

1. Description of the document base

We have 364 compressed files representing the daily AP Newswire stories for the year of 1989:

AP891217.Z : This is the daily story file for 17/12/1989.

The files are *UNIX* compressed .Z streams which can be uncompressed using the **uncompress** utility which is included in a typical *Linux/UNIX* system.

The files have been converted to *SGML* format and cleaned up by AT&T programs. A typical uncompressed daily story is as follows:

```
<DOC>
<DOCNO> AP891231-0001 </DOCNO>
<FILEID>AP-NR-12-31-89 2359EDT</FILEID>
<FIRST>r a PM-MonkeyBusiness      12-31 0269</FIRST>
<SECOND>PM-Monkey Business,0276</SECOND>
<HEAD>Yacht That Took Gary Hart On Famous Cruise Suffered From Fame</HEAD>
<DATELINE>DENVER (AP) </DATELINE>
<TEXT>
    Some text, and some other text, and other texts, etc..
....
....
</TEXT>
</DOC>
```

We will only be interested with the `<DOCNO>` and `<TEXT>` elements to identify and store these documents. The `<DOCNO>` will be used as the document's identifier and the content of the documents will be represented by the data found in consecutive `<TEXT>` elements.

A file contains many documents leading to 84677 documents in 364 .Z files.

2. Tools and Technologies

2.1. Programming Language

The main programming language used throughout the project is *Python*. I used *Python* because I'm very familiar with it thus it speeds me up.

Python is a cross-platform scripting language, has a rich standard library and also has external libraries on the internet that can freely be used.

2.2. External Libraries

The only external library that I've used beside the standard library was the *Python* version of the *Porter Stemmer* algorithm written by Vivake Gupta[0].

2.3. Graphical Interface Design

The graphical user interface was designed with the *Qt Framework*[1] which is a cross-platform user-interface design primarily available for *C++*, *Python*, etc.

The *Python* port of this toolkit is called *PyQt*[2].

2.4. Source code management

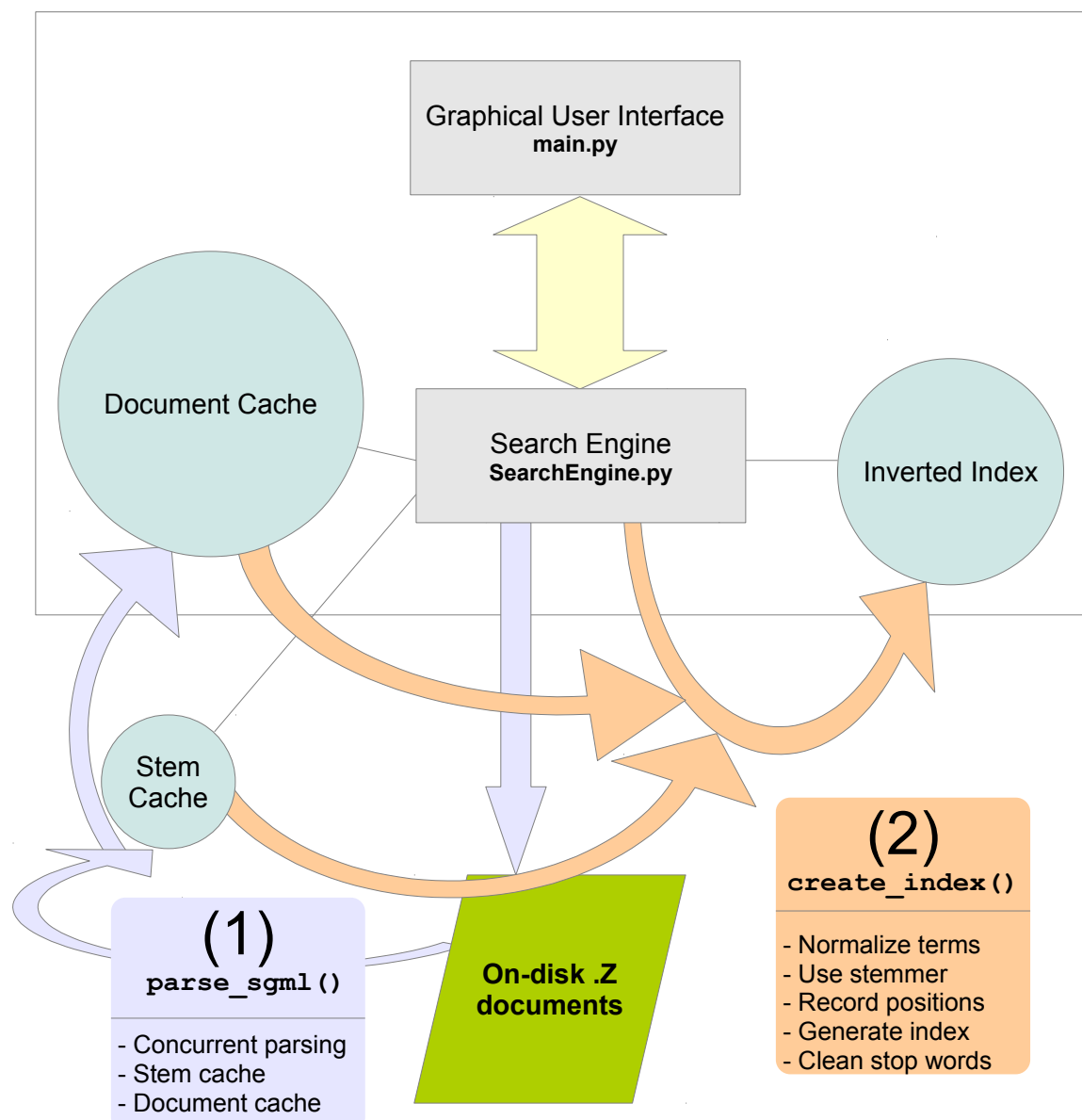
The source code of the Search Engine project can be found at the following *GIT* repository:

<https://github.com/ozancaglayan/SearchEngine>

GIT [3] is a very popular free and open-source distributed source code management heavily used by a lot of free and commercial products.

3. Design

3.1. Overview



3.2. Persistent data storage

None of the caches mentioned in the above figure are stored as an SQL database. Instead they are **pickled** dictionaries which allows fast-access times once they're loaded into memory.

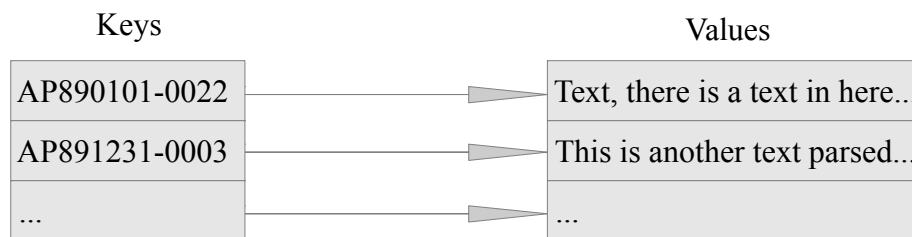
Pickle[4] is a Python object serialization method which implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. **Pickling** is the process whereby a Python object hierarchy is converted into a byte stream, and **unpickling** is the inverse operation, whereby a byte stream is converted back into an object hierarchy.

In this project we use the **cPickle** standard Python library which can be up to 1000 times faster than **pickle** because it is implemented in C. With **cPickle**, we will easily store the **document** and **index** caches on disk and load them into the process memory once the search engine is initialized.

3.3. Caches

3.3.1. Document Cache

The document cache is a simple Python dictionary which maps the parsed **document numbers** to their **relevant text streams**:



The final document cache is a pickled dictionary with 225MB on-disk size.

3.3.2. Stem Cache

Stem cache is an intermediate Python dictionary which is used to speed-up the creation of the inverted index. It stores the mapping between terms and their stems during the creation of the document cache. This dictionary is further referenced by the inverted index creation process to **not calculate the stem** of the terms again and again, thus decreasing redundancy.

