UNIVERSITÀ DI PISA

MULTIMEDIA INFORMATION RETRIVAL AND COMPUTER VISION

Search Engine

Supervisor
Nicola Tonellotto

Student
Alessandro Versari

Contents

Chapter 1

Introduction

This project is a client based search engine which given a collection of documents is able to parse it, to create an index data structure, and then to execute document retrieval on the index itself.

Currently the project is capable of handling only the passenger MS-MARCO collection, but little adjustment to the `spimi.ChunkReader` implementation would make it able to parse any local collection. The source code of the project is available at: https://github.com/just-hms/pulse

The project is named `pulse`, symbolizing its intended speed and efficiency. Throughout this document, any reference to `pulse` are about the executable name of this project.

1.1 Folder structure

The project is organized into a well-structured folder hierarchy to facilitate development, testing, and documentation. Below is a detailed description of the folder structure:

- **cmd/**
  Contains the main entry points of the application, which will be used by the end user:
  - `root.go` - Root command for the CLI.
  - `search.go` - Handles the search functionality.
  - `spimi.go` - Implements SPIMI indexing logic.

  each of the file represent a separate command which the user can execute, each command has its own set of available flags.

- **preprocess/**
  Contains top level preprocess api's used during indexing
- **config/**
  Contains configuration-related files
- **docs/**
  Documentation for the project
- **pkg/**
  Core Package Implementations, including Compression, Preprocessing, Queue Handling, and Indexing:
  - `compression/` - Contains compression algorithms.
  - `engine/` - Search engine core logic, metrics and settings.
  - `spimi/` - Implements the SPIMI algorithm and associated utilities, contains the `msMARCO` chunk reader.
  - `structures/` - Data structures such as heaps, radix trees, and sets used in the project.
  - `word/` - Utilities for stopword removal, Unicode handling, and word processing.

1.1.1 Testing

Following the Go's testing principles, each unit(ish) test is right next to each source file, for example the test for the `spimi.go` is in the same folder and is called `spimi_test.go`

Chapter 2

Usage

The basic e usage of the client interface consists of:

Listing 2.1: Basic usage

```
# indexing
tar xOf collection.tar.gz | pulse spimi

# one-shot search
pulse search -q "what is shakespeare's theatre called"
# interactive search
pulse search -i
# file queries
pulse search -f queries.tsv -k 1000 > results.tsv

# evaluation
trec_eval -m all_trec qrels-pass.tsv results.tsv
```

Parameters such as: `--no-compression`, `--no-stemming`, `--no-stopwords-removal` can be passed to the `spimi` commmand, others like `-k/--doc2ret`, `-m/--metric` (the score metric to use betweeen `TFIDF` and `BM25` and `--conjunctive` can be passed to the `search` command.

Chapter 3

Internals

3.1 Document Processing

Each document undergoes the following steps to extract term information:

1. **Convert content to lowercase**: All text is converted to lowercase for uniformity.
2. **Clean the content**: This step consists of:
   (a) Unicode normalization[1]
   (b) Removal of HTML tags
   (c) Removal of punctuation
3. **Tokenization**: The text is split into individual tokens or words separating using whitespace.
4. **Optional steps**: Depending on the configuration, the following additional steps may be applied.
   (a) Stopwords removal
   (b) Stemming

3.1.1 Compressed reading

The compressed reading is possible via piping, for example:

```
tar xOf data/dataset.tar.gz | pulse spimi
```

[1] https://go.dev/blog/normalization

3.2 Indexing

The indexing is computed using the Single-Pass In-Memory Indexing algorithm which is implemented inside its ononymous package. There are two main components which handle the index generation: the method `.Parse()` of the `spimi.builder` object and the `spimi.Merge` function.

3.2.1 Spimi

The first step of the spimi package is responsible to ingest the corpus, parse the document and encode the partial index information into files once a user defined memory threshold is reached. The document are parsed by the `.Parse()` method using an injected `spimi.ChunkReader` object which is responsible for reading documents from the corpus (`--chunk-size` documents at a time). The documents are then parsed by a set of workers using the one-producer multiple consumer paradigm.

Once the memory threshold is reached, the index is dumped into a set of files which contain the `documentIndex`, the `postingList`, the `Frequencies` and the `Vocabulary`.

Statistical information about the corpus is also extracted during this phase and are encoded into a file named `stats.bin`

3.2.2 Merge

The merge function is responsible to build a vocabulary containing global information about the terms inside the provided corpus. The posting list and frequencies are not merged. This design implementation was chosen to perform each query in parallel over the document partitions.

Each term information is composed by the `DocumentFrequency`, the `MaxTermFrequency`, `FrequenciesStart`, `FrequenciesLength`, `PostingStart` and `PostingLength`, everyone of this field is store inside an unsigned 32bit integer.

3.3 Query execution

The algorithm chosen implement the query executuin is the DAAT (Document At A Time). Both TFIDF and BM25 scoring metrics are implemented, and both of them are available in Conjunctive and Disjunctive mode.

The query execution is carried out by the `engine` package which mainly contains the index loading logic and the query execution logic.

3.3.1 Index loading

At the start of the `search` command all of the necessary file descriptors are open, for fast sparse read access to the disk `mmap` is heaviky used, these components are mapped in memory: `Posting lists`, `Frequencies` and `Term information`. The `Vocabulary` is pre-loaded in memory using a `radix` tree data structure which contains the association `term -> pointer to the term's information in the disk` which is a reference to where in the term's information file the information are actually stored.

3.3.2 Actual query execution

Given a user provided query its content pass through the same pre-processing of the dataset corpus. `Stemming` and `StopWords Removal` options must be set at indexing time and are inherited from the `spimi` command's configuration.

