Information Retrieval & Web Search Project Report
Clustering Documents to Compress Inverted Index

Francesco Vinci 868164

2023/2024

1 Introduction

To search for words in a text, with the power of today's computers, it would be enough to perform a linear search on the text. But if we want to analyze large documents we need more efficient strategies that have an adequate cost/benefit ratio. So we cannot simply increase the computing power of our machines as this could come at very high cost.

The way to avoid linear scanning of a document is to index them beforehand.

In a binary term-document incidence matrix we will have a two-dimensional matrix with rows and columns, the terms are the indexed units, and depending on whether we look at the rows or columns of the matrix, we can have a vector for each term, showing the documents in which it appears, or a vector for each document, showing the terms that occur in it.

This matrix is extremely sparse, that is, it has few non-zero elements.

A better representation is to record only the 1, inverted index positions. We maintain a dictionary of terms. So, for each term, we have a list that records the documents in which the term occurs.

2 Index Construction

The design of indexing algorithms is governed by hardware constraints.

Blocked sort-based indexing has excellent scalability properties, but requires a data structure to map terms to termIDs. For very large collections, this data structure does not all fit in memory. A more scalable alternative is single-pass memory indexing, or SPIMI. SPIMI writes each block's dictionary to disk and then starts a new dictionary for the next block. SPIMI can index collections of any size as long as sufficient disk space is available.

SPIMI-INVERT is called repeatedly on the token stream until the entire collection has been processed.

The tokens are processed one by one during each subsequent call of SPIMI-INVERT. When a term occurs for the first time, it is added to the dictionary and a new posting list is created.

Algorithm 1 SPIMI-INVERT

```
1: procedure SPIMI-INVERT(token_stream)
2:     output_file ← NewFile()
3:     dictionaty ← NewHash()
4:     while free memory available do
5:         token ← next(token_stream)
6:         if term(token) ∉ dictionary then
7:             postings_list ← AddToDictionary(dictionary, term(token)
8:         else postings_list ← GetPostingsList(dictionary, term(token)
9:         end if
10:        if full(postings_list then
11:            postings_list ← DoublePostingsList(dictionary, term(token)
12:        end if
13:        AddToPostingsList(postings_list, docID(token))
14:    end while
15:    sorted_terms ← SortTerms(dictionary)
16:    WriteBlockToDisk(sorted_terms, dictionary, output_file)
17:    return output_file
18: end procedure
```

3 Postings Compression

To create a more efficient representation of the posting lists, we observe that the potings for the frequent terms are close to each other.

If we scroll through the documents in a collection one by one and looking for a frequent term, we will find a document containing that term, then we will skip some documents that do not contain it, then there will be a document with the term again and so on.

The key idea is that the intervals between postings are short and require much less space.

In fact, the gaps for the most frequent terms like “the” and “for” are mostly equal to 1.

But gaps for a rare term that occurs only once or twice in a collection need more space.

For an economical representation of this gap distribution, we need a variable encoding method that uses fewer bits for short gaps.

3.1 Variable byte codes

Variable byte (VB) encoding uses an integer number of bytes to encode a gap.

The last 7 bits of a byte are “payload” and encode part of the gap. The first bit of the byte is a continuation bit. It is set to 1 for the last byte of the encoded gap and 0 otherwise.

Compression is very easy. For a number G we use, in bit:

$$\left\lceil \frac{\lfloor \log_2 G \rfloor + 1}{7} \right\rceil \cdot 8$$

3.2 Gamma code

We can encode posting lists with different ”alignment units”. A somewhat bizarre strategy is the unary code, where we encode a number as n times 1 with a 0 at the end. This doesn’t look

promising with high numbers, but we can use it with elias gamma code. We encode the number with a pair: $\gamma - code$(length, offset)

- **offset** is the binary coded number with the head bit removed
- **length** is the number of bits used to represent the offset encoded in unary

Given a number G is encoded using $2\lfloor \log_2 G \rfloor + 1$ bits

3.3 Delta code

$\gamma - codes$ are relatively inefficient for large numbers: coding for the offset length is inefficient. $\delta - code$ encodes the offset length in $\gamma - code$ instead of unary code, while the offset encoding is the same.

To represent a number G, Elias delta uses, in bits:

$$\lfloor \log_2 G \rfloor + 2\lfloor \log_2(\lfloor \log_2 G \rfloor + 1) \rfloor + 1$$

4 Clustering

Clustering algorithms sort a collection of documents into groups called clusters. The goal is to make each cluster internally similar, but different from other clusters. In other words, documents in the same cluster should be very similar to each other, while documents in different clusters should be quite different.

