Advanced Databases Project : Search engines and Sphinx

Rafaele Antonio , Ramos Perez Nestor Eduardo, and Sefu Kevin

Master in computer science

December 18, 2018

Contents

0.1 Introduction

The last ten years, as shown in Figure 1, have been strongly marked by an exponential growth in the amount of digital data available on cyberspace.

The quantitative explosion of digital data forces new ways of seeing and analyzing the world. New orders of magnitude concern capturing, storing, searching, sharing, analyzing and visualizing data.

The explosion of stored data is not without consequence in our daily lives. Indeed, the way of managing these data influences many area of our lives such as the way we stay in touch with our friends, the way we find customers for our business and even the way we meet our future spouse.

Nowadays, we are living in a quickly changing world. The volume of available information and the connection bandwidth that gives us access to that information grows substantially every year. For instance twenty years ago a 1-million-row database of geographical locations was something mind-blowing but today it is something that can be quickly fetch off from the Internet by anyone.
This rapid transformation of our world has transformed searching from something used only by specific categories of people into something that each of us has to deal with in his daily life. Searching has been transformed from high-end, optional feature to an essential functionality that absolutely has to be provided to end users.

This growth in the amount of data can be easily explained by the recent growth of data generated by applications. Facing this situation, developers and researchers have been pushed to design technologies capable to manage effectively data.

In order to present those technologies and to present searching, we are going to present in the following lines search engines and particularly Sphinx.

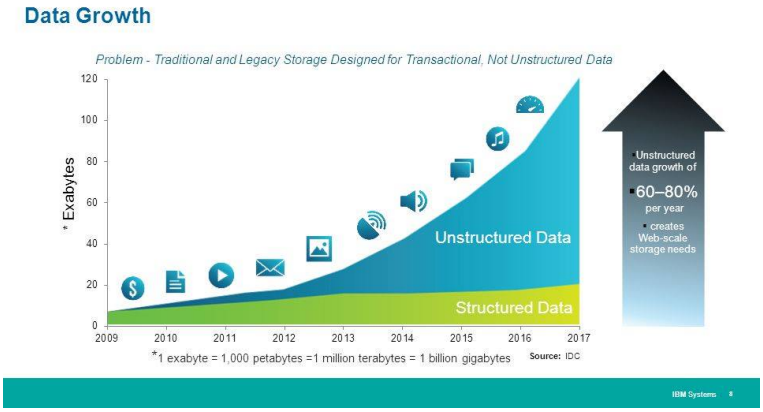

Figure 1: Growth of data since 2009

0.2 Background : Search

0.2.1 What is a search algorithm ?

A search algorithm is the step-by-step procedure used to locate specific data among a collection of data. It is considered a fundamental procedure in computing.
In general search algorithms are classified according to their mechanism of searching. There are four family of search algorithms :

Linear search algorithms : Algorithms which check every record in order to find the one which is associated with a target key in a linear fashion.

Binary, or half interval searche algorithm : Algorithm which repeatedly target the center of the search structure and divide the search space in half.

Digital search algorithms : For those algorithms searching is based on the properties of digits in data structures that use numerical keys.

Hashing : Algorithm that maps keys to records based on a hash function.

0.2.2 Search algorithms illustrations

This section presents illustrations of how the four family of search algorithms work. We present first linear searching, secondly binary searching, then digital searching and finaly hashed searching.

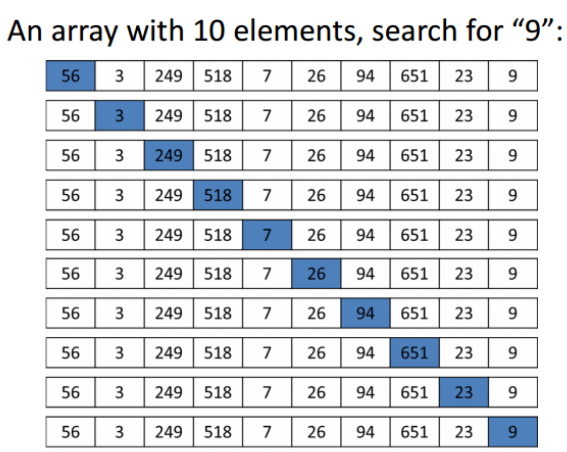

Figure 2: linear search algorithm

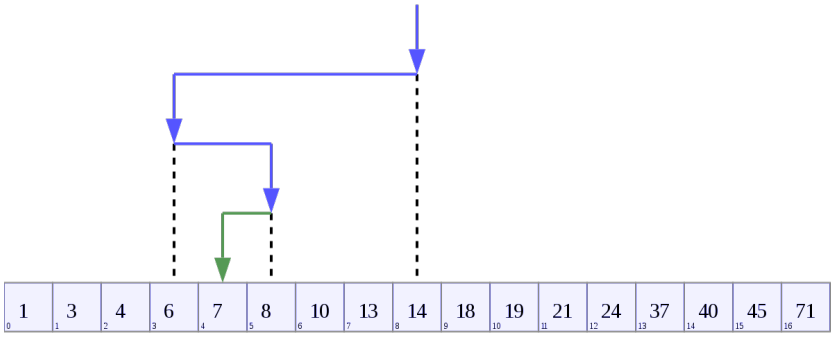

Figure 3: binary search algorithm

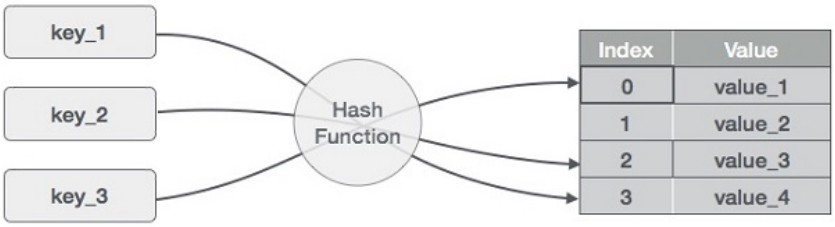

Figure 4: Hashing algorithm

0.2.3 Searching ? The real meaning

Based on the definition of a search algorithm seen in the previous point, Searching can be formally defined as choosing a subset of entries that match given criteria from a complete data set by using a search algorithm.

0.2.4 What is a search engine database ?

A database search engine is a NoSQL database management systems more formally, they can be define as a software program that operates on material stored in a digital database and reports information contains in this database or related to specified terms.

Databases search engines are characterized by :

1. a support for complex search expressions
2. a full text search
3. Stemming (reducing inflected words to their stem)
4. Ranking and grouping of search results

5. Geospatial search

6. Distributed search for high scalability

NoSQl databases are generally used for unstructured data while SQL databases are used for structured data.

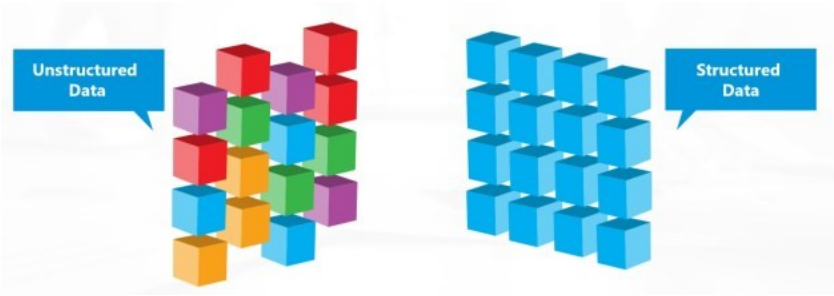

Figure 5: Structured and Unstructured data

0.2.5 Art's state and search engines databases history

To understand the history of search engines database, we have to look at NoSQl database history, for the simple reason that search engines databases are NoSQL database management systems.

Relational database model has been created in 1970 by Edgar Codd. In their initial designing, relational databases has been designed to represent a set of tables containing data fitted into predefined categories. In these systems each tables contains one or more data categories in columns and each row contains a unique instance of data for the categories defined by the columns.
These systems, based on strict application of ACID properties and generally designed to work on single computers, quickly presented limitations especially for big enterprises which was dealing with big amount of data.
An exemple of limitations that faced relational databases was the problem of complexity, with relational databases users had to convert all data into tables but if data didn't fit easily into a table, the databases's structure bacame complex and difficult to work with.

In response to the growing awareness of relational databases limitations, developpers and reasearchers turned to NoSQL databases.
One of the key moments in this shift was in 2007, with the publication of a paper that introduced the amazon's Dynamo distributed NoSQL system. In fact amazon was one of the first major companies that decided to store data in a nonrelational database.
The NoSQL database movement began to really gain momentum in June 2009 with the San Francisco meeting organized by Johan Oskarsson. The purpose of this meeting was to review existing NoSQL technologies and find a name

for these new "open-source, distributed and non-relational" systems. The 2009 meeting in San Francisco is considered today as the inauguration of the NoSQL software developer community.

