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

Francesco Vinci 868164

2023/2024
Github repository of the project

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

```
procedure SPIMI-INVERT(token_stream)
    output_file ← NewFile()
    dictionaty ← NewHash()
    while free memory available do
        token ← next(token_stream)
        if term(token) ∉ dictionary then
            postings_list ← AddToDictionary(dictionary, term(token)
        else postings_list ← GetPostingsList(dictionary, term(token)
        end if
        if full(postings_list then
            postings_list ← DoublePostingsList(dictionary, term(token)
        end if
        AddToPostingsList(postings_list, docID(token))
    end while
    sorted_terms ← SortTerms(dictionary)
    WriteBlockToDisk(sorted_terms, dictionary, output_file)
    return output_file
end procedure
```

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.

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.

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.

```

def solve_tsp(medoids, forward_index_set):
    start = medoids[0]
    tour = [start]
    medoids_list = medoids[1:]

    current = start
    while medoids_list:
        next_medoid = min(medoids_list, key=lambda medoid:
        ↪  jaccard_distance(forward_index_set[current], forward_index_set[medoid]))
        tour.append(next_medoid)
        medoids_list.remove(next_medoid)
        current = next_medoid

    return tour

def apply_tsp_to_medoids(forward_index, clusters):
    medoids = [medoid for medoid, _ in clusters]
    forward_index_set = {doc: set(terms) for doc, terms in forward_index.items()}
    tsp_tour = solve_tsp(medoids, forward_index_set)
    return tsp_tour

```

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.

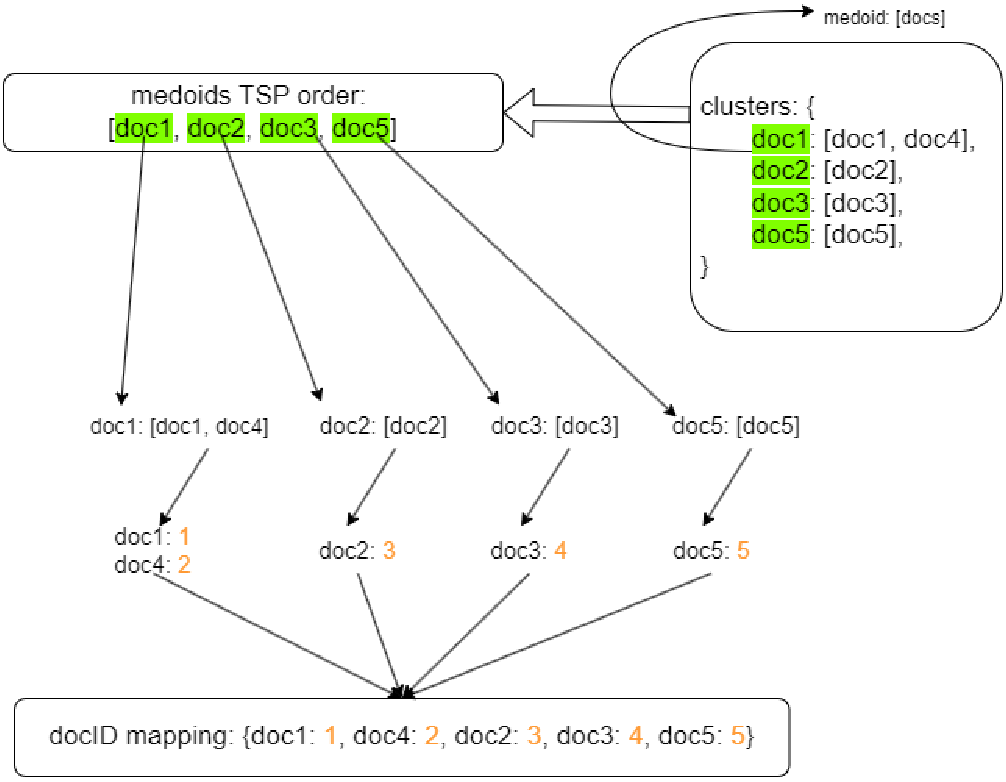

Figure 1: docID reassignment process

```python

def assign_doc_ids(tsp_tour, clusters):
    doc_id = 1
    doc_id_map = {}

    medoid_to_docs = {medoid: docs for medoid, docs in clusters}

    for medoid in tsp_tour:
        docs = medoid_to_docs[medoid]

        for doc in docs:
            doc_id_map[doc] = doc_id
            doc_id += 1

    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.

```python

def reassign_postings(postings, doc_id_map):
    new_postings = {}
    for term_id, docs in postings.items():
        new_postings[term_id] = [doc_id_map[doc] for doc in docs]
    return new_postings
```

7 Evaluation

For the sake of time we use the first 9,766 documents of the RCV1 Dataset with 109,523 unique terms. With a radius of 0.95 we get 178 clusters.

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. bytes without TSP clustering

Encoding	Avg. bytes
VB	30.3696
Elias gamma	28.0396
Elias Delta	26.0911

This will give us an idea of how many bytes 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 bytes to store them.

Table 2: Avg. bytes with TSP clustering

Encoding	Avg. bytes	Bytes saved (%)
VB	28.7706	5.26
Elias gamma	25.122	10.40
Elias Delta	23.973	8.12

8 Implementation notes

8.1 Parsing

The collection is contained in the `/collection` folder in a `.dat` file format.

To process this type of file 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.

As mentioned previously, for a matter of time we cannot parse all the documents in the collection (around 200,000 docs with more than 1 million terms), since subsequently, in the calculation of the clusters and TSP the computation is not feasible in terms of time.

For this reason I force parsing to stop once the docID 36000 is reached. Since the documents in the collection start with a docID equal to 26151, we will have around 9700 documents, as there are some jumps in the numbering.

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, we can choose a fairly large `block_size` so as not to create too many blocks.