Note that this cache is not involved at all during searching, can even be deleted after the creation of the inverted index.

3.3.3. Inverted Index Cache

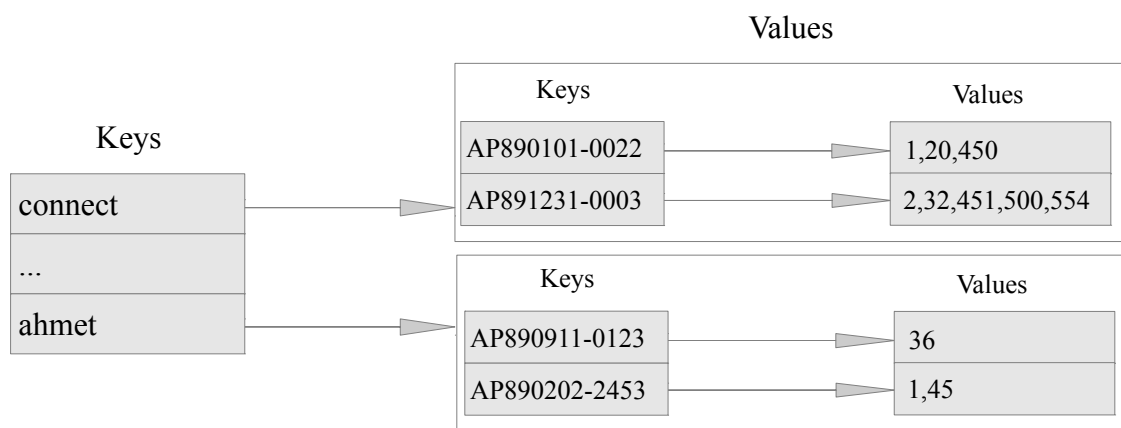
The inverted index cache is a Python dictionary which maps the **terms** against the **posting list**, e.g. The lists of documents in which they show up. This is sufficient for simple boolean retrievals.

All terms are lowercased before any processing happens to reduce redundancy. (Consequently, all queries will be lowercased before being pushed to the search engine.)

The terms starting with one of the '0123456789' characters were discarded to speed up the indexing and reduce the size of the index as they have little significance in the queries. Any leading and trailing punctuation characters are discarded, too.

The terms are then stemmed using the stem cache.

I've also added the support for **phrasal queries** which needs the positions of the terms in the documents in which they appear. This needs to calculate the position of the term in every document and store those positions in the inverted index, thus the values of the inverted index dictionary are dictionaries having the document numbers as keys and the relevant positions as values:



The final inverted index cache is a pickled dictionary with 159MB on-disk size.

3.4. SearchEngine Class

This is the main Python class responsible of doing all of the work. It has 8 public methods.

3.4.1. dump_cache(self, filename)

This method pickles and dumps the relevant Python dictionary to the file called filename.

3.4.2. load(self)

This method unpickles and loads the caches into the process' memory as Python dictionaries.

3.4.3. phrasal_query(self, query)

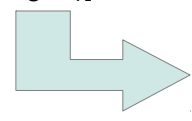
If the query is encapsulated between '!' or "...", the engine assumes that this is a phrasal query and passes it to this method.

We then search the **index cache** with the terms from the phrasal query, normalized and stemmed. Note that **every** term **must** hit a posting list in the index in the case of a phrasal query! So, if a term doesn't exist in the index, we can easily return and assume that no results have been found for the query.

After getting the posting list for every term, we find out the common document set by intersecting those posting lists. Thus, we obtain a list of documents which **all** contains the terms in the phrasal query. Now we can proceed to positional matching.

For each of the documents from the **common document set**, we take the *Cartesian Product* of every term's **position vector**. If there exists a product which contains **consecutive** position indexes, we can conclude that the document contains the phrase. If we formulate:

Let P the phrasal query <Ahmet Ertegun>,
Let CD the common document set containing both Ahmet and Ertegun,
For each document **d** from CD;
 P(Ahmet) <- (35) (Position vector for Ahmet in **d**)
 P(Ertegun) <- (1,26,36) (Position vector for Ertegun in **d**)
 For each product **p** from [P(Ahmet) X P(Ertegun)];
 if **p** is consecutive:
 d contains the phrase!



