# TEMPORAL SEARCH ENGINE

Extraction and Indexation
Paris-Saclay University – M2 AIC 2016

Supervisor:
Xavier Tannier
*(Xavier.Tannier@u-psud.fr)*
Thomas Lavergne
(*Thomas.Lavergne@limsi.fr*)

Members:
VU Trong Bach
NGO HO Anh Khoa
Divya GROVER
Mahmut CAVDAR

# TABLE OF CONTENTS

# INDEX OF IMAGES

# CHAPTER I. DESIGN AND IMPLEMENTATION

## 1. PROBLEM

We are tasked to implement a Temporal Search Engine besides implementing a simple Engine. We have chosen to keep separate the two-parts as it would be better to give the user a choice to search a query by looking at separate results based on document-pertinence and time-pertinence respectively.

## 2. CORPUS

The corpus used in this project is a collection of 1.55 million of web news documents being the articles of francophone press. This collection is released from March 2013 to September 2015.

This corpus has "*feed_list.txt"*, "*index"* folder, "*subindex"* folder*, "data"* folder and *"text"* folder. "*feed_list.txt"* keeps the list of RSS sources with the topics such as economic, sport, politic etc. "*index"* folder has document index with the XML format which keeps information about: "*date", "name", "id", "url", "dct"* document creation time and "*dd"* document download time. This project uses "*dct"* information for temporal search engine. "*dct"* has the format "YYYYMMDDTHHMMSSZ" where YYYY, MM, DD, HH, MM and SS are year, month, day, hour, minute and second respectively. *"subindex"* folder keeps about 10 000 documents between January to April 2015. *"data"* folder keeps all of the web pages downloaded and *"text"* folder has the main content of each article.

## 3. SIMPLE SEARCH ENGINE

The simple engine has been implemented using an indexation scheme based on storing of term-frequencies and document-frequencies of each word in the corpus. The weight measured for a word within each a document is then described by the following formula:

$$wtd = tftd \times log \frac{N}{dft}$$

Where,

$wtd$ = Weight of term (t) in document (d) relative to the corpus

$tftd$ = Frequency of term in the document

$dft$ = Frequency of documents in the corpus containing this word

N = Number of documents in the corpus

### 3.1. Design aspects

It is important to note the following details about the corpus used for simple engine:

1. It consists of a collection of 9,843 files which were extracted from a larger corpus of 1.55 million web-news documents.
2. The original HTML files were filtered into simple txt format, also already provided in the corpus.
3. The size of these 9,843 files is registered to be around 31.4 MB but it is also important to note that the names of these are represented as long strings, which significantly increases its on-disk size.
4. A test.py file is provided with the code, which extracts these specific 9,843 files from the larger corpus, using the sub index folder, which provides for the identifiers of each of these.

Some important constraints that were followed while implementing the simple Engine:

1. Most difficult constraint was to limit the size of indexation data used to 60% of the corpus size, which translates to <18.84 MB of disk-space.
2. Other constraint includes never to exceed live-memory of more than 1GB, at neither indexation process of the corpus nor while querying the index.

Therefore, the design decisions taken to incorporate the nature of this dataset and constraints are:

1. It is impossible to take weight-files of each document as supplementation indexation information, due to the disk-space constraint. It is important to note that, after removal of stop-words, all other words, occur almost with a frequency of 1 in a given

document, which means that saving weight-information a given document, has almost the same size as the corresponding original text document. This feature of saving the weight-files can only be used if provided with a better normalizer (tokenizer, stemmer).

2. This decision directly follows the previous decision of not using the weight-files. It should be noted that if cosine-similarity measure is used to sort documents for a given query, then this measure needs to be calculated on the fly. By the nature of a query, it is assumed to be only a few words long, hence, only weights for these words need to be calculated in both the query and the document. But there are norms used in this measure, norms of both the query and the document. Note that, for ranking, the norm of query will have no effect, so its calculation is completely skipped. But to calculate the norm of each document amount to calculating the weight of each of the words occurring in them. So the second decision, is to store the norms of each document in the corpus into a single file while indexation. This file is much smaller than its alternative and helps in significantly reducing the query time.
3. Even after removing the use of weight-files, the index still exceeded the size-limit, this was due to the fact that in the inverse-index used, each document was identified by its identifier in the corpus, which happens to be a long string name in this case, hence a simple mapping/encoding is used, to represent each document by an integer, which is its lexical rank in the corpus.

The above two decisions, finally allowed the Engine to respect the size constraint. It is noted that the live-memory constraint was never an issue.

## 4. SOLR

Solr builds on open source search technology: Lucene, a Java library that provides indexing and search technology, as well as spell-checking, hit highlighting and advanced analysis/tokenization capabilities.