In fact as shown in the lines above , NoSQL database is a recent movement that grow rapidly and nowadays there exist already 4 popular types of NoSQL Databases namely :

Key-value stores : Every item in the database is stored as an attribute name (or "key") together with its value. Riak, Voldemort, and Redis are the most well-known in this category

Wide-column stores : data are stored together as columns instead of rows and are optimized for queries over large datasets. The most popular are Cassandra and HBase..

Document databases : pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents. MongoDB is the most popular of these databases.

Graph databases : used to store information about networks, such as social connections. Examples are Neo4J and HyperGraphDB..

0.3 Search engines database

This section aimed to give an overview on search engines databases. For this purpose this section will be divided into four sub-sections, first we are going to present a sub family of search engines, namely full-text search engines, secondly we will make a comparaison between traditional relational databases and full-text search engines then we are going to present Solr and Elasticsearch as alternatives to Sphinx and finaly we are going to show the place occupied by sphinx in comparison to Solr and Elasticsearch.

0.3.1 Search engines databases : Full-text search engines

Full-text search engines evolved much later than traditional database engines, as corporations and governments found themselves with more and more unstructured data in electronic format. Those data didn't fit well into the old table-style databases, so the need for unstructured full-text searching was apparent.

Even if search engines and relational databases seem to be different technologies, it is important to underline the fact that search engine technology was inspired heavily by the database world, and many search engines still employ some type of traditional table structures in their underlying architecture.

Full-text indexing phase

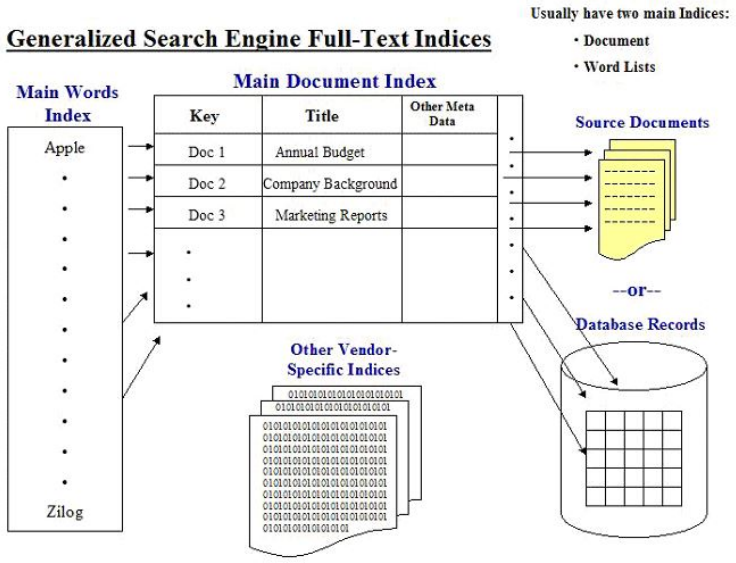

Figure 6: high-level architecture of full-text engines

The Figure-6 explain how the indexing process starts with the insertion by an application of data into a row of the main document index. In the simplest case, the main document index contains one row for each document, and at a minimum contains the name of the external file or document stored in the key field. Additional field values – for example, document title – can be inserted at the same time. If the source of the data resides in a relational database, the primary key in the relational table or view goes in the main document index key field.

Once the data is inserted, the indexing engine opens the external document and creates an ordered word list to load into the main word index. This process is repeated by the engine for every record in the main document index.

While searching is generally available during the indexing process, only completed records are searchable; and for performance questions, most engines batch together a number of records to index more efficiently.

When a user query arrives, either programmatically or as a result of a user request, the full-text engine accesses the sorted and optimized word index to find which documents contain the requested terms. The engine creates a list of documents that qualify, typically provided as a list of pointers into the main document index.

This permits the engine to access and display a result list made up of any fields stored in the main index, calculate a relevance weight, and display a list of results.

0.3.2 Search engines databases vs Relational databases

In order to build the comparison between search engines databases and relational databases, we are going to present first the similaities of those technologies, then the differences and finaly, we will present some advantages of search engines over relational databases.
In the introduction of this section, we presented search engines databases to have similarities with relational databases. Among the similarities we have :

Loading of data : For both technologies, data must be loaded into full-text systems. And like relational systems, the organization of the data and the loading in full-text engines can impact drastically the success of the application. And, as with relational systems, data is loaded into full-text engines both in bulk periodically or on a record-by-record real-time basis.

Query Processing For both relational and full-text engines, a query is processed and passed to the engine to retrieve data. While most relational systems use largely standard syntax, full-text systems generally use proprietary syntax.

Data indexation : With relational databases, once datas are loaded, they should be indexed for optimum performance. The same true for full-text systems.

Although there exists similarities between the two technologies, its possible to find several differences due to fundamental differences between the types of data being indexed and the flexibility retrieval options of those databases. We distinguish as majors differences :

Data structure : Relational engines typically store structured data such that associated datas are stored in the same row, with the components organized into identified fields or columns of information but Full-text engines are optimized for processing formatted and unformatted documents, they also have the capacity to process limited structured data such as document titles, authors, and descriptions.

The synthax Query: There is much wider variety of query language syntax in full-text engines. Many engines do support the classic "AND", "OR" and "NOT" operators in some form; some engines even allow for complex nested queries using parenthesis or other nesting syntax. A modern pseudo-standard is the "Internet syntax", which allows for "+" and "-" to be used as a shorthand for Boolean operators. Many search engines actually support multiple syntaxes that can often be configured when a search is performed.

Weighting : Typically with relational databases when a query is executed, the database engine returns rows that match the specified query in arbitrary order, or sorted by a field specified in the query and records that do not match the query are simply not returned.

Most advanced full-text engines will also retrieve only those documents that match the query, but additionally "weight" each returned document with a relevancy score, so that the results set is not just an unordered list of matching records. Typically, the weight is calculated using proprietary algorithms based on the vendor, although most engines provide syntax for affecting the final weight. Full-text queries typically return a much larger percentage "records" than a traditional query, but frequently only the higher ranking matches are interesting.

Joins operation : The join operation is not provided by Search engines ; instead they often have a much simpler arrangement of data.

Outer Joins operation : Full-text engines doesn't offer a direct analog to the "outer join " operation. Any required complex data gathering tasks would usually be performed prior to indexing the full-text data.

About the advantages that are offered by search engines compared to relational databases we can retain mainly :

Optimization of Textual Data handling: Full-text engines are specifically optimized in the handling of textual data. In particular, proper customer names, city, street names and other geographic markers are all textual in nature. This type of textual data is often subject to multiple valid and invalid spellings.

More Query Operators : Search engines offer many more query operators, such as language stemming, thesaurus, fast wildcard matches, statistically based similarity and word densities, proximity, Soundex.

Granularity of Index Structure : Index structure of full text engines is more granular, allowing for rapid indexed access to specific words and phrases.

0.3.3 Solr and Elasticsearch

In this section, we present Solr and Elasticsearch. Solr and Elasticsearch are two data management technologies. We present them in this section in order to have an overview of existing technologies in the field of data management.

As said in the introduction, nowadays we are overwhelmed by an enormous amount of data. Every kind of text, audio and visual data, which are thought to be transformed into pieces of information, are stored for long periods of time for processing.
The growth of the amount of data is not only associated with the data, but it concerns also technologies that collect, process, store, and analyze the data. Solr and Elasticsearch are tools that propose a solution to the problem of data storage.

Although Solr and Elasticsearch have similar features, there are many parameters that differentiates one from the other such as intended use, type of use, query and indexing performances.

Solr

Solr is a software written in Java, which was started to be developed in 2004. Solr's major features include fulltext search, hit highlighting, faceted search, real-time indexing, dynamic clustering, database integration, NoSQL features. Data in Solr are named as documents. A correlation between Java and Solr can be made as follow :

1. Java Class = Solr Document

2. Java Class Attribute = Document Field

Elasticsearch

Elasticsearch is a open-source tool for real-time, full-text and distributed search, released in 2010 using the Lucene library.
Elasticsearch was designed for Scalability, security, and easy management. By means of its query language that indexes structural and nonstructural, and time-based data, provides rapid search and powerful analysis capabilities.

With Elasticsearch each entry is a structured JSON document. In other words, each data sent to Elasticsearch for indexing is a JSON document.

Solr and Elasticsearch

In this section, we are going to make a technical comparaison between Solr and Elasticsearch. Our comparaison will start first with a presentation of both technologies features, secondly we will present access and data processing and finaly we will show indexing and searching features.

Solr and Elasticsearch general features

The following table present an overview of Solr and Elastcisearch features.