Once the query has been processed and the token are extracted the terms information are retrieved using a radix data structure which contains the term -> pointer associations.

Each query is processed separately in each document partition, then each result of each partition is merged into a single global one, which is then returned to the final user. An heap data structure is used to keep track of the first `k` documents to return, every time a possible document is compared against the worst one which is contained in the root of the heap, in this way the time complexity to check if a document must be insert is O(1).

After the merging of the results the heap data is extracted and returned as a sorted list after reversing it.

3.3.3 Disjunctive Conjunctive

Conjunctive and Disjunctive query are both possible. The default behaviour is Disjunctive, but changing it is as simple as providing the search command a flag named `-conjunctive`.

3.4 Compression

Compression is opt-out and is used to write down to file two key components of the index: the posting list and the frequencies.

For the `posting list` the `variable-bytes` compression algorithm is used to encode the delta between each of the document inside a term's posting list. Storing the deltas between each posting element is crucial because it reduces the dimension of the

numbers to be stored using `variable-bytes` further reducing the memory footprint. The representation of the i-th element is as follow

$$d_i = \begin{cases} VariableBytes(\text{docID}_i), & \text{if } i = 0 \\ VariableBytes(\text{docID}_i - \text{docID}_{i-1}), & \text{if } i > 0 \end{cases}$$

For the `frequencies` the `Unary` compression is used. This choice is justified because `Unary` performs exceptionally well for small numbers, requiring only 1 bit for a frequency value of 1. Since frequency values in postings lists are almost always 1, this approach achieves high compression efficiency.

The implementation of the compression algorithms can be found inside the `compression` package.

Chapter 4

Evaluation

All of the evaluation phase can be run using the `./docs/run.sh` script which:

- builds the index using different configurations
- print out information about the indexing time and disk usage
- execute queries using different search configuration and scoring metrics
- prints out the information about the queries execution time and some quality measure using `trec_eval`

4.1 Index

In the 4.1 table we can o the disk usage of the index data structure at the end of the indexing phase, each row represent a different configuration given by the end user.

Configuration Name	**DocIDs**	**Frequencies**	**Vocabulary**
No-Preprocessing	453MB	63MB	163MB
No-compression	901MB	901MB	139MB
Normal	310MB	40MB	138MB

Table 4.1: Indexing Disk Usage

The 4.2 table contains execution time of the indexing phase depending on predefined user configuration.

Configuration Name	Indexing Time
No-Preprocessing	3m26.589s
No-compression	2m34.998s
Normal	2m44.135s

Table 4.2: Indexing Time

4.2 Query execution time

The following tables are generated using the `TREC DL 2020 queries` and `TREC DL 2020 qrels`. The index used in every execution (a part from the stemming one) uses all the features available (stemming, stopwords removal and compression)

The 4.3 table shows the average elapsed time of the queries and the encountered standard deviation.

Configuration Name	Avg Query	σ Query
Conjunctive-TFIDF	21.433ms	23.1051ms
Conjunctive-BM25	21.7478ms	23.6765ms
Disjunctive-TFIDF	49.8264ms	47.4329ms
Disjunctive-BM25	49.0416ms	45.9393ms

Table 4.3: Query Execution Times

4.3 Quality measures

The tables below present common retrieval quality measures, extracted from the output of the `trec_eval` tool. The tool was executed for all configurations available in the search engine to assess its performance.

Configuration Name	p@5	p@10	r@1000
Conjunctive-TFIDF	0.5955	0.4750	0.2212
Conjunctive-BM25	0.5955	0.4636	0.2212
Disjunctive-TFIDF	0.5074	0.4667	0.6703
Disjunctive-BM25	0.6333	0.5759	0.7540

Table 4.4: Precision and Recall Metrics

Configuration Name	ndcg_cut@10	map_cut@10
Conjunctive-TFIDF	0.4243	0.1235
Conjunctive-BM25	0.4275	0.1219
Disjunctive-TFIDF	0.3775	0.0990
Disjunctive-BM25	0.4890	0.1411

Table 4.5: NDCG and MAP Metrics

4.3.1 Stemming

The stemming functionality, though not necessarily enhancing precision, is expected to improve the generalization of the index. To evaluate this, an additional benchmark was conducted to compare recall performance with and without stemming.

Configuration Name	r@1000
Disjunctive-BM25	0.7540
Disjunctive-No-stemming-BM25	0.6927

Table 4.6: Recall Metric Using Stemming or not

Chapter 5

Limitations

- **Conjunctive query execution:** The current optimizations for conjunctive query execution include are: if a term is not present in a partition, the result of the partitition is set to empty and early-returned ; if one or more terms in the query are not found in the document being processed, the metric score is not computed, and a score of 0 is directly assigned. No further improvement has been made. For example a performance improvement could be to order the posting lists from the smallest (in document ID size) to the biggest so that once the first posting list is terminated the query can be early-stopped.
- **Max Score Implementation:** Currently, the Max Score optimization has not been implemented. Adopting this feature would require adjustments in how posting lists are written. In the current implementation, if compression is enabled, only document ID deltas are stored to reduce dimensionality and improve the performance of variable byte encoding. This approach is still viable but delta must be calculated inside blocks (this is needed to implement the `nextGEQ` function).
- **Document Partitioning:** The number of document partitions created during indexing is determined by the user provided `memoryThreshold` parameter, which dynamically affects the number of partitions created. Ideally, the number of partitions should be specified as a configurable argument for the program.
- **Scalability for Larger Datasets:** The vocabulary is loaded entirely into memory, leading to an increase in memory usage as the corpus size grows. Although a space-optimized trie is utilized, providing efficiency for the relatively small dataset used, this approach may face limitations with larger datasets. Exploring alternative data structures such as B+trees would be beneficial to improve scalability while maintaining performances.