lucene - Class LockFactory, n.d., *http://lucene.apache.org/core/4_8_0/core/org/apache/lucene/store/LockFactory.html*, retrieved July 2015.

LuMongo, n.d., LuMongo Realtime Time Distributed Search, *http://lumongo.org/*, retrieved July 2015.

McCandless, M., Hatcher, E., and Gospodnetić, O., 2010. *Lucene in Action*, Manning, 2nd Edition.

Nagi, K., 2007. Bringing Information Retrieval Back To Database Management Systems, *Proceedings of IKE'07, International Conference on Information and Knowledge Engineering.*

Neo4j, n.d., *http://www.neo4j.org*, retrieved July 2015.

Pessach, Y., 2013. *Distributed Storage: Concepts, Algorithms, and Implementations*, CreateSpace Independent Publishing Platform.

Plugge, E., Hawkins, D., and Membrey, P., 2010. *The Definitive Guide to mongoDB: The NoSQL Database for Cloud and Desktop Computing*, Apress.

Rabl, T. et al., 2012. Solving big data challenges for enterprise application performance management, *Proceedings of the VLDB Endowment, Volume 5 Issue 12, pp 1724-1735.*

Redix, n.d., *http://redis.io/*, retrieved July 2015.

Solr, n.d., Solr - Apache Lucene - The Apache Software

Foundation! http://lucene.apache.org/solr/, retrieved July 2015.

Solr-1045, n.d., Build Solr index using Hadoop MapReduce, *https://issues.apache.org/jira/browse/SOLR-1045*, retrieved July 2015.

Solr-1301, n.d., Add a Solr contrib that allows for building Solr indices via Hadoop's Map-Reduce., *https://issues.apache.org/jira/browse/SOLR-1301*, retrieved July 2015.

Smiley, D., Pugh, E., Parisa, K., Mitchell, and Apache M., 2015. *Solr Enterprise Search Server*, Packt Publishing, 3rd Edition.

Storm, n.d., Storm - The Apache Software Foundation, *https://storm.apache.org/*, retrieved July 2015.

Wikipedia-dumps, n.d., Wikipedia article dump, *https://dumps.wikimedia.org/enwiki/*, retrieved July 2015.

ZooKeeper, n.d., Apache Zookeeper, *https://zookeeper.apache.org/*, retrieved July 2015.