Starting from an inverted index we can group similar documents together using the Jaccard Distance as a method to represent the "similarity".

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Where A and B are two sets that contain the documents in which term A and B appears.

Therefore it is necessary to modify the inverted index into a structure that facilitates these calculations. The ideal structure is a forward index, which it maps each document to a list of terms.

Before moving forward it is important to introduce the concept of medoid. In clustering, medoids are central points of a cluster that represent the center of the cluster itself. Unlike centroids, which are calculated as the arithmetic mean of all points in the cluster, medoids are actual data points of the dataset. The medoids are chosen in such a way that the total distance between the medoid and all other points in the cluster is minimized, we also use the Jaccard distance in this case to represent the distance between points in the cluster and the medoid.

The best-known algorithm that uses medoids is k-means. In this algorithm:

1. k random medoids are initially selected.
2. Each point in the dataset is assigned to the closest medoid, thus forming k clusters.
3. The new medoids are calculated for each current cluster, choosing the point that minimizes the sum of the distances to all the other points in the cluster.
4. The steps are repeated until the medoids no longer change.

For a matter of computational time we will do nothing more than select a **random** medoid, in particular the first element inserted in the cluster

5 TSP: Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a classic problem in optimization and computer science. It focuses on finding the shortest possible route that visits a set of cities exactly once and returns to the starting city.

In practice, solving TSP for a large number of documents directly is computationally expensive. One approach is to first cluster the documents into smaller groups using clustering techniques and then solve TSP for each cluster individually.

In this case it can be useful to find the path that minimizes the distance between the medoids.

So, given a dictionary of metodids `{medoid: [docID1, docID2, ...], ...}` we can apply a greedy approach of the TSP algorithm using the jaccard distance to measure the distance between two medoids.

Below we see the project code to solve the TSP problem.

```
1
2  def solve_tsp(medoids, forward_index_set):
3      start = medoids[0]
4      tour = [start]
5      medoids_list = medoids[1:]
6
7      current = start
8      while medoids_list:
9          next_medoid = min(medoids_list, key=lambda medoid:
           ↪  jaccard_distance(forward_index_set[current], forward_index_set[medoid]))
10         tour.append(next_medoid)
11         medoids_list.remove(next_medoid)
12         current = next_medoid
13
14     return tour
15
16 def apply_tsp_to_medoids(forward_index, clusters):
17     medoids = [medoid for medoid, _ in clusters]
18     forward_index_set = {doc: set(terms) for doc, terms in forward_index.items()}
19     tsp_tour = solve_tsp(medoids, forward_index_set)
20     return tsp_tour
21
```

6 DocIDs Reassignment

To assign document IDs linearly, cluster by cluster, using TSP-induced ordering between medoids, we can proceed as follows:

- We order the cluster medoids according to the TSP order.
- We assign IDs to documents in each cluster based on TSP order.
- Within each cluster, the order of documents can be arbitrary.

In this way, each document gets a unique ID assigned linearly, respecting the order of medoids determined by the TSP.