	Solr	Elasticsearch
Initial Release	**2004**	**2010**
Licence	Open Source	Open Source
Database as a Service	No	No
SQL	No	No
Implementation Language	Java	Java
Server OS	All Os with a Java VM	All Os with a Java VM
Based	Apache Lucene	Apache Lucene
Admin Interface	Embedded Available	Plugin needed (Kopf,Marvel etc ...)

Figure 7: Solr and Elasticsearch features

Solr and Elasticsearch data access and processing

	Solr	Elasticsearch
Access Methods	RESful http API, Java API	RESful http API, Java API
Supported client language	.Net, Erlang, Java, JavaScript (XML or JSON), Perl, PHP, Python, RubyScala Clojure	Clojure, Cold Fusion, Erlang, Go, Groovy, Haskell, Java, Javascript, kotlin, .NET, Ocaml, Perl, PHP, Phyton, R, Ruby, Scala, Smalltalk, Vert.x
Data import	JDBC, MySQL, CSV, XML, Tika, URL, Flat File	Wikipedia, MongoDB, CouchDB, RabbitMQ, RSS, Sofa, JDBC, Filesystem, Dropbox, ActiveMSQ, LDAP, Amazaon SQS, St9, OAI, Twitter, DynamoDB, Git, GitHub, Hazelcast, JMS, Kafka, neo4j, Redis, Solr, St9, Subversion

Figure 8: Solr and Elasticsearch data access and processing

Solr and Elasticsearch indexing and searching features

The difference between Solr and Elasticsearch in terms of indexing and searching features comes from their basic structures and the plugins developed by users. The following table present a brief summary of indexing and searching features of both technologies.

	Solr	Elasticsearch
Custom Analyzers and tokenizers	yes	yes
Multiple index	yes	yes
Realtime searching/indexing	yes	yes
Multiple document types per schema	no	yes
Schema changes	yes	yes
Field Copying	yes	yes
Distribued search	after 2012	yes
Term Vectors API	yes	yes
Autocomplete	yes	yes
Highlighting	yes	yes
Structured Query DSL	no	yes

Figure 9: Solr and Elasticsearch indexing and searching features

Solr and Elasticsearch performances

Solr and Elasticsearch are quite similar tools due to the fact that there are Lucene-based. In studies on performance comparisons, it was found out that Solr and Elasticsearch do not greatly have superiority over each other. However, sometimes Solr had better performance over Elasticsearch based on intended use, place of use, and changes in some parameters, and vice versa. In this part of the project, we present some results of several studies made on Solr and Elasticsearch performances.

Study 1 : Search with Indexing

This study aimed to conduct a search on servers with similar features (4 nodes, 4 index shards, no replication, 16 GB per node) with 20 million documents. As shown on the following graphic, we observe that for the same number of search inquieries Elasticsearch queries are finished before Solr ones.

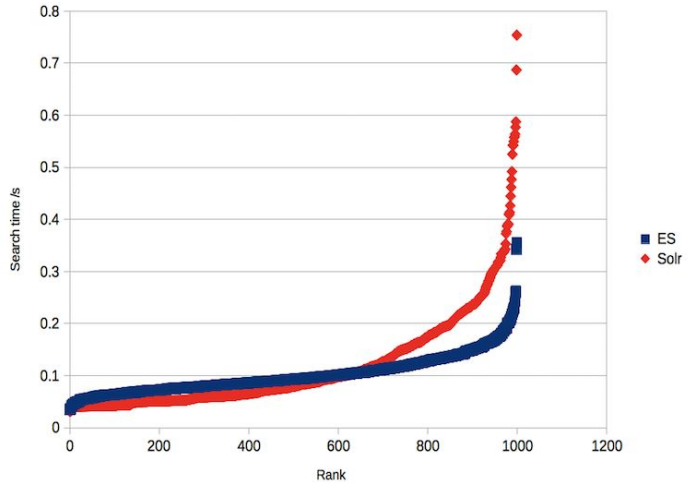

Figure 10: Search with Indexing

Study 2 : Search with Indexing Load In the second study, servers had to make search inquiries while engaged in indexing process. This study was also conducted with 20 million entries on servers with similar features. As shown in the graphic we observe that Elasticsearch completed search inquiries before Solr.

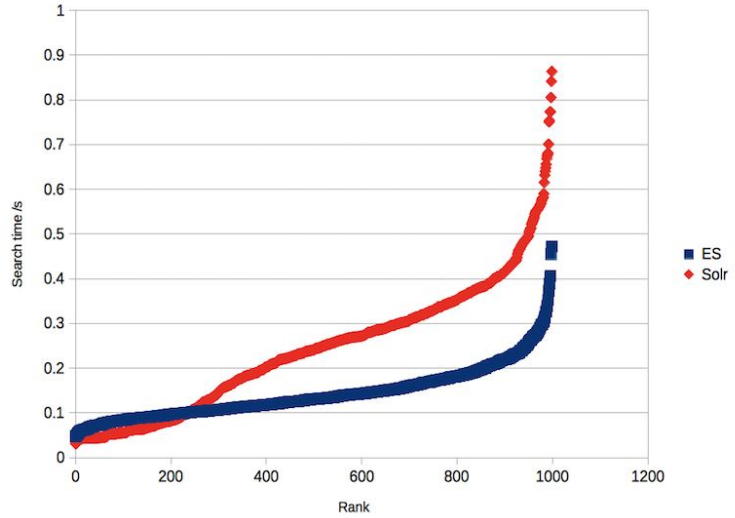

Figure 11: Search with Indexing Load

Study 3 : Query per secondes This study aimed to observe Query per seconds. Query per secondes is a common measure of the amount of search traffic an information retrieval system, such as a search engine or a database, receives during one second. We observe here that Elasticsearch reached 30 QPS speed, while Solr was remained at 15 QPS speed when sent similar search requests.

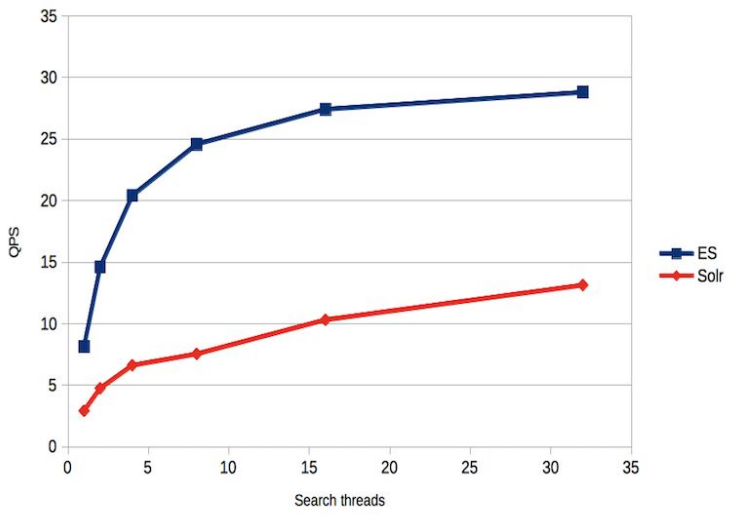

Figure 12: Query per seconds

Trend of Sphinx, Solr and Elasticsearch

In this section, we present the historical trend of Sphinx, Solr and Elasticsearch.
This trend is based on the popularity of those technologies.
The popularity has been measured according to following parameters :

Number of mentions of the system on websites : measured as number of results in search engines queries. (Google, Bing and Yandex).

General interest in the system : The frequency of searches in Google Trends is used for this parameter.

Number of job offers, in which the system is mentioned

Number of profiles in professional networks, in which the system is mentioned. : for this parameters Linked In and Upwork are data source

Relevance in social networks : By counting the number of Twitter tweets, in which the system is mentioned.

The following figure present Sphinx, Slor and Elasticsearch historical trend from July 2014 to July 2018.

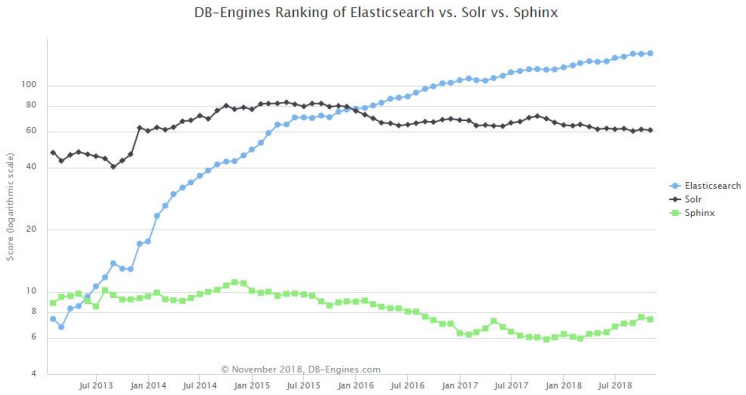

Figure 13: Historical trend of Sphinx, Solr and Elasticsearch,

Regarding to this graphics, sphinx seems to be worse than Solr and Elasticsearch. But we will try to show the opposite in the next section and to show how it can be very interesting to use Sphinx for an application.