Solr is built to find documents that match queries. Before perform queries we need to add documents to Solr server. Solr's schema provides an idea of how content is structured. When we add a document, Solr takes the information in the document's fields and adds that information to an index. The Solr install includes the bin/post tool in order to facilitate getting various types of documents easily into Solr. **bin/post** features the ability to crawl a directory of files, optionally recursively even, sending the raw content of each file into Solr for extraction and indexing.

When we perform a query, Solr can quickly consult the index and return the matching documents. Solr can be queried via REST clients, cURL, wget, Chrome POSTMAN, etc., as well as via the native clients available for many programming languages. In this project we used java client library.

## 5. CLUSTERING

This section shows DBSCAN Clustering, the main technique used in our temporal search engine. Clustering is a family of methods for unsupervised classification by organizing data into homogeneous groups. In our case, a set of unlabeled time series is put into clusters where all its sequences grouped are coherent. DBSCAN (Density Based Spatial Clustering of Application with Noise) is an algorithm of clustering technique based on density, which determines arbitrary shaped clusters and filters noises. Its main idea is growing a cluster by adding the neighborhood objects which exceed a threshold predefined and by calculating distance between two objects. In short, for our temporal search engine, DBSCAN Clustering finds the documents having the similarity of its time data for creating the document groups which represents different periods in a timeline.

In DBSCAN algorithm, each cluster is expanded with a level of density, which discovers arbitrary shape of the data. In detail, this algorithm needs two threshold parameters: the minimum number of neighborhood object *minObjs* and the maximum distance between two objects, namely *radius epsilon Eps*. Moreover, a distance calculation method should be

selected for the different datasets. Therefore, an *Eps-Neighborhood* is an object within a radius *Eps* of an object, a *Core object* contains at least a minimum number of *Eps-Neighborhood* and a *Noise object* which is not a *Core object* are discarded. On the whole, this algorithm could be explained as below:

• An object P which has not been visited is selected and DBSCAN finds all Eps-Neighborhoods.

• If P is a *Core object*, a cluster including P and its *Eps-Neighborhoods* is formed and P is marked as visited. DBSCAN repeats this finding *Core object* step with *Eps-Neighborhoods* of P.

• If P is a *Noise object*, DBSCAN visits the next object in the dataset.

DBSCAN is used after having the result dataset from SOLR. In fact, in the previous sections, SOLR is mentioned as a search engine which returns all of the documents belonging to user request in the corpus and this dataset is a list of document with its creation time. Hence, the mission is clearly grouping these documents by using its time similarity, which means that time frames of an event are detected. In our case, because this kind of time data is unlabeled and the number of document groups could not be predefined, clustering technique with DBSCAN algorithm, an unsupervised classification technique based density becomes a method appropriate.

Main advantages of DBSCAN:

• DBSCAN does not require the number of clusters, as opposed to K-Means, another clustering algorithm. Our data is a set of article having a variety of creation times and the number of clusters depends on the event requested by user, hence it is impossible to know document group number.

• DBSCAN is aware of noise problem. In reality, web news readers are reminded of a past event in an article. However, if this event is not enough significant, it could be not mentioned in a host of other articles. To put it in a different way, if *Eps-Neighborhoods* of this article do not exceed *minObjs*, it is a *Noise object*.

• To compare with Agglomerative Hierarchical Clustering merging the two closest clusters until a single, DBSCAN does not have to calculate group average values which is not easy to be detected. In addition, merging step cause trouble for noisy.

Main disadvantage of DBSCAN:

• It is not simple to predefine two parameters *minObjs* and *Eps*. The selection of these thresholds depends on the dataset knowledge of programmer.

• The computational complexity of DBSCAN is $O(n^2)$ where n is the number of documents of a request. Our corpus has 1,55 million of document, but for each user request, a smaller number of documents is evaluated. Moreover, our data has only one dimension, which improves the complexity.

# CHAPTER II. PROJECT REALISATION

## 1. SIMPLE ENGINE

The implementation of the Engine consists of about 700 lines of codes, excluding the tools used, that were already implemented. It is described as follows:

1. A package called indexer, consisting the three classes:
    a. Indexer1: Traditional indexation, i.e. it calculates the weight-files for each of the documents in the specified corpus, not used due to disk-space constraint.
    b. Indexer2: It calculates the norm-file for the corpus, as described in the design.
    c. Indexer3: It calculates the inverse-index, containing for every word, its document frequency and corresponding term-frequency in each document. It should be noted that in this file, each document is represented by an integer. Hence the corresponding encoding file is also produced by this class.
2. A package called engine, consisting of two classes:
    a. Engine: It has methods to do the following tasks:
        i. To parse and load the inverse-index, encoding-file and norm-file.
        ii. To sort and return a list of documents for a given query.
    b. TestEngine: Its main purpose is to return a list of documents for the given queries and also copy then into a results folder. It also does performance analysis.