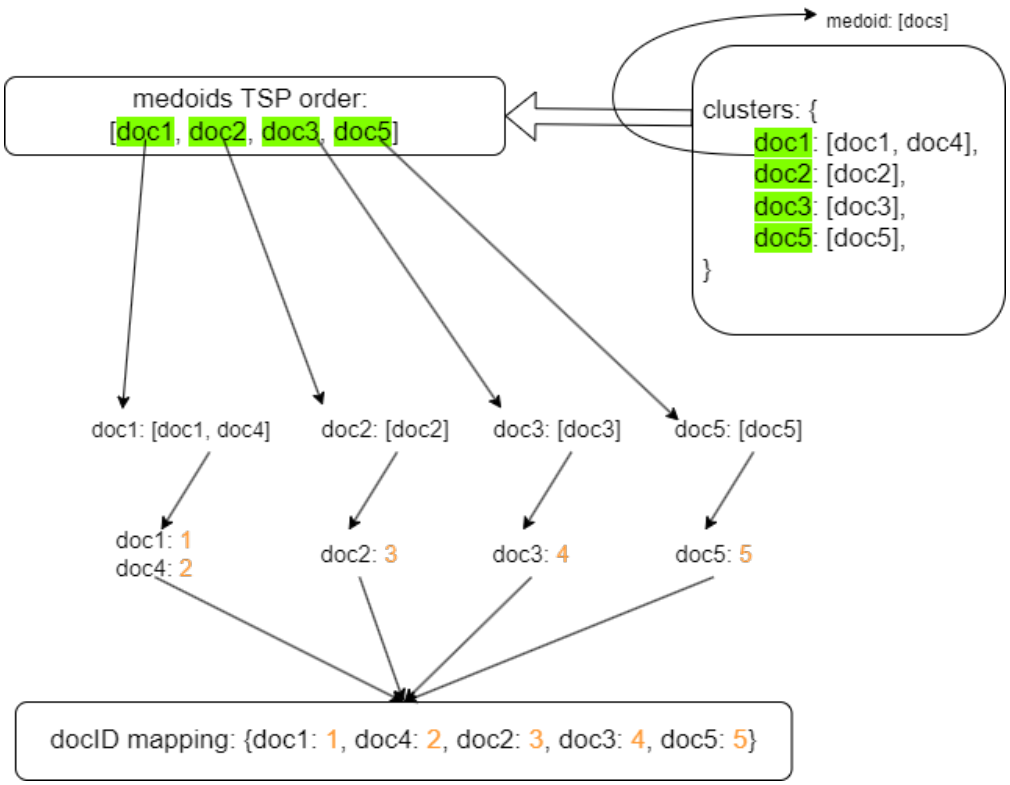

Figure 1: docID reassignment process

```
1
2  def assign_doc_ids(tsp_tour, clusters):
3      doc_id = 1
4      doc_id_map = {}
5
6      medoid_to_docs = {medoid: docs for medoid, docs in clusters}
7
8      for medoid in tsp_tour:
9          docs = medoid_to_docs[medoid]
10
11         for doc in docs:
12             doc_id_map[doc] = doc_id
13             doc_id += 1
14
15     return doc_id_map
```

For each posting list, we need to update the documents based on the new mapping-based IDs. We can do this very easily thanks to the dictionary that maps the docIDs to the new docIDs that follow the TSP order.

```
1
2  def reassign_postings(postings, doc_id_map):
3      new_postings = {}
4      for term_id, docs in postings.items():
5          new_postings[term_id] = [doc_id_map[doc] for doc in docs]
6      return new_postings
```

7 Evaluation

We use a portion of the RCV1 Dataset with 4,384,612 unique terms and 398,667 documents.

To see how compression improves we need to make a comparison. So we do a first compression phase on the initial inverted index. Calculating the d-gaps and compress them using VB, $\gamma - code$ and $\delta - code$.

Table 1: Avg. bits without TSP clustering

Encoding	Avg. bits
VB	9.5889
Elias gamma	8.8771
Elias Delta	8.3056

This will give us an idea of how many bits on average are used to encode d-gaps. From this data we can base a comparison that allows us to observe the improvements.

So starting from the inverted index we perform the clustering obtaining a medoid for each cluster, then we apply the TSP on the medoids and finally we reassign the docIDs according to the TSP order.

With this procedure now the documents that are most similar to each other, in addition to belonging to the same cluster, will also have docIDs that are very close to each other, minimizing the necessary quantity of bits to store them.

We use different values for radius to observe how compression and execution time changes based on the number of clusters.

We can see the radius as a threshold of belonging to a cluster. In fact, a document that wants to enter a cluster must have a Jaccard distance, with respect to the set of terms of the medoid, that is less than or equal to the radius.

Table 2: Avg. bits with TSP clustering

Radius	0.94	0.95	0.96	0.97	0.98	0.99
Num. of clusters	1420	715	351	173	71	29
VB	9.25	9.2942	9.3359	9.3746	9.4049	9.4441
Elias gamma	7.648	7.8011	7.9588	8.0839	8.1743	8.3043
Elias Delta	7.249	7.3824	7.5184	7.626	7.7022	7.813

