Search Engine Implementation

CiTIUS Centro Singular de Investigación en Tecnoloxías Intelixentes

USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

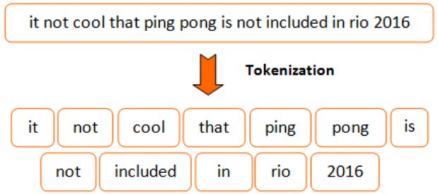
CAMPUS VIDA

Tecnologías de Gestión de Información No Estructurada

Prof. Dr. David E. Losada

Máster Interuniversitario en Tecnologías de Análisis de Datos Masivos: Big Data

SE components



Tokenizer. raw strings => tokens (or features)

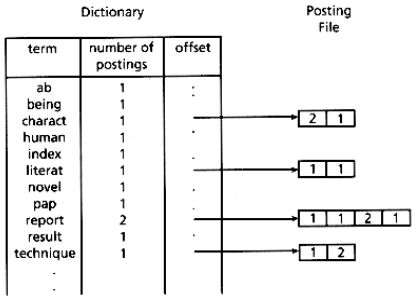
a poor tokenization method will affect all other parts of the system

Indexer. indexes docs with appropriate data structures.

can be run offline

challenges: index large amounts of documents quickly with a limited amount of memory

must support addition and deletion of documents



SE components

Scorer/Ranker. takes a query and returns a ranked list of documents

the challenge is to implement a retrieval model efficiently

Feedback/Learner. This is the module that is responsible for relevance feedback or pseudo feedback

SE: two additional optimizations

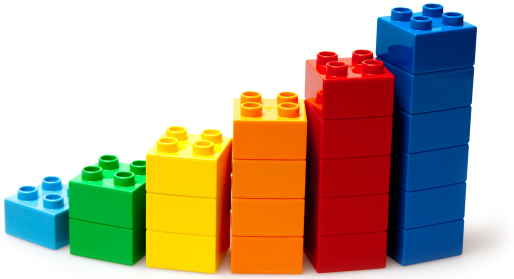
to be more efficient in both speed and disk usage…

Compression. The documents could consume hundreds of Tbs. We can simultaneously save disk space and increase disk read efficiency by losslessly compressing the data (usually integers) in our index

Caching. a cache between the frontfacing API and the document index on disk.

saves frequently-accessed term information (so the number of slow disk seeks during query-time is reduced)

tokenization

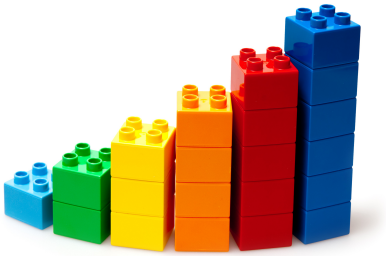


segments the document into countable features or tokens (typically words)

the **raw count** can be used by the scorer to calculate some weighted term representation

no TF-IDF-like calculations (requires corpus stats and, furthermore, we'd like our scorer to be able to use different scoring functions as necessary)

EJEMPLOS



whitespace_tokenizer(Mr. Quill's book is very very long.)

⬇

{Mr.: 1, Quills: 1, book: 1, is: 1, very: 2, long.: 1}

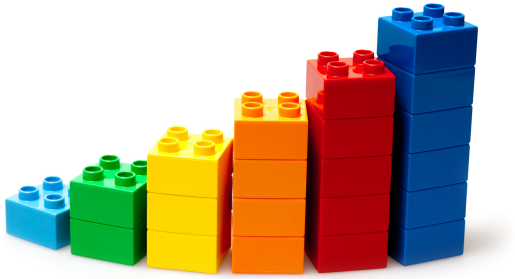
handling **uppercase** & split words based on **punctuation**

⬇

{mr.: 1, quill: 1, 's: 1, book: 1, is: 1, very: 2, long: 1, .: 1}.

we could use **bigram words**, **POS**-tags, grammatical parse tree features, or any combination

doc & term IDs



another common task of the tokenizer is to assign **document IDs**

much more efficient to refer to docs as unique numbers as opposed to strings (e.g. paths or urls)

term IDs. We could set mr.→term id 0, quill→term id 1 and so on

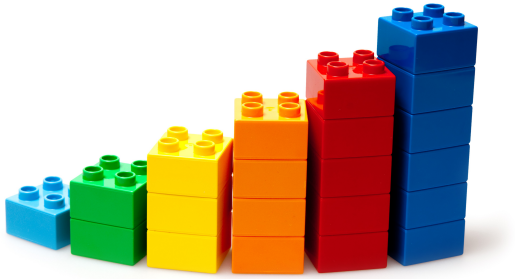
{mr.: 1, quill: 1, 's: 1, book: 1, is: 1, very: 2, long: 1, .: 1}

⬇

{1, 1, 1, 1, 1, 2, 1, 1}

a real doc vector would be much **larger** and much **sparser** (most of the dimensions will have a count of zero)