## 2. TEMPORAL SEARCH ENGINE

### 2.1. Solr

In the Solr universe, documents are composed of fields which are more specific pieces of information. In our case in each document has 6 different fields as id,url, creation date, download date, title and information.

Fields can contain different kinds of data. A title field, for example, is text (character data). An example of document file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<add>
    <doc>
        <field name="id">44e1d295cb9157c145a1a475cbab7aef</field>
        <field name="url">http://www.leparisien.fr/faits-divers/en-images-les-plus-gros-braquages-de-l-histoire-19-02-2013-2580481.php</field>
        <field name="dct">20130219T115700Z</field>
        <field name="dd">20130728T234746Z</field>
        <field name="title">EN IMAGES. Les cinq plus gros braquages depuis vingt ans</field>
        <field name="info">EN IMAGES.
        Les cinq plus gros braquages depuis vingt ans

        Tweeter
        100 millions de dollars (75 millions d'euros).
        En février 2003, une équipe de braqueurs italiens s'attaque au Diamond Center d'Anvers (Belgique).
        Ils font main basse sur des diamants, de l'or et des bijoux.
        L'opération est surnommée «le braquage du siècle» car les malfaiteurs ont déjoué le système de surveillance, les détecteurs infrarouges
et le coffre aux 100 millions de combinaisons.
        AFP/WIM HENDRIX
        Réagir
        Le braquage à Bruxelles d'un fourgon de la Brink's contenant 37 millions d'euros en diamants s'est soldé par l'un des butins les plus
élevés dans l'histoire récente du crime organisé.
        Mais ces vingt dernières années, d'audacieux malfrats ont déjà réussi à dérober des sommes plus impressionnantes.
        Vos amis peuvent maintenant voir cette activité Supprimer X
         Retour en images sur ces braquages de haut-vol aux montants vertigineux.
        LeParisien.fr</field>
    </doc>
</add>
```

*Figure 1. An example of document file*

In our core url, title and id have ‘string’ data type. Creation date and download date have ‘date’ data type. Lastly info field has ‘text_general’ data type. ‘text_general’ can contain very long string data in contrast to ‘string’.

