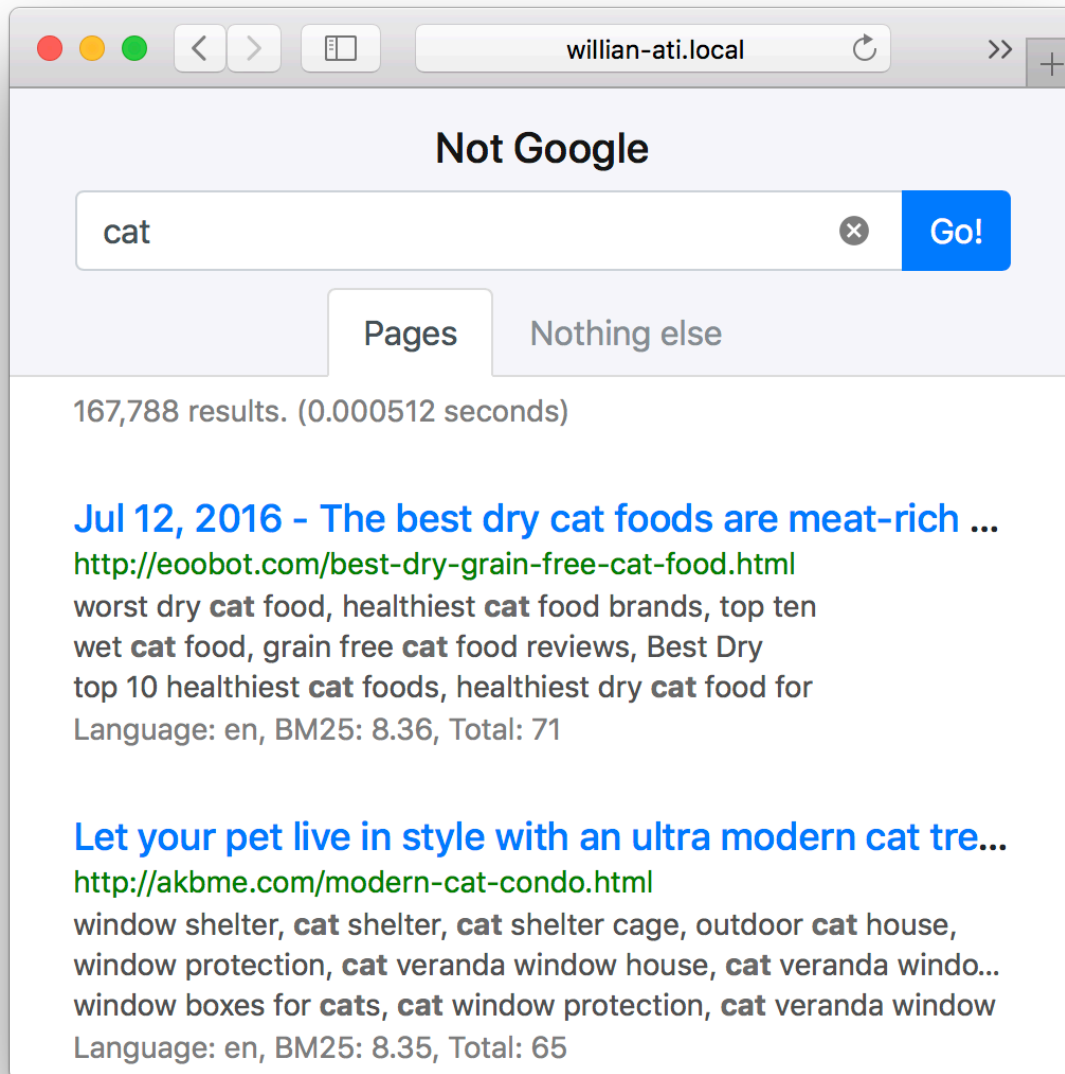


# Not Google

A Web Search Engine that is not google.