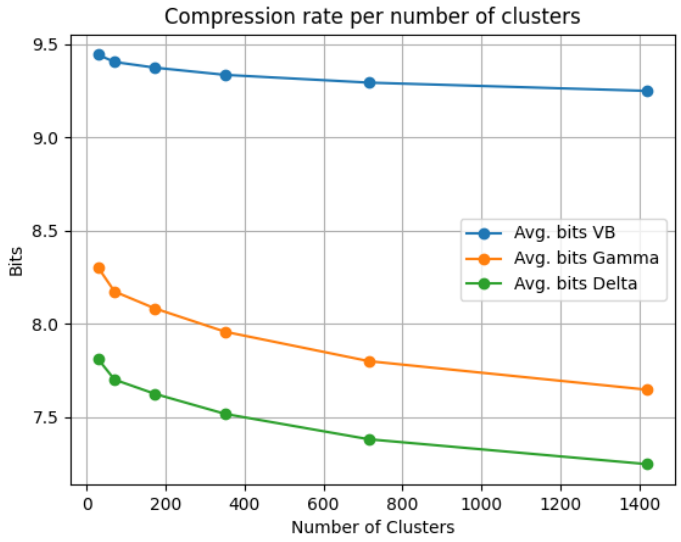

Figure 2: Compression rate (n. bits per posting) as a function of the number of clusters

By increasing the radius we observe that the clusters decrease, this is because more documents will be considered similar to each other and will therefore be assigned to already existing clusters without the need to create new ones. Instead, by decreasing the radius we will have a more restricted threshold for belonging to a cluster and therefore new clusters will be created.

Furthermore, we also observe that the number of average bits grow as the number of clusters decreases. This is due to the fact that with fewer clusters the documents are less distributed and therefore we will have clusters with many documents and consequently growing d-gaps.

The execution time increases with a high number of clusters and therefore with a lower radius, this is mainly due, in addition to having a high number of terms, also to the fact that having a more restrictive threshold we are forced to calculate the jaccard distance between all the sets of terms in the forward index and all the medoids, in the hope that the candidate docId can fit into an already existing cluster, and then finally having to create a new cluster, thus adding one more element to compare for the next set of terms.

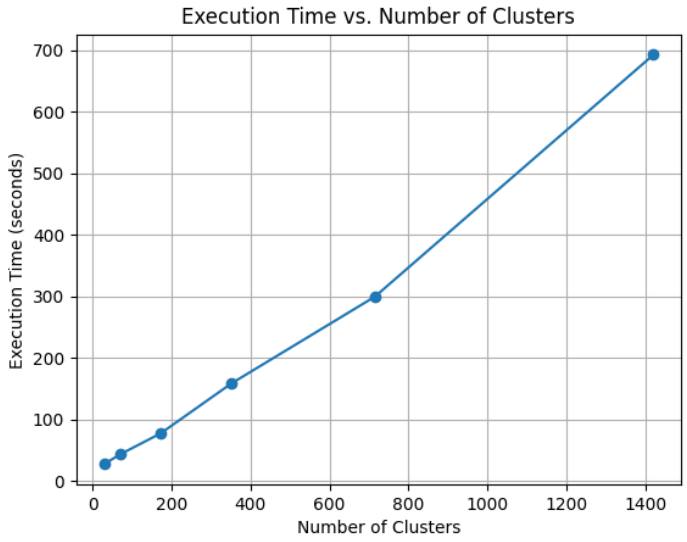

Figure 3: Execution time vs. number of clusters

8 Implementation notes

8.1 Parsing

Collections are contained in the `/collections` folder in a `.dat` file format.

I use two collections:

- lyrl2004_tokens_test_pt0.dat: with 199328 documents
- lyrl2004_tokens_test_pt1.dat: with 199339 documents

for a total of 398667 documents and 4384612 terms.

To process this type of files I followed the directions on the collection publisher page.
Each document in a file is represented in a format used by the SMART text retrieval system. A document has the format:

```
.I <docID>
.W
<textline>+
<blankline>
```

The parser visit each line of the file, if it encounters `.I` it will create a new document, if it encounters `.W` then it will assign the terms in the next lines to the current document until it encounters an `blankline`.
In this way we have a data structure in which each document has its own terms assigned.
Unlike before, where I limited the number of documents to parse so as not to end up with an inverted index that was too large and thus reaching an infeasible computational time for the medoid calculation, now I do not limit the parsing in any way.

8.2 SPIMI-invert

I opted to use a version of the SPIMI algorithm faithful to the theoretical one which constructs the inverted index in blocks which are then merged, using `block_size` as the value indicating the saturation of the main memory.

The algorithm is divided into three functions:

- `spimi_invert(documents, block_size)`: Constructs the inverted index in blocks.
- `write_block_to_disk(index, block_id)`: Writes an index block to disk.
- `merge_blocks(num_blocks)`: Merges all blocks into a single inverted index.

The value of `block_size` depends on various factors, the amount of available memory, the size of the documents, and so on. However, with modern computers often having several gigabytes of RAM, I choose a fairly large `block_size` so as not to create too many blocks.