The main query for a Solr search is specified via the `q` parameter. Next request allows to see the top ranking documents for the query.

*“localhost:8983/solr/REI/select?q=Catalogne”*

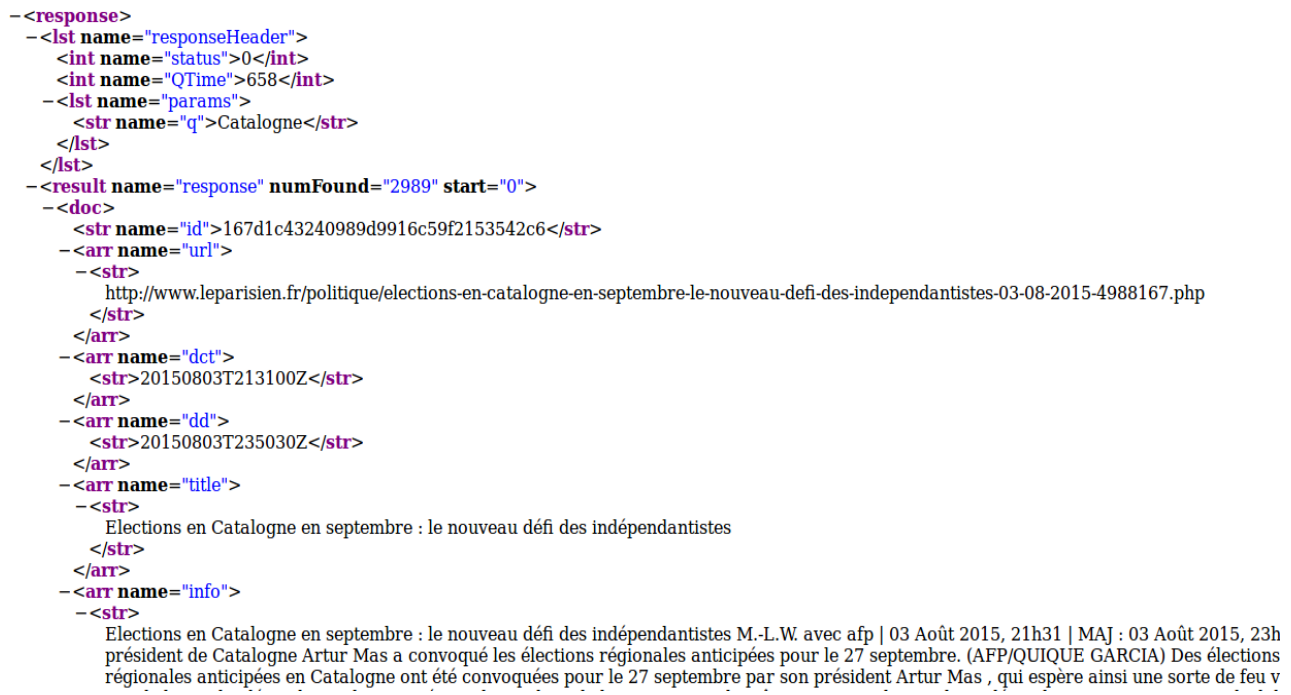



*Figure 2. The answer of Solr*

## 2.2. DBSCAN Clustering

This section describes the implementation of DBSCAN clustering and the connection to Solr.

First of all, a class for describing the articles of dataset is defined: “Document” class in package “SolrConnection” with the information of a document such as identification “id”, “title”, creation date “date”, “url” and similarity between the document and the user request “score”. Especially, “sortedIndex” variable is used for sorting and calculating the distance in clustering step.

To create a Solr connection, library Solrj 6.2.1 is used in the class “Solr” of package “SolrConnection”. This class creates the connection Client – Server from the Solr server built in “http://localhost:8983/solr/REI”. The user request is processed by method “getReponse(String queryString)” and it returns a “TreeSet<Document>” having all documents need to be clustered.

In the third place, clustering analysis is applied with "Clustering" class of package "Clustering". This step requires two parameters mentioned in the previous sections: *minObjs* and *Eps*, which affects significantly the result of clustering.

- minObjs: "Adaptive Methods for Determining DBSCAN Parameters" of Kedar Sawant [1] provides an automatic method for calculating this threshold. This method uses the average of *Eps-Neighborhoods* of all objects, which means that a different *minObjs* is determined by a different number of documents for each user request and that improves the flexibility of clustering.

$$minObjs = \frac{1}{n}\sum_{i=1}^{n} O_i$$

  Where $O_i$ is the number of *Eps-Neighborhoods* of Object i and n is the number of documents in our document TreeSet.

- Eps: The selection of this parameter depends mainly in the type of data being a set of web news articles. It is calculated by using the time of whole period when the user request appears.

$$Eps = \frac{time_{firstdocument} - time_{lastdocument}}{m}$$