## Table of contents

- [Features](#)
- [How to run](#)
  - [Requirements](#)
  - [Installation](#)

- [Recommand for running](#)
    - [Virtual env](#)
- [Usage](#)
  - [Example usage](#)
    - [Download wet files](#)
    - [Lexicon extraction](#)
    - [Sort Merging](#)
  - [Actual usage](#)
  - [More options](#)
- [Notes on Running on Servers](#)
  - [HPC](#)
    - [Load Python 3 module](#)
- [Benchmark](#)
  - [Speed](#)
    - [Full mode](#)
    - [No Chinese mode](#)
    - [Dumb mode](#)
    - [Merging and Sorting](#)
  - [Size](#)
  - [Memory](#)
- [How it works](#)
  - [File Structure](#)
  - [Questions](#)
- [Future Work](#)
  - [Distributed](#)
  - [Speed up](#)
  - [Query optimization](#)
- [Development](#)

# Introduction

---

This project build a text based web search engine for a subset of the internet (~ 1/300 the living public crawlable internet). The purpose of a web search engine is to allow fast and effective information retrival on large amount of data that living on a web.

This project focuses on two main componant that makes a working text based web search engine: Inverted Index building and Query Processing. Inverted Index Building are futher devided into two stages: Lexicon Extraction stage and Merge & Sort stage.

# Features

---

## All

- Language detection

- 7 languages support including Chinese (English, French, Germany, Italian, Latin, Spanish, Chinese)

## Inverted Index Building

- Binary I/O and Storage
- Progress & Speed display
- Impact Score precomputation
- Compressed Inverted Index using Blockwise varbyte

## Query Processing

- Modern Web Browser Based Responsive UI (Desktop and Mobile)
- Conjunctive & Disjunctive Query
- Automatic Conjunctive or Disjunctive discovery
- Smart Snippet Generation
- Paging support
- Fast and low cost query based on Impact Score and memory efficient design pattern and index-guessing
- Speed up via Multilayer Cache Support
- Live module reload and cache usage report

# How to run

## Requirements

- Linux/ macOS/ OS with GNU tools
- Python 3.5+
- MongoDB
- Redis

## Installation

Please consider the [Recommand for running](#) section before Installation.

If you insist to install directly without virtual environment, It will be okay.

```
1 $ pip install -r requirements.txt
```

Notice you **have to** use the packages specified in `requirements.txt`, since there are some packages, though using the same import name in python headers, they use different packages than commonly used ones. Those packages are specialized for this project, and support for extra feature they originally don't have.

## Recommand for running

It is recommended to use virtual environment for python packages to avoid package conflicts.

### Virtual env

For the first time of for this project, start a new venv from as:

```
1 $ pyvenv .env
```

And then or for later use, activate it:

```
1 $ source .env/bin/activate
2 # For the first time:
3 $ pip install -r requirements.txt
```

## Use pre-built Inverted Index

### Files

There are 4 files that is required before Query process to run :

```
1 - docIDwet.tsv
2 - inverted-index-300-with-score.ii
3 - mongo-dump.zip
4 - redis-dump.rdb
```

### Download

These files are available at AWS for download:

```
1 https://s3.amazonaws.com/not-google/Inverted-Index/docIDwet.tsv
2 https://s3.amazonaws.com/not-google/Inverted-Index/inverted-index-300-with-score.ii
3 https://s3.amazonaws.com/not-google/Inverted-Index/mongo-dump.zip
4 https://s3.amazonaws.com/not-google/Inverted-Index/redis-dump.rdb
```

### Usage

Put `docIDwet.tsv` and `inverted-index-300-with-score.ii` into `./data/` dir as they are configured in `config.ini` by default. Unzip `mongo-dump.zip` and use `mongorestore` to restore into MongoDB. Put `redis-dump.rdb` into redis storage location and restart redis to load DB. Redis storage location can be found at:

```
1 $ redis-cli
2 > config get dir
3 1) "dir"
4 2) "/usr/local/var/db/redis" <- # Here
```

## Build Inverted Index

The running of the whole inverted index building has been divided into 3 parts:

- Download wet files
- Lexicon extraction
- Sort Merging

For the first **Lexicon extraction** stage, use python script `extract_lex`, and for **Sort Merging** stage use `merge.py`

### Example usage

#### Download wet files

```
1 $ ./scripts/dl.sh 300
```

This will download `100` wet files to `data/wet`. (change `100` to get more or less)

## Lexicon Extraction stage

```
1 $ python extract_lex.py --urlTable "data/url-table.tsv" data/wet/*.warc.wet.gz | sort > "data/all.lex"
```

This will extract all lexicons (that in language English, French, Germany, Italian, Latin, Spanish and Chinese) from the `wet` files in `data/wet/`, and write the sorted lexicons to `data/all.lex`.

## Sort Merging stage

```
1 $ cat "data/all.lex" | python merge.py > "data/inverted-index.ii"
```

This will read all **sorted** lexicons, merge them into inverted lists and write to `data/inverted-index.ii`.

## Actual usage

Example usage is not practical when you wants to:

- Run on many wet files
- Use binary for performance boost

So there are smarter version provided for these needs:

```
1 # extract all wet files in `data/wet`
2 $ ./scripts/extract-all.sh
3 * Dealing: data/wet/CC-MAIN-20170919112242-20170919132242-00000.warc.wet.gz
4 Building prefix dict from the default dictionary ...
5 Loading model from cache /var/folders/dy/dh2zyqj93fg72s9z4w2tnwy00000gn/T/jieba.cache
6 Loading model cost 0.828 seconds.
7 Prefix dict has been built succesfully.
8 40919records [05:00, 136.23records/s]
9 ...
10 $ ./scripts/merge.sh
```

`extract-all.sh` will individually extract and sort lexicons into fex files to `data/lex`.

`merge.sh` will take all **sorted** lex files and merge them into the final ii file `data/inverted-index.ii`.

All operations are done in binary.

## More options

More options over `extract_lex.py` can be fetched help:

```

1  $ python extract_lex.py -h
2  usage: extract_lex.py [-h] [-b] [-s <number>] [--skipChinese] [-T <filepath>]
3                          [--bufferSize <number>] [-u] [-c]
4                          <filepath> [<filepath> ...]
5
6  Extract Lexicons from WETs
7
8  positional arguments:
9    <filepath>            path to file
10
11 optional arguments:
12   -h, --help            show this help message and exit
13   -b, --binary          output docID as binary form
14   -s <number>, --startID <number>
15                        docID Assignment starting after ID
16   --skipChinese         if set, will not parse chinese words
17   -T <filepath>, --urlTable <filepath>
18                        if set, will append urlTable to file
19   --bufferSize <number>
20                        Buffer Size for URL Table Writing
21   -u, --uuid            use UUID/ if not specified, use assign new ID mode
22   -c, --compressuuid    compress UUID in a compact form, only valid in UUID
23                        mode

```

Note that `uuid` isn't tested for use. It was built for compatibility of distributed system.

## Notes on Running on Servers

Scripts are created for copying necessary executables to server. Use of example:

```

1  $ ./scripts/deploy.sh user@server:path

```

## HPC

Distributed version of this building program is not completed, you will not be able to use it on Hadoop or Spark or Hive. However you could use HPC as ordinary server to run the program.

There were works done for preparation of this program to be distributable. Please read [Future Work > Distributed](#) section.