(35,1)
(35,26)
(35,36) ✓

3.4.4. search(self, query)

This method first hands out the query to `phrasal_query()` if necessary. If the passed query is not a phrasal one, it splits the query, searches for **&&** and **||** operators which denotes a boolean **AND** and a boolean **OR**.

Once the terms are splitted, the posting list for each term is fetched from the index and the relevant boolean operation is done over the posting list to reduce the result set.

Note: Python has a nice `set()` data structure which simulates a mathematical set: It doesn't hold redundant elements, it has `intersection()`, `union()`, etc. methods. This data structure is used whenever possible to not re-write a posting intersection/union algorithm.

3.4.5. create_document_cache(self)

This method first creates a multiprocessing pool. A multiprocessing pool is a worker process pool which adds concurrency to a Python program. Processes are used instead of threads because Python incorporates a *Global Interpreter Lock (GIL)*[5] which dramatically decreases threading performance.

This multiprocessing pool then calls the `parse_sgml()` helper function for every compressed **.Z** document concurrently.

The `parse_sgml()` helper function returns a dictionary and a set for every **.Z** file: A set of terms and a dictionary of document numbers against document texts. These two return values are cumulatively used to populate the main **document cache** and an intermediate set of **terms**.

Once the processing is over, the **document cache** is dumped onto the disk using the `dump_cache()` method.

3.4.6. `create_stem_cache(self)`

This method concurrently fetches chunk of terms, stems them using the *Porter Stemmer* and stores them in the **stem cache**.

Once the processing is over, the **stem cache** is dumped onto the disk using the `dump_cache()` method.

3.4.7. `clean_stop_words(self)`

Once the inverted index is created, this method traverses it to delete the keys if they are stop words.

Checking for each term whether it is a stop word or not during the inverted index creation is a very very slow operation and it involves checking a term again and again unnecessarily.

The stop words are determined using the `stopwords.txt` file given in the project archive.

3.4.8. `create_index(self)`

The method first ensures that the **document cache** and the **stem cache** are available. It then iterates over each document in the **document cache** and processes it according to the following algorithm:

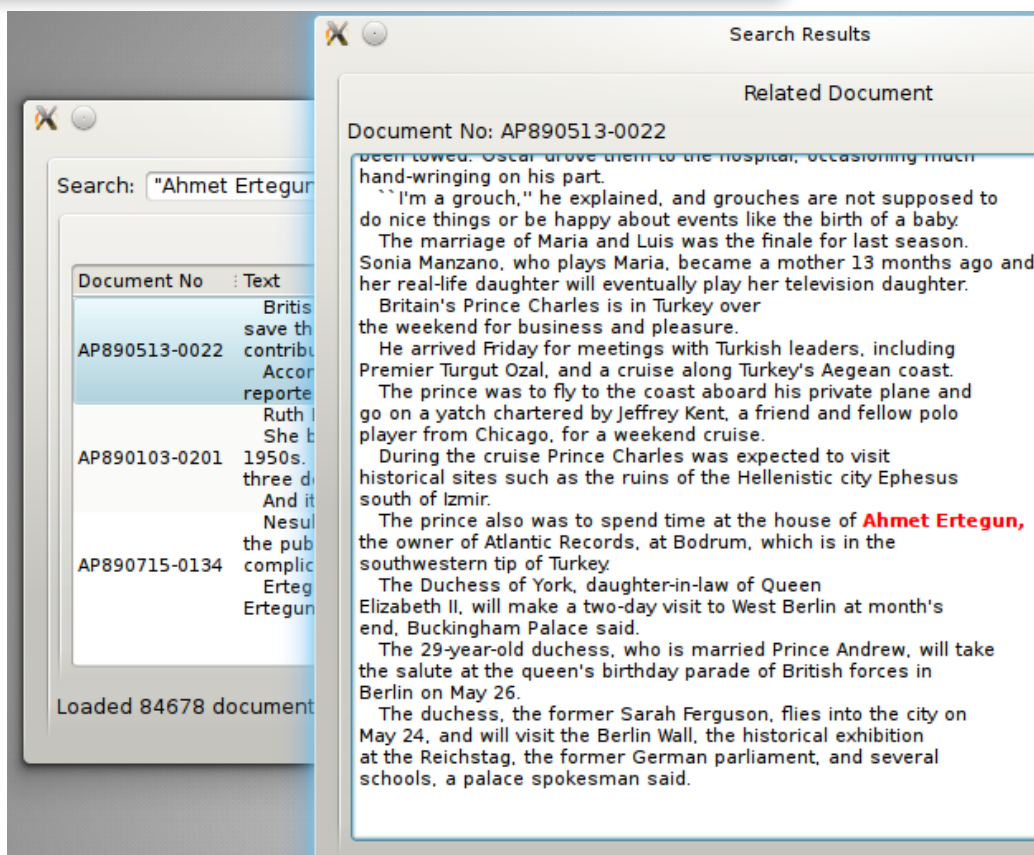
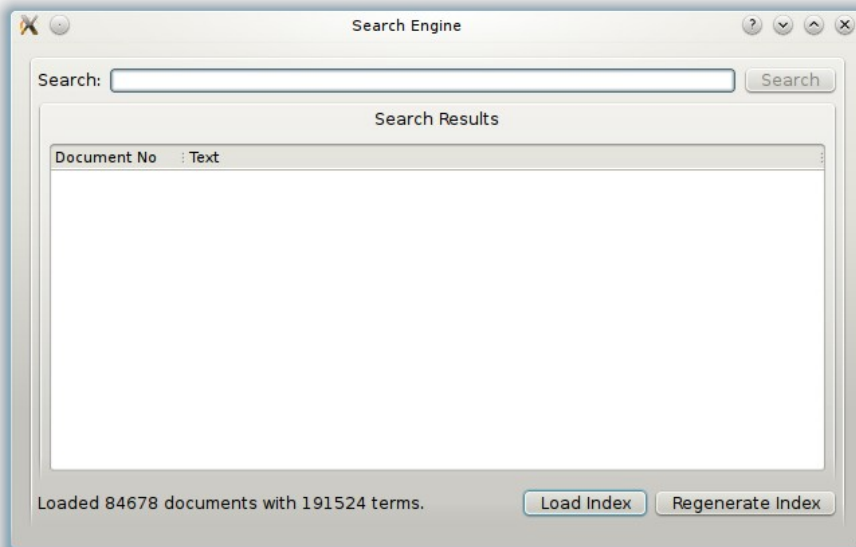
- Split the document by whitespace characters,
- Strip any leading and trailing **punctuation** characters, **lowercase** it and discard it if the first character is a **digit** or a **\$** sign.
- Find the **position** of the term in the document,
- Get the stem of the term from the **stem cache**,
- If the index doesn't have the stem, store the stem with an initial dictionary {docno: "pos"}
- If the index has the stem, store a new position:
 - for an existing document (*we're still in the same document*) **or**,
 - for a new document (*we've changed document but the stem has already been encountered for an earlier document*).

Once the processing is finished, `clean_stop_words()` is called to clean the stop words and finally the **inverted index** is dumped onto the disk using the `dump_cache()` method.

4. Graphical User Interface

The graphical user interface contains a textbox that the query should be entered and a result area in which every search result can be double-clicked to see the document in detail.

4.1. Screenshots



5. References

- [0]: <http://tartarus.org/martin/PorterStemmer/python.txt>
- [1]: <http://qt.nokia.com/>
- [2]: <http://www.riverbankcomputing.co.uk/software/pyqt/intro>
- [3]: <http://git-scm.com/>
- [4]: <http://docs.python.org/library/pickle.html>
- [5]: <http://docs.python.org/library/threading.html>