  Where m is a number predefined. In this dataset, **m = 4**. Our data is a collection of web news articles from March 2013 to September 2015 and the whole period is about two years. Therefore, some thresholds could be predefined and m = 4 is the final result.

  - If it is more than one year, *Eps* is three months and m = 4.
  - If it is about a half of year, *Eps* is two months and m = 3.
  - If it is about a one month, *Eps* is one week and m = 4.
  - If it is about one week, *Eps* is 2 days and m = 3.5.

## 2.3. BURSTY FEATURES IDENTIFICATION

Assume the number of documents that contain the feature fj in a cluster Wi, denoted as ni,j, follows a generative probabilistic model, which is a model based on an unknown probability distribution.

P(ni,j) can be modeled using a hyper-geometric distribution. As a result, the probability that the feature fj in the time window Wi can be modeled by a hyper-geometric distribution. Note: hyper-geometric distribution is computational expensive such that its computational time growth quadratically with ni,j. We model the hypergeometric distribution using the binomial distribution, since the computation of binomial distribution is far more efficient. Furthermore, both distributions will eventually be the same when the database is large. Hence, P(ni,j) is modeled by a binomial distribution, and is computed as follows.

$$P_g(n_{i,j}) = \binom{N}{n_{i,j}} p_j^{n_{i,j}} (1-p_j)^{N-n_{i,j}}$$

Where N is the number of documents in a cluster.

n*i.j* is the number of documents of cluster j that contain f*i*.

pj is the expected probability of the documents that contain the feature fj in cluster j, and is therefore the average of the observed probability of fj in all clusters containing fj.

$$p_j = \frac{1}{L'} \sum_{i=0}^{L'} P_o(n_{i,j})$$

$$P_o(n_{i,j}) = \frac{n_{i,j}}{N}$$

where L′ is the number of time windows containing fj .

We discuss how likely an important feature fj will be wrongly taken as a weakword or stopword below. Suppose that fj is a bursty feature in a bursty event Ek, such that fj only appears with high frequency in the hot periods of Ek. It implies ni,j is large in the cluster Wi where Ek is a bursty event. If it occurs, all the observed probability in every sliding cluster,