## Load Python 3 module

```

1  $ module load python

```

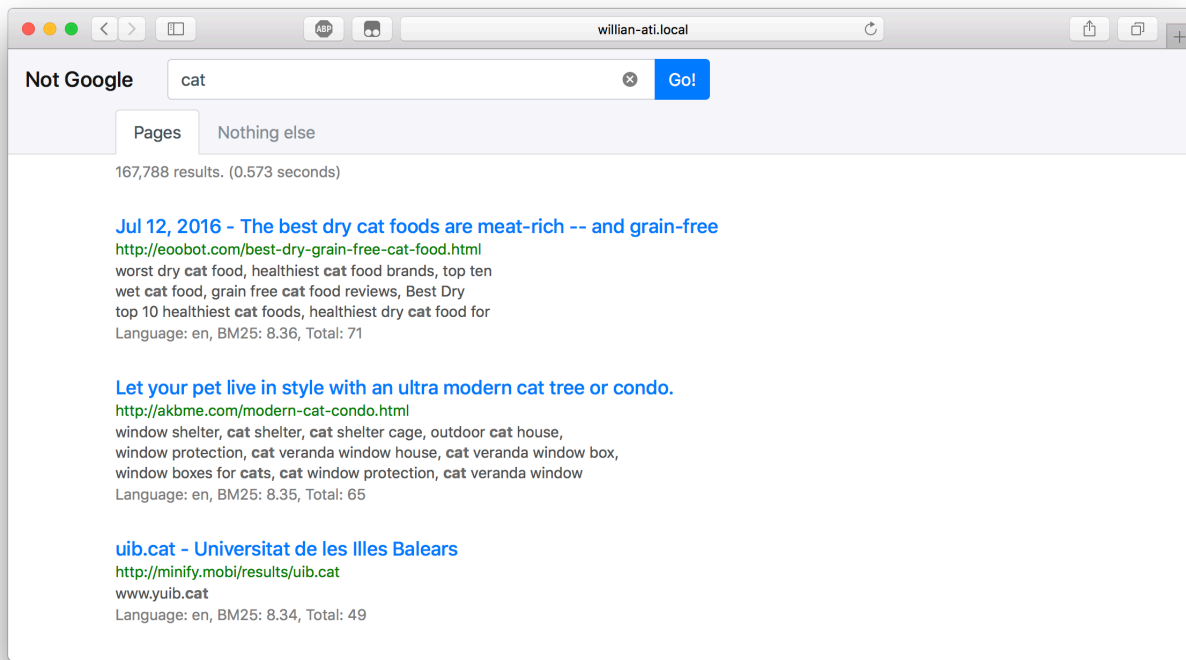
## HTTP Server

```

1  $ python query_http.py

```

Result:



## Benchmark

The following tests are done using Macbook Pro 2016 Laptop

### Speed

Sorting and merging speed are significantly low compared to `lexicon extraction`.

So the testing are mostly about `lexicon extraction`.

### Full mode

(Language detect on, Chinese on, binary)

```
1 $ python extract_lex.py --binary data/wet/* > "data/delete-this.log"
```

~ 136 records/s

~ 5 mins/wet

### No Chinese mode

(Language detect on, Chinese off, binary)

```
1 $ python extract_lex.py --binary --skipChinese data/wet/* > "data/delete-this.log"
```

~ 166 records/s

~ 4 mins/wet

## Dumb mode

(Language detect off, Chinese off, binary)

```
1 $ python extract_lex.no_language.py --binary data/wet/* > "data/delete-this.log"
```

~ 513 records/s

~ 1.3 min/wet

Speed is significantly faster however in this mode search result is going to be fairly bad, because all languages are jammed together. And for non-latin language it's even un-searchable.

## Merging and Sorting

```
1 $ ./scripts/merge.sh
```

~ **530k** lines/s (for input)

~ **52k** inverted lists/s (for output)

~ **13** s/wet (including Chinese words)