0.4 Sphinx

0.4.1 Overview of Sphinx

Sphinx can be define as an external solution for database search, that simply means that sphinx runs outside the main database used for your application. Sphinx takes data from the database application and creates indexes that are stored on a file system. These indexes are highly optimized for searching and the client application uses an API to search the indexes. Sphinx interacts with the database using a data source driver which comes along with Sphinx.
Some major Sphinx features are :

1. High indexing speed (up to 10 MB/sec on modern CPUs)
2. High search speed (average query is under 0.1 sec on 2 to 4 GB of text collection)
3. High scalability (up to 100 GB of text, up to 100 Million documents on a single CPU)
4. Supports distributed searching (since v.0.9.6)
5. Supports MySQL (MyISAM and InnoDB tables are both supported) and PostgreSQL natively

6. Supports phrase searching
7. Supports phrase proximity ranking, providing good relevance
8. Supports English and Russian stemming
9. Supports any number of document fields (weights can be changed on the fly)
10. Supports document groups
11. Supports stopwords, that is, that it indexes only what's most relevant from a given list of words
12. Supports different search modes ("match extended", "match all", "match phrase" and "match any" as of v.0.9.5)
13. Generic XML interface which greatly simplifies custom integration
14. Pure-PHP (that is, NO module compiling and so on) search client API

0.4.2 Installation of Sphinx

This section will present how to install Sphinx on Linux, Windows and Mac OS.

Installation on Linux

1. Download the latest stable version of the sphinx source, available here.

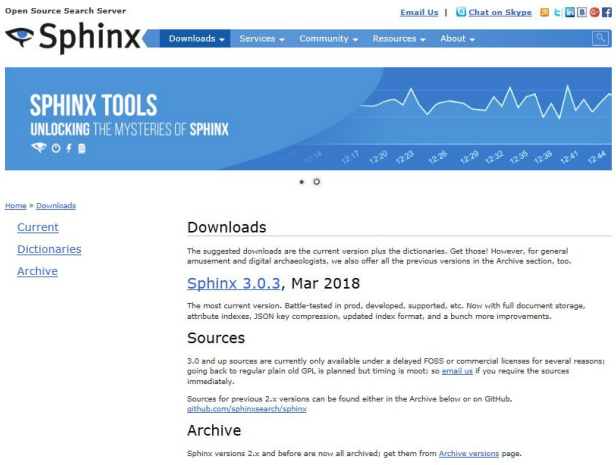

Figure 14: Sphinx downloading page

2. Extract the downloaded file anywhere on your file system and go inside the extracted sphinx directory:

```
$ tar -xzvf sphinx-0.9.9.tar.gz
$ cd sphinx-0.9.9
```

Figure 15: Sphinx installation Linux.

Note that the name of the file differ according to the version of Sphinx you have downloaded.
3. Running of the configure utility :

```
$ ./configure --prefix=/usr/local/sphinx
```

Figure 16: Sphinx installation Linux

4. Building of the source code :

```
$ make
```

Figure 17: Sphinx installation Linux

5. Installation of the application :

```
$ make install
```

Figure 18: Sphinx installation Linux

Installation on Windows

1. Download the Win32 binaries of Sphinx here. Then we have to choose the binary depending on whether we want MySQL support, or PostgreSQL support, or both.

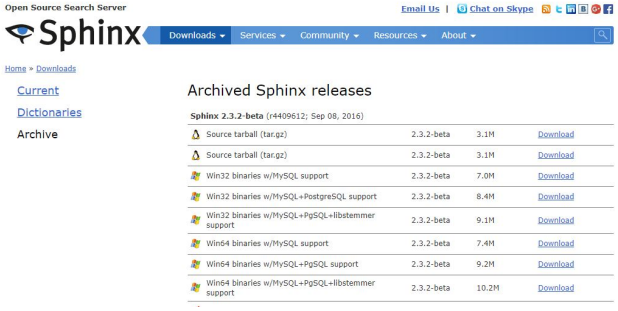

Figure 19: Sphinx installation Windows

2. Extract the downloaded ZIP to any location. In this example we suppose that the zip has been extracted at :
$C: \ sphinx$.

3. Write the following command on your command prompt. This command will install the searched system as a Windows service.

```
C:\sphinx\bin\searchd -install -config C:\sphinx\sphinx.conf -servicename
```

Figure 20: Sphinx installation Windows

```
bin\searchd -install -config C:\sphinx\sphinx.conf -servicename SphinxSearch
```

Figure 21: Sphinx installation Windows

The previous command line, as said before will install the searchd but will not run Sphinx. Before starting the Sphinx service we need to create the sphinx.conf file and create indexes.

installation on Mac OS

1. The first step on Mac OS is similary to the first step of Sphinx installation on Linux.

2. Running of the configuration utility :

```
$ ./configure -prefix=/usr/local/sphinx
```

Figure 22: Sphinx installation Mac OS

3. For 64 bit Mac we have to use the following command to configure:

```
LDFLAGS="-arch x86_64" ./configure --prefix=/usr/local/sphinx
$ make
$ sudo make install
```

Figure 23: Sphinx installation Mac OS

4. Run the make command :

```
$ make
```

Figure 24: Sphinx installation Mac OS

5. Write the following command to finalize the installation

```
$ sudo make install
```

Figure 25: Sphinx installation Mac OS

Checking of the installation

After the installation of Sphinx, if everything went fine then a directory /usr/local/sphinx should have been created on the system and this directory should be strutured like the following image :

```
abbas@abbas:/usr/local/sphinx$ tree
.
|-- bin
|   |-- indexer
|   |-- indextool
|   |-- search
|   |-- searchd
|   `-- spelldump
|-- etc
|   |-- example.sql
|   |-- sphinx.conf.dist
|   `-- sphinx-min.conf.dist
`-- var
    |-- data
    `-- log

5 directories, 8 files
```

Figure 26: Checking of the installation

0.4.3 Sphinx Licence

Sphinx Technologies Inc. is a tiny, private US-based company founded and led by Sphinx author, Andrew Aksyonoff.
Sphinx is a free and open source software which can be distributed or modified under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation, either version 2 or any later version.

However, for project that use Spinx and does not want to disclose the source code as required by GPL. It requires for the project author to obtain a commercial license by contacting Sphinx Technologies Inc. at :
`http://sphinxsearch.com/contacts.html`

0.4.4 How does Sphinx perform searching ?

Sphinx data flow

This section explain how Searching is performed by Sphinx. Its important to know that a complete search solution consists in four key components :

A data source : This represent the soruce of data that will be used by the indexer. Most Sphinx sites use MySQL or another SQL server for storage. But that's not a fundamental requirement—Sphinx, can work just as well with non-SQL data sources.

indexer : This program fetches the data from the data source and creates a full-text index of that data in order to make the search faster.

searchd : This program talks to the (client) program, and uses the full-text index built by indexer to quickly process search queries. It also offer a few other useful functions such as building snippets, splitting a given text into keywords.

Application : This component represent the client application.

In summary the data flow with Sphinx can be represented by the following diagram :

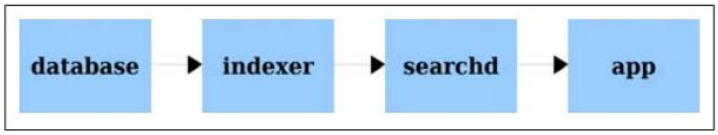

Figure 27: Sphinx data flow

This diagram means that data travel from the storage to the indexer, which build the index and passes the index to searchd program and the searchd program discuss with the client application.

Interaction of database, Sphinx and application

In order to explain how the interaction between those component is done lets consider the following diagram :

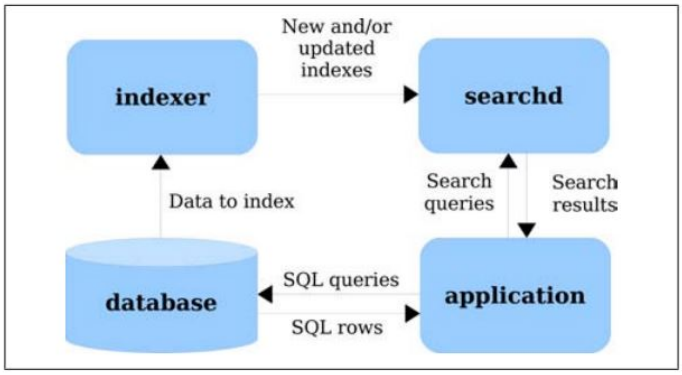

Figure 28: Components interaction

The idea express by this diagram is the following :
Searchd is a server that run countinously. This server talks with the client application and answers to search queries in real time just as a relational database answers data queries.
The indexer is the program that will periodically pulls the data from the database, builds indexes and pass them to searchd server.