Po(ni,j ) will be large, whereas the number of clusters that contain fj will be small. Therefore, it is possible in theory that fj will be wrongly taken as a weakword or stopword. However, it is most unlikely that fj will be wrongly taken as a weakword or stopword for the following reasons. Feature distributions are sparse in nature. Even though fj may be only related to Ek as its bursty feature, it is most likely that fj will appear in other time windows where Ek is not a bursty cluster. In other words, for the same feature fj , the number of documents that contain it, ni,j, will be large in some time windows and small in some other windows. On average, the observed probability, Po(ni,j), for such a feature fj , will not be large as the one for a weakword or stopword.

### 2.4. DECISION FOR A BURSTY EVENT FROM BURSTY FEATURES

As our work of clustering, each cluster has a concern with at least one event. So, we consider these clusters as periods of time when the event happened. We can calculate the probability of E in the set of documents D with the Byes Probability.

$$P(E_k|\mathsf{D}) = \frac{P(\mathsf{D}|E_k)P(E_k)}{P(\mathsf{D})}$$

Where Ek is an event in [1:L], L the total number of clusters.

We show how to compute P (D|Ek) and P (Ek) below.

- **Computing P (Ek):** Let the total number of clusters be L. We consider that a bursty feature bj is a time-series of length L, such that the i-th value in the time-series is the bursty probability Pb (i, bj) in the cluster Wi. We solve the problem of computing the probability of bursty features to be grouped together (P (Ek)) as to compute the probability of the corresponding time-series to be grouped together. This can be achieved by computing the similarity among a set of time-series given Ek. In this paper, we take a simple yet efficient and effective approach to compute P (Ek), that is, by computing the overlapping areas among different time-series.

$$P(E_k) = \frac{\bigcap_{j=0}^{|B|} \Big|_{e_j=1} a_j}{\bigcup_{j=0}^{|B|} \Big|_{e_j=1} a_j}$$

where aj is the area covered by the feature bj in the time-series.