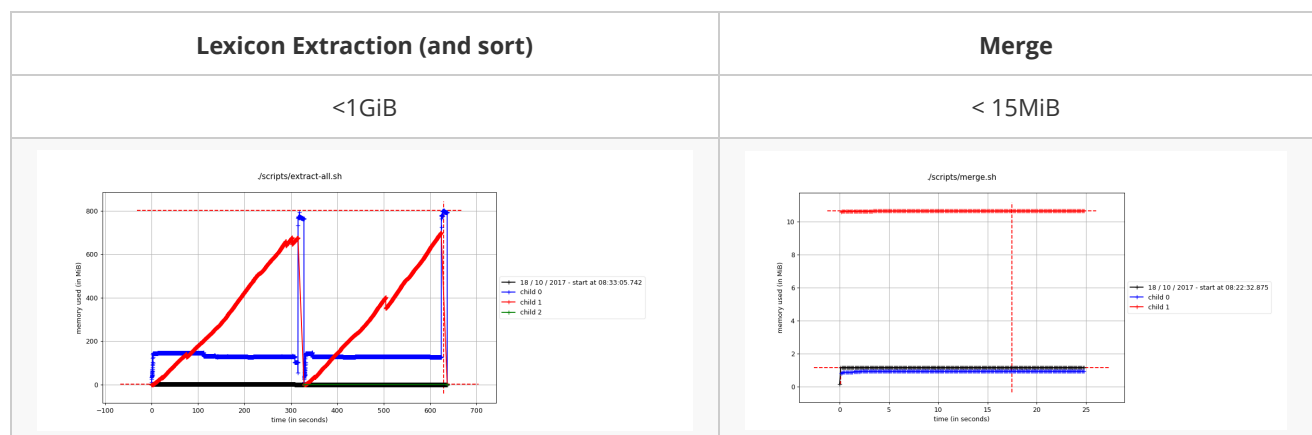
## Size

~ **15k** inverted lists/MB

```
1 #count line using:
2 $ cat "data/inverted-index.ii" | wc -l
```

## Memory

Depends on buffer size, for default



The memory usage are mostly used by GNU Unix Sort, by default GNU Unix Sort would take 80% of system, after that sort would use temporary file to store them.

Luckily modern computers has a memory typically much greater than 1GiB. So as long as the wet file size maintain as the current scale, this wouldn't be a problem.

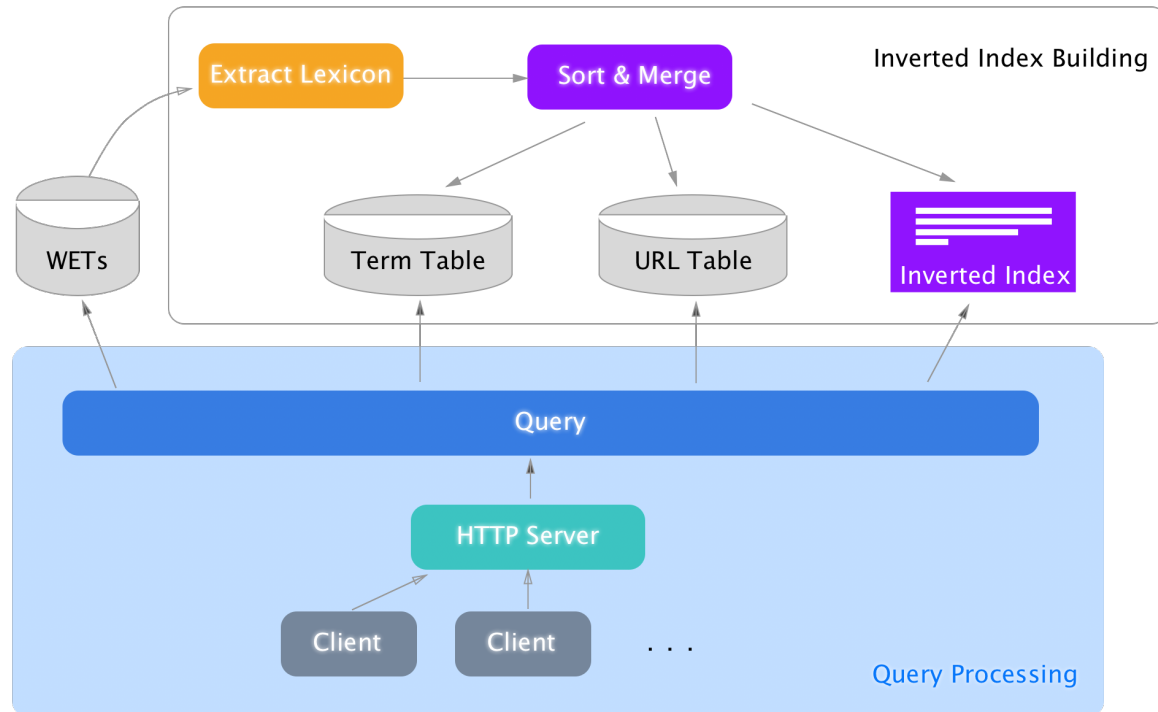


(In other words, on very low memory computers, it might slow down. )