0.4.5 How to use Sphinx ?

In order to explain how to use Sphinx, we will present an example of the utilisation because it seems to use easier. For this example, we will :

1. Create a database (data source of our example) named test;

```
CREATE DATABASE test;
```

Figure 29: Create database command

2. We create a table test and fill it with arbitrary values;

```
DROP TABLE IF EXISTS test.documents;
CREATE TABLE test.documents
(
id INTEGER PRIMARY KEY NOT NULL AUTO_INCREMENT,
group_id INTEGER NOT NULL,
group_id2 INTEGER NOT NULL,
date_added DATETIME NOT NULL,
title VARCHAR(255) NOT NULL,
content TEXT NOT NULL
);
REPLACE INTO test.documents ( id, group_id, group_id2, date_added, title, content ) VA
( 1, 1, 5, NOW(), 'test one', 'this is my test document number one. also checking sear
( 2, 1, 6, NOW(), 'test two', 'this is my test document number two' ),
( 3, 2, 7, NOW(), 'another doc', 'this is another group' ),
( 4, 2, 8, NOW(), 'doc number four', 'this is to test groups' );
DROP TABLE IF EXISTS test.tags;
CREATE TABLE test.tags
(
docid INTEGER NOT NULL,
tagid INTEGER NOT NULL,
UNIQUE(docid,tagid)
);
INSERT INTO test.tags VALUES
(1,1), (1,3), (1,5), (1,7),
(2,6), (2,4), (2,2),
(3,15),
(4,7), (4,40);
```

Figure 30: Creation of the table

3. Now that we have create and fill the table, we are going to edit the config fill of Sphinx. You can find this file on your Sphinx directory in the etc folder.

4. Edit the config file.

```
source src1
{
type = mysql
sql_host = localhost
sql_user = test
sql_pass =
sql_db = test
sql_port = 3306 # optional, default is 3306
sql_query = \
SELECT id, group_id, UNIX_TIMESTAMP(date_added)
AS date_added, title, content \
FROM documents
sql_attr_uint = group_id
sql_attr_timestamp = date_added
sql_query_info = SELECT * FROM documents WHERE id=$id
}
```

Figure 31: Editing of the config file.

Note :
The value of the host, user, password and database name must be fill according to user system

5. Once the configuration is end, we can run the indexer by using the following command :

```
$ /usr/local/sphinx/bin/indexer --all
```

Figure 32: Running of the index.

6. The previous command should give, the following output :

```
Sphinx 0.9.9-release (r2117)
Copyright (c) 2001-2009, Andrew Aksyonoff

using config file '/usr/local/sphinx/etc/sphinx.conf'...
indexing index 'test1'...
collected 4 docs, 0.0 MB
sorted 0.0 Mhits, 100.0% done
total 4 docs, 193 bytes
total 0.034 sec, 5561 bytes/sec, 115.26 docs/sec
total 2 reads, 0.000 sec, 0.1 kb/call avg, 0.0 msec/call avg
total 7 writes, 0.000 sec, 0.1 kb/call avg, 0.0 msec/call avg
```

Figure 33: Running of the index.

7. In this step your are going to querie the index, to see if it works by writting the following command (we are searching for the word "test"):

```
$ /usr/local/sphinx/bin/search test
```

Figure 34: Checking of the index.

8. As we can see on the following image, the index works very well and gave us as result the document 1,2,4 which are the ones that we exepected :

```
displaying matches:
1. document=1, weight=2, group_id=1, date_added=Thu Jul 22 17:26:25 2010
        id=1
        group_id=1
        group_id2=5
        date_added=2010-07-22 17:26:25
        title=test one
        content=this is my test document number one. also checking search w
2. document=2, weight=2, group_id=1, date_added=Thu Jul 22 17:26:25 2010
        id=2
        group_id=1
        group_id2=6
        date_added=2010-07-22 17:26:25
        title=test two
        content=this is my test document number two
3. document=4, weight=1, group_id=2, date_added=Thu Jul 22 17:26:25 2010
        id=4
        group_id=2
        group_id2=8
        date_added=2010-07-22 17:26:25
        title=doc number four
        content=this is to test groups

words:
1. 'test': 3 documents, 5 hits
```

Figure 35: Query result.

The previous example has shown a way to use Sphinx full-text searching for asking data from the database. Full-text searching is a very interesting alternatives in searching because :

1. It quicker than traditional search because of the indexation of words
2. It gives result that are sorted by relevance
3. It performs very well on huge databases with millions of records
4. It skips the common words such as the, an, for etc ...

0.4.6 Index for faster search

In the previous section we talked about, the indexer (indexes) but we didn't see how this indexes optimize our queries. The aim of this section is to present indexes and see how they help in searching.

Lets start first with a formal definition (from wikipedia):

Definition : As show by the Figure 36, a database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of slower writes and increased storage space.

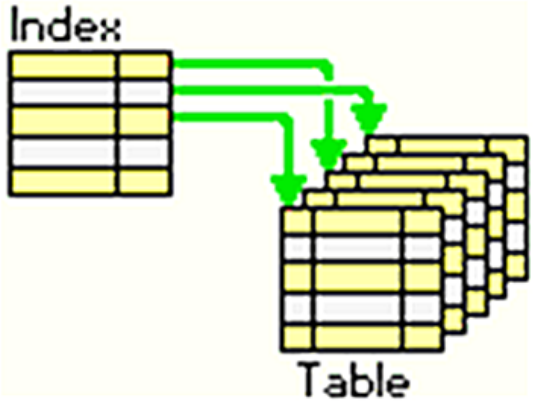

Figure 36: Index diagram

To understand this definition lets use an example :

A bookstore has a catalog of all the books that are available in the store. If the bookseller want to look quickly for a particular book, he will search through the catalog instead of searching through every isle or shelf. In this example the catalog acts as an index of all the books.
In the computing world an index is something similar. It saves the user from the trouble of having to search through every record in the database. Instead, it optimize the search by searching in a subset of data. This set of data is called an index and it is separate from the original data stored in the database.

Indexes with Sphinx

Indexes in Sphinx are a bit different from databses indexes in the sence that the data that Sphinx indexes is a set of structured documents and each document has the same set of fields. This is similary to SQL, where each row in the table corresponds to a document and each column to a field.

Sphinx allows also the adding of indexes attributes that are highly optimized for filtering. These attributes are not full-text indexed and do not contribute to matching. However, they are very useful at filtering out the results we want, based on attribute values.

0.4.7 Sphinx vs MySQL and Lucene based tools

As said in 0.0.3 section subsection Trend of Sphinx, Solr and Elasticsearch. We are going to show in this section advantages of Sphinx over MySQL and Lucene based tools as Solr and Elasticsearch. We will show also benefits of using Sphinx in an application.
There exists many Database Management System that offer a support for full-text indexing and searches (MySQL) and there exists also many external full-

text search engines (Solr, Elasticsearch). But Sphinx stands out of these other alternatives by the fact that :

1. Sphinx has higher indexing speed, estimated to be 50 to 100 times faster than MySQL FULLTEXT and 4 to 10 times faster than other external search engines.

2. Sphinx provide an higher searching speed since it depends heavily on the mode, Boolean vs. phrase, and additional processing.The searching speed is estimated to be 500 times faster than MySQL FULLTEXT in cases involving a large result set with GROUP BY and two times than other external search engines.

3. For search engines relevancy is among the key features and Sphinx performs very well in this area.

4. Sphinx has better scalability than MySQL, Solr and Elasticsearch. It can be scaled vertically (utilizing many CPUs, many HDDs) or horizontally (utilizing many servers).

The following table present Sphinx performances in a study that used approximately 3.5 Million records with around 5 GB of text.

	MySQL	Lucene	Sphinx
Indexing time, min	1627	176	84
Index size, MB	3011	6328	2850
Match all, ms/q	286	30	22
Match phrase, ms/q	3692	29	21
Match bool top-20, ms/q	24	29	13

Figure 37: Sphinx performances vs MySQL and Lucene

The main contestants of Sphinx are *ElasticSearch* and *Solr*.
We saw in different comparisons about these search engines that in term of performance the three of them are at the same level. So we focused on what Sphinx is better at.
For the topic of our application, we don't really need data visualization and analysis. We just want an application for the user to search for items in a fast way. What really pushes Sphynx to the top is the tight relation it has with SQL. MySQL language being used worldwide, the implementation of Sphinx could be easier for programmers that have already used MySQL.

0.5 Build an Application with Sphinx

0.5.1 Architecture of the database

The architecture we thought is represented at Figure 38. Unfortunately, we had some issues implementing it with Sphinx because of the indexing system being complicated and not well documented. Thus we decided to create only one table in the database containing all the attributes of table 1. For the purpose

of the comparison between Sphinx and MySQL the one table doesn't create a performence gap between these two technologies.