- **Computing P (D|Ek):** We assume the feature distribution is independent, and formulate P (D|Ek) as follows. The idea behind it will be explained after introducing the formulation.

$$P(\mathsf{D}|E_k) = \prod_{j=0}^{|B|} \left(\frac{|D_j|}{|M|}\right)^{e_j} \left(1 - \frac{|D_j|}{|M|}\right)^{1-e_j}$$

$$M = \bigcup_{\substack{j=0 \\ e_j=1}}^{|B|} D_j$$

Here, M is a set of documents that contain the bursty event Ek. The first component computes the probability of the documents that contain the bursty feature bj in M, whereas the second component computes the probability of the documents that do not contain the bursty feature bj in M . In other words, if ej= 1, it implies that the bursty feature bj belongs to the event Ek, and we compute the first component; if ej=0, we compute the second component. Hence, P (D|E) computes the production of the probability of D under M where M is constructed by the given Ek.

Finally, we can decide what periods are “hot” or non with the probability of events in the set of documents.

## 2.5. Visualization

For visualization part we used D3.js library. D3.js is a JavaScript library for manipulating documents based on data. D3 helps visualization of data using HTML, SVG and CSS. (web standards) Our principal page gets json data by making GET request to second page. Second page makes a request to Solr client side for get all related result, applies clustering processes and returns json result to first page. Here are the results:

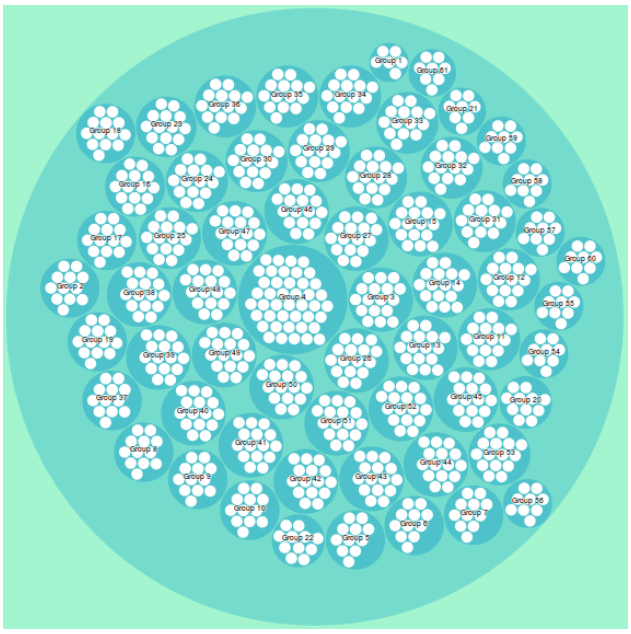


*Figure 3. Result of D3.js*

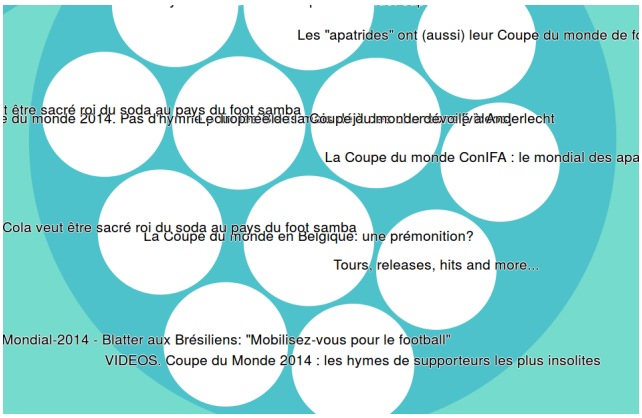



*Figure 4. A closer look of result*

Result structure should be like below: each cluster has a name and children. Each child has name and size of circle.

{"name": "cluster1",
"children": [{
"name": "cluster1_1",
"children": [{
"size": 10.050272,
"name": "example1”
}, {
"size": 9.991996,
"name": "example1” …

# 3. EVALUATION

## 3.1. Performance Analysis of Simple Engine

Time performance during various stages are as follows:

1. Indexer2: 178secs and 60secs for the stem-normalizer and token-normalizer respectively.
2. Indexer3: 68secs and 50secs for the stem-normalizer and token-normalizer respectively.