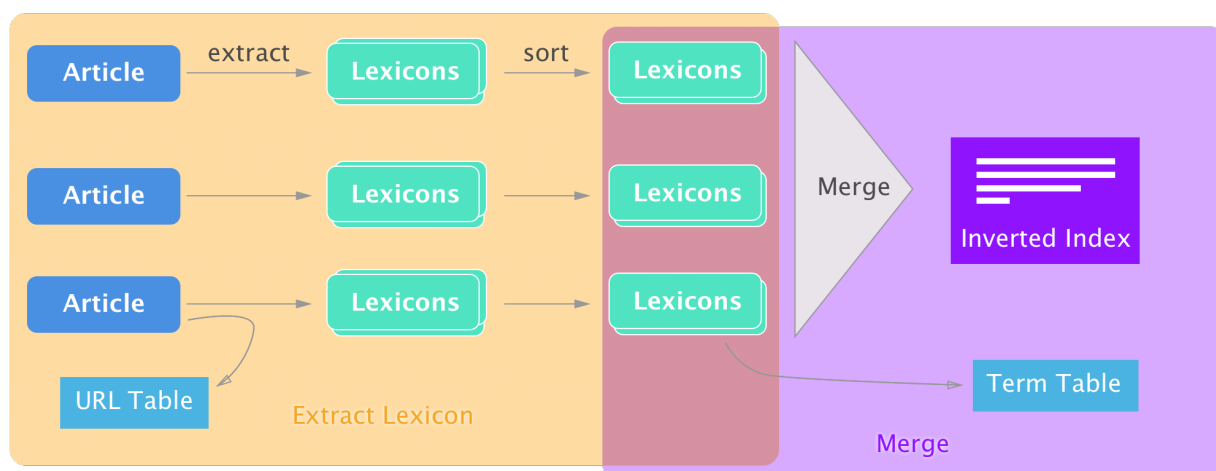
## How it works

## Structure

## Overall structure



Detailed **Inverted Index Building** Structure explaining what **Extract Lexicon** and **Merge** do.