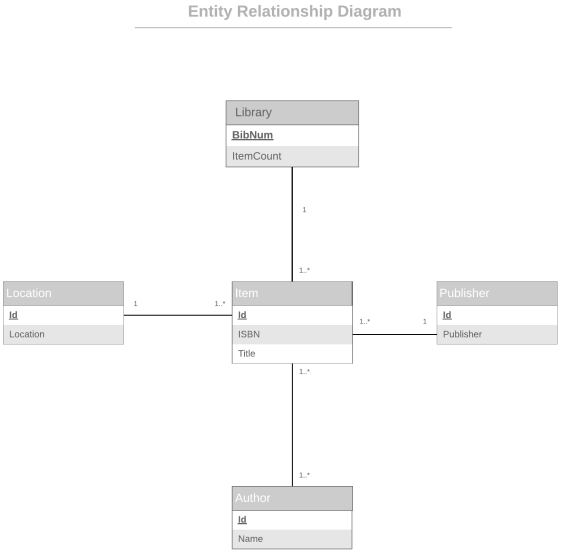

Figure 38: Relation model

0.5.2 Topic of the application

We created an application in Java with a graphical interface using JavaFX. The main topic of the application is about items searches in the Seattle library. We created an application that could be used by someone who wants to search for an item in the library of Seattle.

The dataset was downloaded from the website *Kaaggle*[1]. The size of the file we downloaded was about 2.1 GB. The dataset was contained in a csv file with the following columns 1

[1] `https://www.kaggle.com/datasets`

Attribute name	Attribute type	Details
BibNum	int	The unique identifier for a cataloged item within the Library's Integrated Library System (ILS)
Title	varchar	The full title of an item
Author	varchar	The name of the first author of the title, if applicable
ISBN	varchar	Comma-delimited list of ISBN(s) for this title
PublicationYear	varchar	Date (year) of publication
Publisher	varchar	The name of the publishing company for this item
Subjects	varchar	A comma-separated list of the subject authority records associated with the title, including Motion Pictures, Computer Programming, etc. Typically these are highly specific
ItemType	varchar	Horizon item type. Look up value descriptions in HorizonCodes using CodeType "ItemType"
ItemCollection	varchar	Collection code for this item. Look up value descriptions in: Integrated Library System (ILS) Data Dictionary
FloatingItem	varchar	Label that indicates if an item floats
ItemLocation	varchar	Location that owned the item at the time of snapshot. 3-letter code. Note: as of 2017, some items are "FLOATING" which means they don't necessarily belong to a specific branch. Location of given copy could change based on where the item is returned
ReportDate	date	The date when this item count was collected from the ILS (Horizon)
ItemCount	int	The number of items in this location, collection, item type, and item status as of the report date

Table 1: Attributes table

The datasets contains 714.566 items registered in the database. Those items can include books, music cds, dvd's, e.t.c.

0.5.3 Loading data into the database

In order to load the csv file into the database we created an sql script :

```
USE testSphinx;
DROP Table IF EXISTS Books;
CREATE TABLE Books (
 BibNum int NOT NULL,
 Title varchar(255) DEFAULT ' ',
 Author varchar(255) DEFAULT ' ',
 ISBN varchar(255) DEFAULT ' ',
 PublicationYear varchar(255) DEFAULT ' ',
```

```
 Publisher varchar(255) DEFAULT ' ',
 Subjects varchar(255) DEFAULT ' ',
 ItemType varchar(255) DEFAULT ' ',
 ItemCollection varchar(255) DEFAULT ' ',
 FloatingItem varchar(255) DEFAULT ' ',
 ItemLocation varchar(255) DEFAULT ' ',
 ReportDate DATE,
 ItemCount int NOT NULL,
 PRIMARY KEY (BibNum)
);

LOAD DATA LOCAL INFILE '/home/ieduardoramos/Desktop/library-collection-inventory.csv'
INTO TABLE Books
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
```

0.5.4 Configuration of sphinx

In this section we are going to explain how we configured sphinx and implemented it in the application. Everything was done using Linux and more precisely, Ubuntu and the application use Java (version java 11.0.1) as a programming language.

Sphinx.conf file

First of all, before starting to manipulate the configuration file, it is important to know what kind of queries will be executed in the database. In fact, we will use sphinx to create indexes which will speed up the requests made in the database. For example, in our case, we will need several indexes for :

- all the attributes
- the attribute "Title"
- the attribute "Author"
- the attribute "ISBN"
- the attribute "PublicationYear"
- the attribute "Publisher"

Taking all of this into consideration, we can start and complete the configuration file.
To begin, we will create a source which will be used by all the other sources. It will contain the user and the database we want to use.

```
source basic
{
  type          = mysql
  #SQL settings
  sql_host      = localhost
  sql_user      = user #your username
  sql_pass      = password #your user password
  sql_db        = database #Name of the database
  sql_port      = 3306 # optional, default is 3306
}
```

Now, after we added this source in the configuration file, we can create the indexes. The first index that we will create will be used to search in all the attributes :

```
source search_all : basic
{
  sql_query     = \
  SELECT * \
  FROM Books
}
index searchAll
{
  source        = search_all
  path          = /var/lib/sphinxsearch/data/search_all
  docinfo       = extern
}
```

The creation of the other indexes are very similar. In the source section, we set how the index should work and where it has to search in the database. In the index section, we set where to store the index in the computer and what his source is.
For the configuration of the "sphinx.conf" file, there is a lot of other existing parameters but we don't need them in our case. The other indexes are very similar :

```
source search_title : basic
{
  sql_query     = \
  SELECT BibNum,Title \
  FROM Books
  sql_field_string = Title
}
index searchTitle
{
```

```
  source        = search_title
  path          = /var/lib/sphinxsearch/data/search_title
  docinfo       = extern
}

source search_author : basic
{
  sql_query     = \
  SELECT BibNum,Author \
  FROM Books
  sql_field_string = Author
}
index searchAuthor
{
  source        = search_author
  path          = /var/lib/sphinxsearch/data/search_author
  docinfo       = extern
}

source search_ISBN : basic
{
  sql_query     = \
  SELECT BibNum,ISBN \
  FROM Books
  sql_field_string = ISBN
}
index searchISBN
{
  source        = search_ISBN
  path          = /var/lib/sphinxsearch/data/search_ISBN
  docinfo       = extern
}

source search_publisher : basic
{
  sql_query     = \
  SELECT BibNum,Publisher \
  FROM Books
  sql_field_string = Publisher
}
index searchPublisher
{
  source        = search_publisher
  path          = /var/lib/sphinxsearch/data/search_Publisher
  docinfo       = extern
}
```

```
source search_date : basic
{
  sql_query     = \
  SELECT BibNum,PublicationYear \
  FROM Books
  sql_field_string = PublicationYear
}
index searchDate
{
  source        = search_date
  path          = /var/lib/sphinxsearch/data/search_date
  docinfo       = extern
}
```

We can notice that the difference between the first index and the other ones is the "sql query". In fact, in this case we want to search in a special attribute and for this, because sphinx works with IDs, it is important to put the BibNum, which is the primary key, and the attribute that indicate where to search. Another important detail is to add the type of the attribute. For example, all the data of the attribute Title are Strings and so, it is necessary to add this detail in the file. It is also possible to determine other attribute type as integer, date, character and so on.
Finally come the configuration of the searchd which will contains the port and other variables to run the Sphinx daemon :

```
searchd
{
  listen            = 9312:sphinx       #SphinxAPI port
  listen            = 9306:mysql41      #SphinxQL port
  log               = /var/log/sphinxsearch/searchd.log
  query_log         = /var/log/sphinxsearch/query.log
  read_timeout      = 5
  max_children      = 30
  pid_file          = /var/run/sphinxsearch/searchd.pid
  seamless_rotate   = 1
  preopen_indexes   = 1
  unlink_old        = 1
  #Path where all the indexes are stored :
  binlog_path       = /var/lib/sphinxsearch/data
}
```

Once everything is done, we can save the configure file and launch the following commands to create the indexes and execute the sphinx daemon :

```
sudo indexer --all
```

```
searchd
```

Sphinx API implementation in Java

Following section 0.4.2, we downloaded a sphinx repertory to install the search engine. In this same repertory, to create the API of sphinx, we will open a terminal and execute the following commands :

```
cd api
cd java
make
```

These commands will install the api of sphinx which, after the "make", should appear in the java repertory with a name similar to "sphinxapi.jar". After that, the '.jar' file will be added to the java project. This last step depends on how the java project is compile and executed. In this case, we use Eclipse(version 4.8.0) as an IDE and to add the sphinx api we did the following :

```
Select a java project
Select the "Project" tab of Eclipse and go to :
Properties -> Java Build Path ->  Add External JARs...
-> sphinxapi.jar #search for the file -> Open -> Apply and Close
```

It is now possible to use Sphinx in the java project.