3. TestEngine: The query time is the order of 1ms.

Memory used is about 400Mb during indexation and about 100Mb during querying. The heap-memory usage for these two processes during runtime are also shown below:

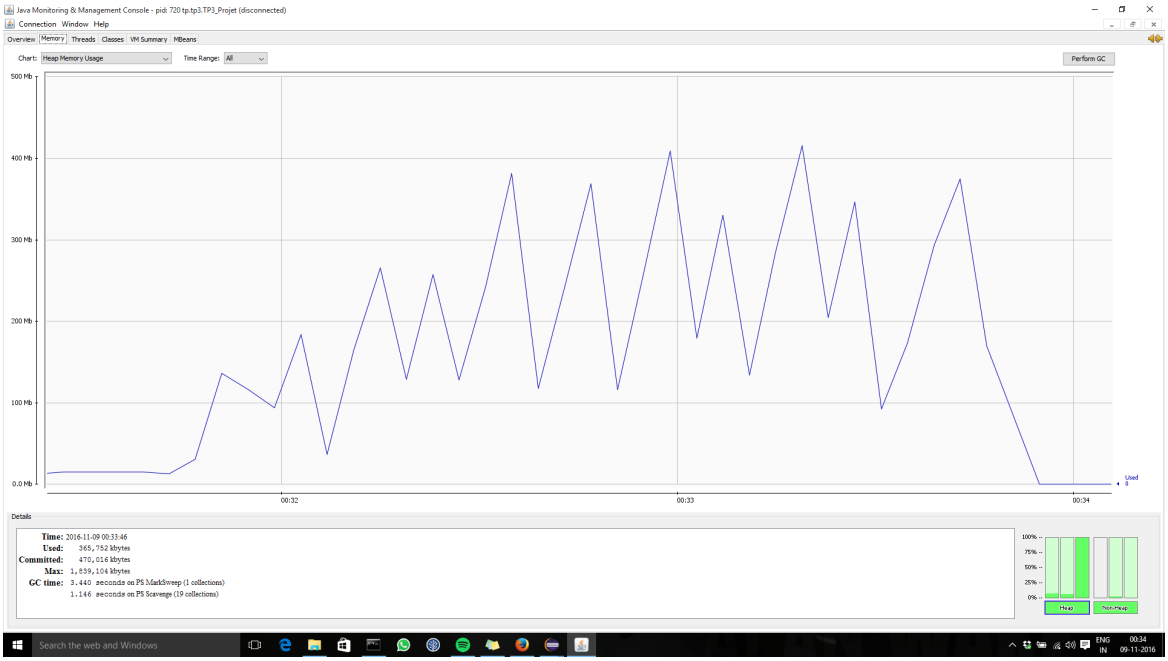



*Figure 5. Heap Memory usage during Indexation*

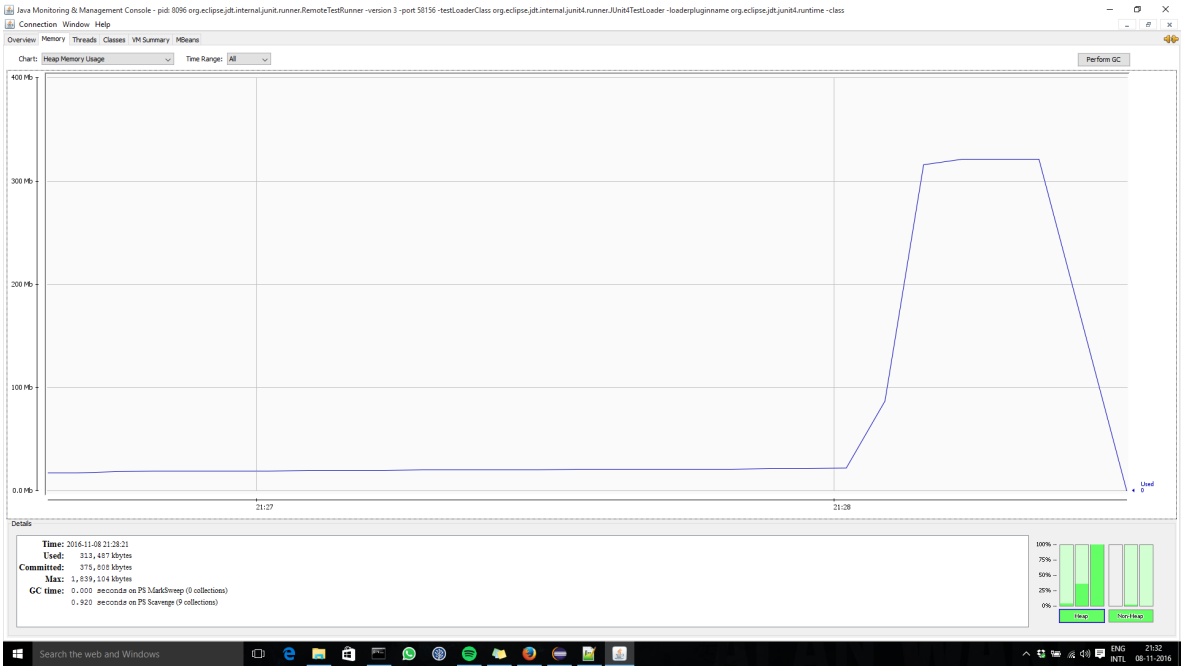
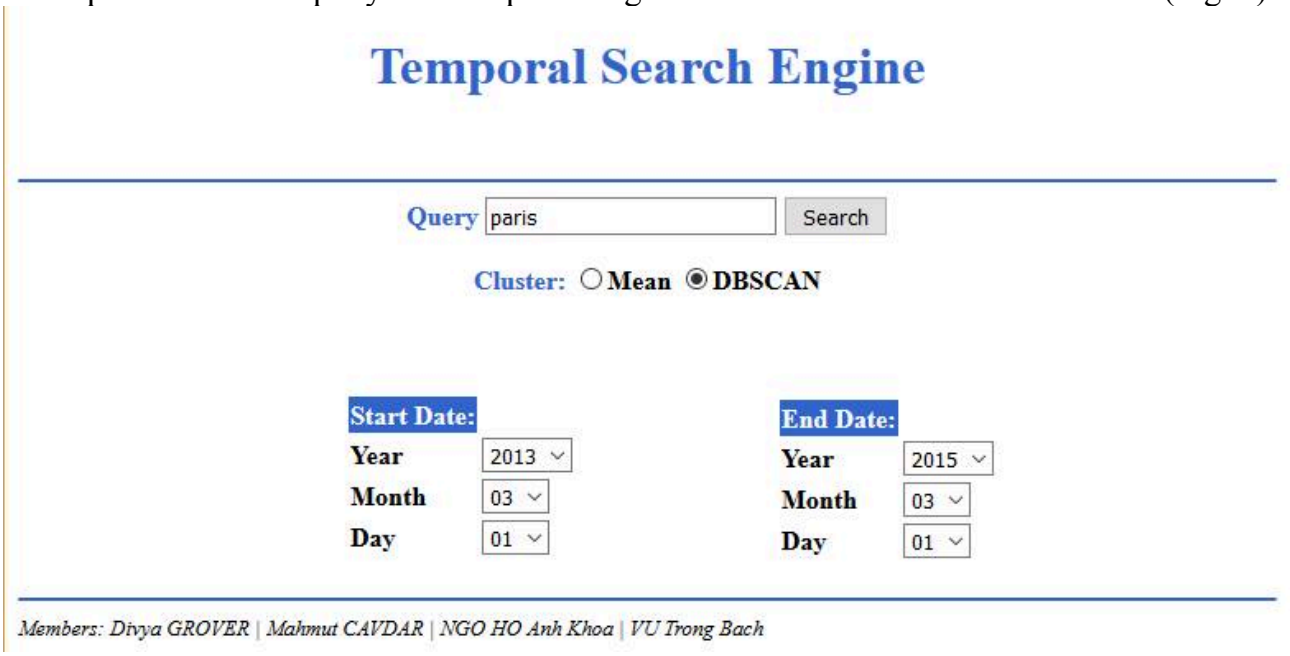


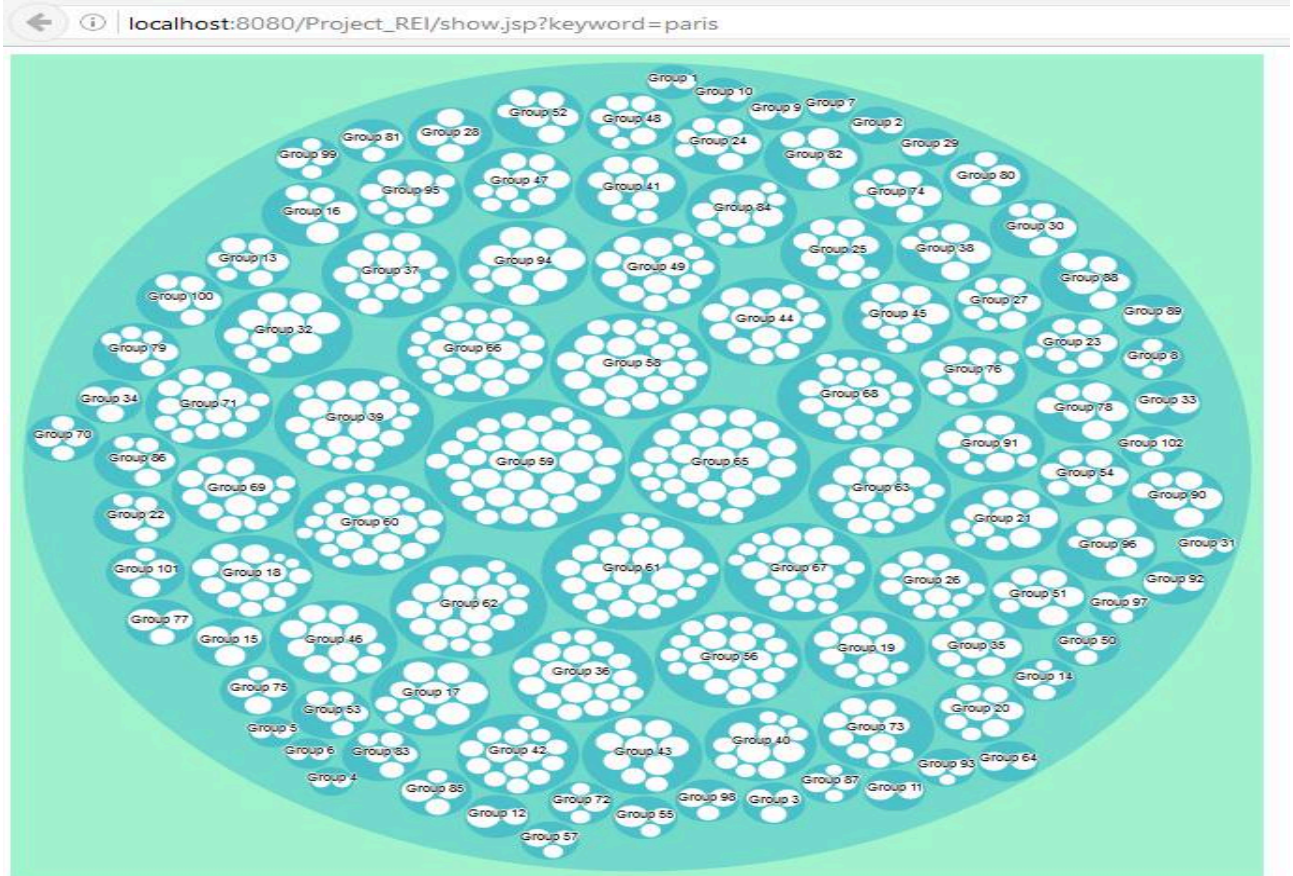

*Figure 6. Heap Memory usage during Querying*

## 4. INTERFACE

A simple interface to query the Temporal Engine has been created. It looks like this(Fig. 7):




The results for the query are visualized as follows(Fig. 8):



# CHAPTER III. CONCLUSION

The following observations can be pointed out from realization of this project:

- For the Simple search-engine implementation, keeping the index small was a challenge. It depends highly on the kind of similarity measure chosen, as it directly decides what information needs to be directly stored in the index. For more difficult similarity measures, or tighter constraints on index-size, more complex indexation techniques such as binary-tree/pointer-based/distributed indexing must be used.
- For the Temporal search-engine, Solr provides the functions necessary for indexation and a library Solrj for java connection, which improves the limits of Simple search-engine. In addition, the techniques such as clustering and feature identification require the special parameters. The selection of these parameters has the important effects to the final result of the documents requested. The simple interface implemented helps the user to search easier information.

# REFERENCES


❖ ***Document***

[1] Kedar Sawant, *Adaptive Methods for Determining DBSCAN Parameters,* IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 1 Issue 4, June 2014.

[2] Glory H.Shah, *An Improved DBSCAN, A Density Based Clustering Algorithm with Parameter Selection for High Dimensional Data Sets,* Nirma University International Conference On Engineering, 2012.

❖ ***Website***

[3] Solr and Solrj - Homepage: **http://lucene.apache.org/solr/**.

[4] D3 Javascript Library – Homepage: **https://d3js.org/**