## File Structure

```
1 | └─ README.md # This file source code
2 | └─ README.pdf # This file
```

```

3 |─ config.ini # Configuration file
4 |─ data/
5 |   |─ docIDwet.tsv      # (generated) File Index of WET
6 |   |─ expect_terms.tsv  # (generated) For process estimate of ii-build merge stage
7 |   |─ inverted-index.ii # (generated) final Inverted List
8 |   |─ lex/              # (generated, ii-build) intermediate lexicons Files
9 |   |   |─ CC-MAIN-20170919112242-20170919132242-00000.lex
10 |   |   └─ ...
11 |   |─ url-table-table.tsv # (generated, deprecated) Index of URL Table
12 |   |─ url-table.tsv      # (generated, deprecated) URL Table
13 |   |─ wet/              # (downloaded, uncompressed) WET Files
14 |   |   |─ CC-MAIN-20170919112242-20170919132242-00000.warc.wet
15 |   |   └─ ...
16 |   └─ wet.paths          # WET Files URLs
17 |─ decode-test.py         # (ii-build) Test code for lexicon file verification
18 |─ extract_lex.no_language.py # (ii-build) Dumb mode version of `extract_lex.py`
19 |─ extract_lex.py         # (ii-build) main file for `Lexicon Extraction`
20 |─ index.html            # client side index html
21 |─ merge.py              # (ii-build) main file for `Merge`
22 |─ miscellaneous/        # miscellaneous files
23 |   |─ ...
24 |   |─ deprecated/
25 |   |   |─ extract.sh*
26 |   |   |─ hadoop-test.sh*
27 |   |   |─ map-reduce-test.sh*
28 |   |   └─ setup-env.sh*
29 |   |─ dumbo-sample.sh*
30 |   └─ testbeds/
31 |       └─ ...
32 |─ modules/              # Modules
33 |   |─ BM25.py           # (query) BM25 computation
34 |   |─ BlockReader.py    # (query) High level Blockwized II Reader
35 |   |─ Heap.py           # (query) Customized Data Structures
36 |   |─ IndexBlock.py     # (query, ii-build) Blockwized II Reader and Writer
37 |   |─ LexReader.py      # (query) WET file reader
38 |   |─ NumberGenerator.py # (ii-build) binary compatible docID generator
39 |   |─ Snippet.py        # (query) snippet generation
40 |   └─ query.py          # (query) main Query process
41 |─ query_cmd.py          # (query, deprecated) cmd line interface for query processing
42 |─ query_http.py         # (query) HTTP interface for query processing
43 |─ requirements.txt      # requirement for python dependencies
44 |─ scripts/
45 |   |─ deploy.sh*        # (ii-build) helper script for deploy
46 |   |─ dl.sh*            # (ii-build) helper script for download
47 |   |─ expect_terms.sh*  # (ii-build) `expect_terms.tsv` generator
48 |   |─ extract-all.sh*  # (ii-build) main script for `Lexicon Extraction`
49 |   |─ extract-doc-parallel.sh* # (ii-build) parallel URL Table Rebuilder
50 |   |─ generate_toc.rb*   # (ii-build) helper script for markdown ToC generation
51 |   └─ merge.sh*         # (ii-build) main script for `Merge`
52 |─ static/               # Static files to server client
53 |   |─ lib/              # library used
54 |   |   └─ ...
55 |   |─ main.js           # main js file
56 |   └─ style.css         # main css file
57

```

# Query Efficiency

Several techniques has been used to speed up and reduce memery usage in query

- Precomputed impact score (see [Ranking>BM25](#)) has reduced live ranking score computation to 100x times on large query result
- Design pattern: Defered result (see [Design patterns](#))
- Index guessing for wet files

As wet file and docID grows, searching for a specific wet file may take longer time as for each query may call multiple times of them at least. Thus Index guessing are used to speed up the time to find which wet file are to read for a given docID.

Index guessing are done in very simple algorithm:

$$\frac{docID}{|WET|_{docIDs,avg}} \quad (156)$$

as docID are assigned in order of WET files.

## Ranking

### BM25

Result to return for a given query are mainly base on ranking BM25.

$$BM25(d, Q) = \sum_{t \in Q} IDF(t) \times \frac{f_{d,t}}{K_d + f_{d,t}} \times (k_1 + 1) \quad (106)$$

where

$$IDF(t) = \log \frac{N - n_t + 0.5}{n_t + 0.5} \quad (81)$$

and

$$K_d = k_1 \times (1 - b + b \times \frac{|d|}{|d|_{avg}}) \quad (149)$$

### Precomputed impact score

We can easily know that  $\frac{f_{d,t}}{K+f_{d,t}}$  can be precomputed.

Since  $\frac{f_{d,t}}{K+f_{d,t}} \in [0, 1)$ ,

In order to store them in ii file efficiently, linear quntization is used:

$$\left\lfloor c \times \frac{f_{d,t}}{K + f_{d,t}} \right\rfloor \quad (150)$$

where  $c = 2^{(8-1) \times 2} = 16384$ .

Chosing  $c = 2^{(8-1) \times 2}$  dues to support for large resolution and varbyte encoding in max of 2 bytes.

A better solution would be  $c = 2^8 = 256$  to use a smaller resolution and fix-byte encoding as they would store always in 1 byte, however decision has been made and precomputation is running at the time it has to be done for later stages.