0.5.5 Screenshots and practical uses of the application

The Seattle library charge us about creating an application that could be used by everyone. This application goal is to help the client know what type of item can be found in the library. The requirements were to create an user friendly application and use a effective search engine that could be used with MySQL. The Seattle library has, as a database, a single table with all the items in the library.

The figure 39 shows the main page application. It is a simple JavaFX scene with a search bar and some parameters. The first parameter or filter is *Type of the research*. It allows the client to search for an author, an item title, a specific year, e.t.c.
The second parameter is *Advanced parameters* which allows the client to set how the results will be ranked. There are three *ranking modes* :

- **Proximity ranking mode** : the results are ranked by the distances calculated internally by this sphinx mode. We can configure the mode to give a certain weight for a specific document.

- **Match any ranking mode** : the results will show a match if any word from the query is found.
- **None ranking mode** : a weight of 1 is assigned to all matches which makes it faster than the other modes

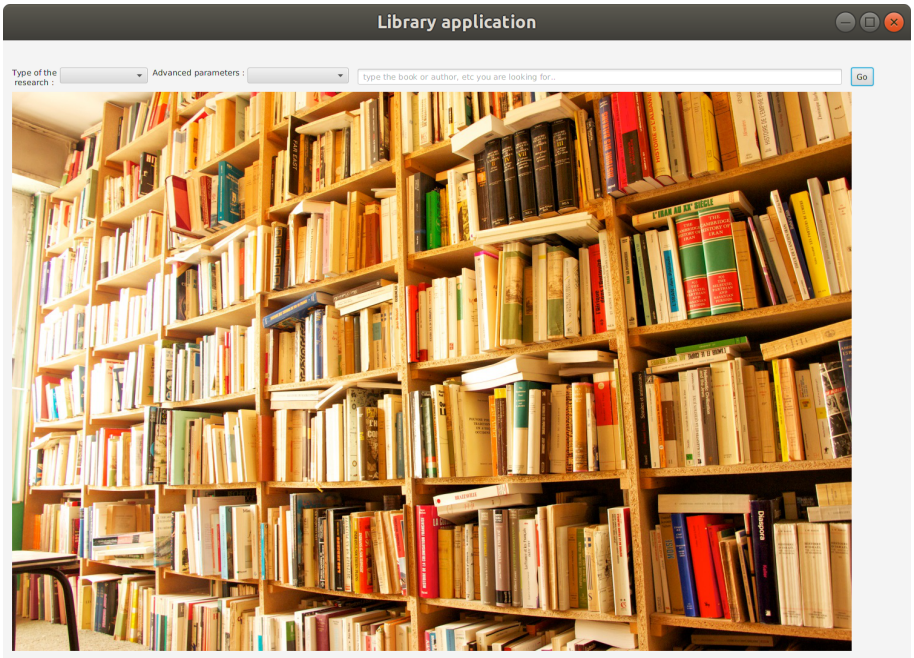

Figure 39: application homepage

The screenshot 40 show the results for the query "john arthur" in the authors research with the *proximity ranking mode*. We display the *Title, Author and ISBN* for each item found. We displayed in figure 41 the same research but with the *None ranking mode* to highlight the differences between the two modes.

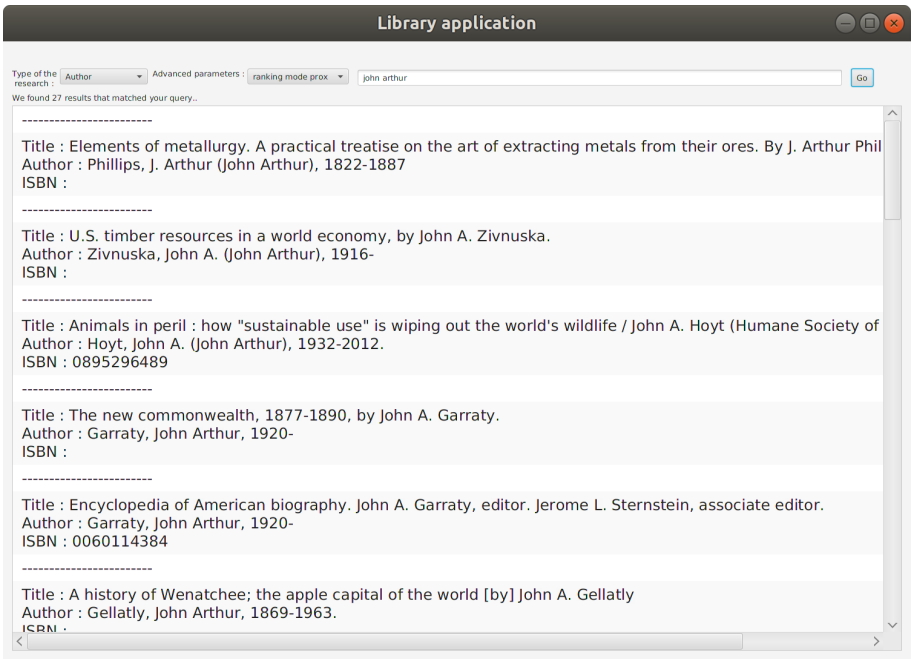

Figure 40: research by author with proximity ranking mode

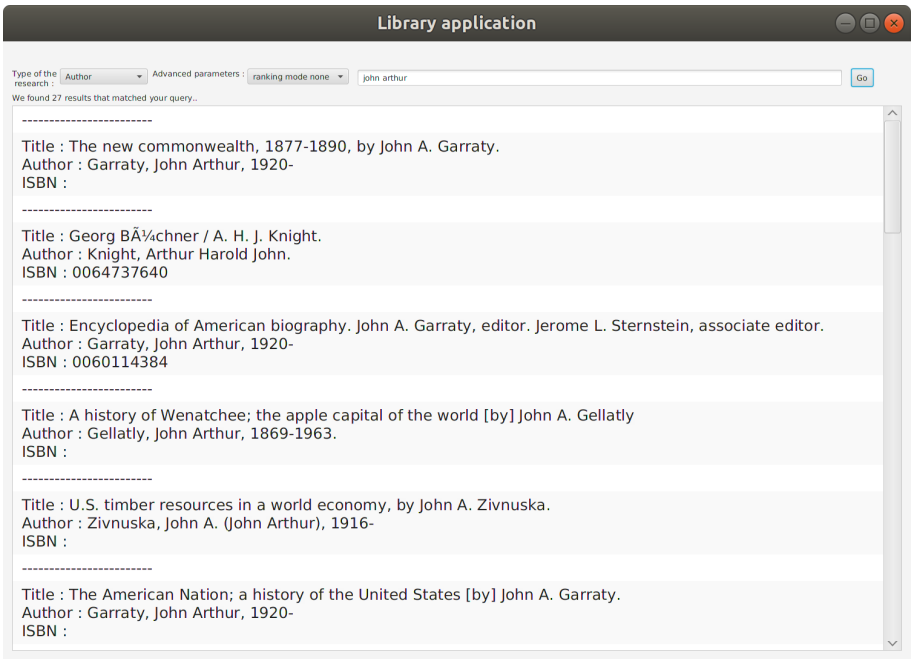

Figure 41: research by author with None ranking mode

0.5.6 Comparison of some queries between Sphinx and MySql

As it was mentioned before, sphinx queries use indexes to find results. To begin the comparison, we will write many mysql queries and see the amount of time it takes to execute them. These queries will then be translated into sphinx queries and the execution time will be compare.

- **Search for books with the word "repertory" in his title.**
  The mysql query :

  ```
  SELECT BibNum
  FROM Books
  WHERE Title LIKE "%repertory%";
  ```

  The result can be seen in figure 42

```
mysql> Select BibNum FROM Books WHERE Title LIKE "%repertory%";
+---------+
| BibNum  |
+---------+
|  157148 |
| 3163686 |
| 2706378 |
|  447186 |
| 2483484 |
| 3207560 |
|  104463 |
| 2851176 |
| 1393242 |
|  497361 |
|  762532 |
|   97029 |
|  114942 |
|  243738 |
|  329134 |
| 1860436 |
|  448273 |
| 1794909 |
| 1828295 |
|  531152 |
| 3221087 |
|  261646 |
|  747187 |
|  583884 |
|  399622 |
|  399691 |
|  411157 |
|  602703 |
| 1890735 |
|  525920 |
|  248599 |
+---------+
31 rows in set (0.20 sec)
```

Figure 42: All books with the word "repertory" in his title with mysql

The sphinx query :

```
SELECT id
FROM searchTitle #Search in the right index
WHERE MATCH('repertory')
LIMIT 1000; #Set to 1000 if not, sphinx will by default give 20 results
```

The result can be seen in figure 43

```
mysql> Select id from searchTitle WHERE MATCH('repertory') LIMIT 1000;
+---------+
| id      |
+---------+
|   97029 |
|  104463 |
|  114942 |
|  157148 |
|  243738 |
|  248599 |
|  261646 |
|  329134 |
|  399622 |
|  399691 |
|  411157 |
|  447186 |
|  448273 |
|  497361 |
|  525920 |
|  531152 |
|  583884 |
|  602703 |
|  747187 |
|  762532 |
| 1393242 |
| 1794909 |
| 1828295 |
| 1860436 |
| 1890735 |
| 2483484 |
| 2706378 |
| 2851176 |
| 3163686 |
| 3207560 |
| 3221087 |
+---------+
31 rows in set (0.01 sec)
```

Figure 43: All books with the word "repertory" in his title with sphinx

- **Search for all the books made by "John William Ransom".**
  The mysql query :

  ```
  SELECT BibNum
  FROM Books
  WHERE Author LIKE "%John William Ransom%";
  ```

  The result can be seen in figure 44

```
mysql> SELECT BibNum FROM Books  WHERE Author LIKE "%John William Ransom%";
+--------+
| BibNum |
+--------+
|     12 |
|   1215 |
|   1437 |
|  17324 |
|  21229 |
|  82428 |
|  88128 |
|  90824 |
| 100442 |
| 109534 |
| 123656 |
| 212073 |
| 216093 |
| 319189 |
| 319191 |
| 319193 |
| 319198 |
| 319214 |
| 319219 |
| 319222 |
| 319226 |
| 319230 |
| 319233 |
| 319236 |
| 319239 |
| 319243 |
| 319251 |
| 319254 |
| 319257 |
| 319259 |
| 319441 |
| 319445 |
| 364265 |
| 364266 |
| 502249 |
| 589937 |
| 706917 |
| 717226 |
| 258885 |
+--------+
39 rows in set (0.12 sec)
```

Figure 44: All the books made by "John William Ransom" with mysql

The sphinx query :

```
SELECT id
FROM searchAuthor
WHERE MATCH('John William Ransom')
LIMIT 1000;
```

The result can be seen in figure 45

```
mysql> Select id from searchAuthor WHERE MATCH('John William Ransom') LIMIT 1000;
+--------+
| id     |
+--------+
|     12 |
|   1215 |
|   1437 |
|  17324 |
|  21229 |
|  82428 |
|  88128 |
|  90824 |
| 100442 |
| 109534 |
| 123656 |
| 212073 |
| 216093 |
| 319189 |
| 319191 |
| 319193 |
| 319198 |
| 319214 |
| 319219 |
| 319222 |
| 319226 |
| 319230 |
| 319233 |
| 319236 |
| 319239 |
| 319243 |
| 319251 |
| 319254 |
| 319257 |
| 319259 |
| 319441 |
| 319445 |
| 364265 |
| 364266 |
| 502249 |
| 589937 |
| 706917 |
| 717226 |
| 258885 |
+--------+
39 rows in set (0.00 sec)
```

Figure 45: All the books made by "John William Ransom" with sphinx

- **Search how many books the authors with "Zack" as a name wrote.**
  The mysql query :

  ```
  SELECT count(*),Author
  FROM Books
  WHERE Author RLIKE "[[:<:]]Zack[[:>:]]" #Zack is not a subword.
  GROUP BY Author;
  ```

  The result can be seen in figure 46

```
mysql> SELECT count(*),Author FROM Books WHERE Author RLIKE "[[:<:]]Zack[[:>:]]" GRO
UP BY Author;
+----------+-----------------------------------+
| count(*) | Author                            |
+----------+-----------------------------------+
|        1 | Arnstein, Zack, 1980-             |
|        1 | Brown, Zack (Zack Danger),        |
|        1 | Burgess, Zack                     |
|        1 | Bush, Zack                        |
|        1 | Clark, Zack Loran                 |
|        2 | Davisson, Zack                    |
|        1 | Eswine, Zack, 1969-               |
|        2 | Giallongo, Zack, 1979-            |
|        3 | Greenburg, Zack O'Malley          |
|        1 | Hample, Zack, 1977-               |
|        1 | Kaplan, Zack                      |
|        1 | King, Zack                        |
|        2 | McDermott, Zack                   |
|        4 | Norris, Zack                      |
|        2 | Parsons, Zack                     |
|        3 | Whedon, Zack                      |
|        1 | Zack, Arnold                      |
|        2 | Zack, Devora                      |
|        1 | Zack, Ian,                        |
|        1 | Zack, Naomi, 1944-                |
|       12 | Zombie, Zack (Fictitious character) |
+----------+-----------------------------------+
21 rows in set (0.71 sec)
```

Figure 46: Find number of book written by the authors who has "Zack" as a name with the help of mysql

The sphinx query :

```
SELECT count(*) AS total,Author
FROM searchAuthor
WHERE MATCH('Zack')
GROUP BY Author
LIMIT 1000;
```

The result can be seen in figure 47

```
mysql> SELECT count(*) AS total,Author FROM searchAuthor WHERE MA
TCH('ZACK') GROUP BY Author LIMIT 1000;
+-------+-----------------------------------+
| total | author                            |
+-------+-----------------------------------+
|     1 | Brown, Zack (Zack Danger),        |
|     1 | Zack, Arnold                      |
|     1 | Arnstein, Zack, 1980-             |
|     2 | Parsons, Zack                     |
|     2 | Zack, Devora                      |
|     3 | Whedon, Zack                      |
|     3 | Greenburg, Zack O'Malley          |
|     1 | Hample, Zack, 1977-               |
|     2 | Giallongo, Zack, 1979-            |
|     4 | Norris, Zack                      |
|     1 | King, Zack                        |
|     1 | Zack, Ian,                        |
|     1 | Zack, Naomi, 1944-                |
|     1 | Eswine, Zack, 1969-               |
|     2 | Davisson, Zack                    |
|    12 | Zombie, Zack (Fictitious character) |
|     1 | Burgess, Zack                     |
|     2 | McDermott, Zack                   |
|     1 | Kaplan, Zack                      |
|     1 | Clark, Zack Loran                 |
|     1 | Bush, Zack                        |
+-------+-----------------------------------+
21 rows in set (0.00 sec)
```

Figure 47: Find number of book written by the authors who has "Zack" as a name with the help of sphinx

For every query, it is easily noticeable that sphinx is much more faster than mysql, some times it is even a hundred time faster than mysql. Larger is the table, greater will be the use of sphinx in compare to mysql.

0.6 Conclusion

With previous sections we explain the main concept of Sphinx and we reviewed the main features of Sphinx and the reasons why Sphinx could be use in an environment where the programmers and databases managers use MySQL language as their daily language. Sphinx allows you to query searches with high performances and using MySQL with your SphinxAPI. Thus it is quick to adapt to sphinx with MySQL knowledge.

Sphinx fits perfectly in the increasing need of searches engines that can manage enormous amounts of data and performing queries very fast. It is clear from what we saw through this report that Sphinx is much more than just a query performer, Sphinx implements several functionalities (i.e, filters, sorting parameters, ranking parameters, ...) making it a complete search engine.

On the other hand, the configuration of Sphinx is complex. It takes some

time to handle the configuration file because of all the possibilities Sphinx offers. Another difficulty to Sphinx is the few documentation existing. Sphinx has an official website with some documentation, mostly for the installation, but documentation about the API is rare and out-to-date.

We recommend the utilization of Sphinx in the two following scopes :

- It is quite fast and powerful for indexing and querying huge volumes of documents using limited computing resources

- Sphinx is good for structured data (predefined text fields and non-text attributes)

0.7 References

1. Sphinx Search Beginner's Guide, Abbas Ali
2. Introduction to search with Sphinx, Andrew Aksyonoff
3. An Analysis on the Comparison of the Performance and Configuration Features of Big Data Tools Solr and Elasticsearch, Mustafa Ali AKCA*1, Tuncay AYDOĞAN2 , Muhammer İLKUÇAR3
4. Big Data will impact every part of your life, Charlie Stryker,(video)
5. DB-Engines, Knowledge Base of Relational and NoSQL Database Management Systems
6. A performance comparison of SQL and NoSQL databases, Yishan Li University of Auckland
7. Full Text Search Engine as Scalable k-Nearest Neighbor Recommendation System , Ján Suchal and Pavol Návrat
8. Indexing Bibliographic Database Content Using MariaDB and Sphinx Search Server, Arie Nugraha
9. Elasticsearch VS. Solr VS. Sphinx: Best open source search platform comparison, by Anna Klimenko
10. Sphinx Official website : `http://sphinxsearch.com/`