## Design patterns

---

In query processing, the following design patterns are commonly used to achieve efficiency in both time and memory.

### List Comprehension

This is a concept that is very close to lambda expression in many other languages than python.

When it's used, variable construction time among with other replicative costs are reduced to improve overall execution speed.

### Deferred result

In python, keyword `yield` create a `yielding` object that may defer result from where it's called. In combination with list comprehension and iterable, it would produce the real result only on final iteration.

A dedicated data structure `FixedSizeHeap` is commonly used in this project to catch deferred result, as the memory would be limited to the size of the Heap, as well keeping an ordered result for certain priority (think of BM25).

## Questions

---

### Inverted Index

Why inverted index building so slow?

Because Language Detect and Chinese Word Separate uses HMM model (pre-trained) to compute. They are computational intensive.

How much docIDs are supported? Why?

In short: ~2 billion.

The binary encoding entropy used for docID generation was  $(256 - 3) = 253$  out of 256 per byte.

The number of encoding bytes are chosen for 4 as default so there would be  $(256 - 3)^3 \times (128 - 3) - 2 = 2,024,284,623 \approx 2 \text{ billion}$  documents supported.

Why -3 ?

The -3 was for `\t` `space` and `\n` 3 different kinds of separation characters. Those characters are used to separate words document IDs frequency and future added features.

Compare with plain text number entropy 10 out of  $2^8 = 256$ ,  $2^8 - 3$  is a much better figure.

Why unix sort?

Unix sort is incredibly fast and supports streaming.

The encoding had been paid much care to support using unix sort.

Is Unix Sort ok?

They are, as long as you treat the stream as binary, I set flag `LC_ALL=C` to do that.

Why encode docID in binary form not Text Frequencies (IF)?

Because **most** TF are below 10. They take up 1 byte to store. Thus using binary form for it **won't benefit** much.

Hence designing and **computing** each time when accessing an encoding that convert to and from number would both take more time.

Why Merge Stage take so little memory ?

Beacuse the design of `merge.py` has take as much advantage of streaming as possible.

It doesn't wait till an Inverted List is completed to unload memory, it streams out all `doc Item` as long as they get them.

## Query

## Other

Why not C++

Beacuse packages on high level langauges are richer than C++.

If effieency is the convern, there are language that are close to C++ level of proformance e.g. Swift, Go

## Other open source library

---

This project had been use a modified python package `warc` . It has been modified to support reading and fast lookup `wet` files to adapt to this project (check [warc3-wet](#) ).

During this project , a request has been made to fix an deprecation problem. (check [Pull Request](#) (Japanese) )

## Future Work

---

There are several works can be done easily but requires more careful thoughts

## Distributed

---

### UUID approach

The whole program is written in a `Map and Reduce` concept. They can be easily ported to Hadoop MapReduce. Here is a list of what has been done:

- Python package distribution with virtual env support using Hadoop
- UUID and UUID compression support (higher entroy encoding for UUID)

And what to be done is:

- Map Reduce compatiple URLTable Generation
- Glue code to pipe them all

Considering Hadoop Stream isn't actually efficient, Spark would be a good replacement for that, though how to port `Language Detection` and `Chinese Support` to Scala and Spark.

### Multi Stage Approrach

An parallel version of Lexicon Extracion has been implemetend based on previous result of `docIDwet.tsv` .

It speed up 5x to the previous run.

This gives a hint on splitting current Lexicon Extracion to 2 parsing stages for runnning: One for generating docID. The other for distributed detailed care taken Lexicon Extraction for each WET file, for a given starting number of docID.

## Speed up

---

Change a language like `Go` might incredibaly speed up exection. (But packages?)

## Query optimization

---

`IDF` calculations may also be pre-computed.

## Development

---

Add new requirements if new python packages are used

```
1 | $ pip freeze > requirements.txt
```

If to Change of README.md file. There is a ruby script to build Markdown Table of Content:

```
1 | $ ruby scripts/generate_toc.rb
```