“**feature generation**” (or “**text vectorization**”)



defines the **building blocks** of our document objects

once we define how to vectorize documents, we can index them, cluster them, and classify them, among many other **text mining** tasks

indexer

designed to be able to index data that is much larger than the amount of system memory

only load portions of the raw corpus in memory at one time

when running queries on our indexed files, we want to ensure that we can return the necessary term statistics fast enough

scanning over every document in the corpus to match terms in the query will not be sufficient

inverted index

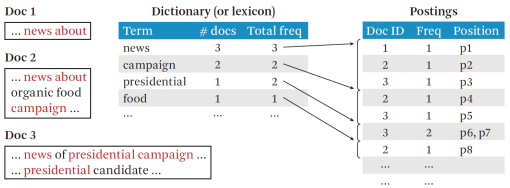


Figure 8.1 Inverted index postings and lexicon files.

the main **data structure** used in a search engine

allows for **quick lookup** of documents that contain any given term

- the **lexicon** (a lookup table of term-specific information, such as doc frequency and where in the postings file to access the per-document term counts)

- the **postings** file (mapping from any term integer ID to a list of doc IDs and frequency information of the term in those documents)

jeemplo

Doc 1

... news about

Doc 2

... news about
organic food
campaign ...

Doc 3

... news of presidential campaign ...
... presidential candidate ...

Dictionary (or lexicon)

Term	# docs	Total freq
news	3	3
campaign	2	2
presidential	1	2
food	1	1
...	...	...

Postings

Doc ID	Freq	Position
1	1	p1
2	1	p2
3	1	p3
2	1	p4
3	1	p5
3	2	p6, p7
2	1	p8
...	...	
...	...	

posicion de esa palabra

Figure 8.1 Inverted index postings and lexicon files.

The **position information** can be used to check whether all the query terms are matched within a certain <u>window of text</u> (e.g., phrase matching)

The arrows in the image are actually integer **offsets** that represent bit or byte indices into the **postings file**

doc IDs (and positions) stored in **increasing order** (to facilitate **compression**)

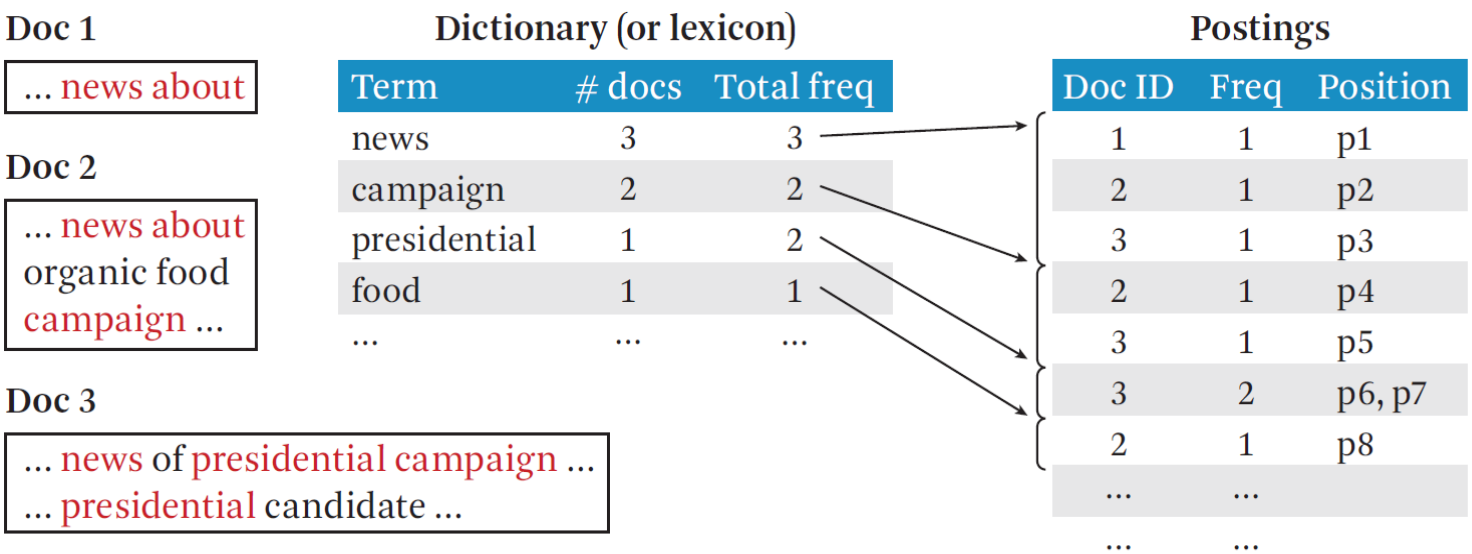


Figure 8.1 Inverted index postings and lexicon files.

large difference in **size** of the lexicon and postings file

we often assume that the lexicon can fit into main memory and the postings file resides on disk, and is seeked into based on pointers from the lexicon

Indexing: process of creating these data structures based on a set of tokenized documents

sorting-based indexing

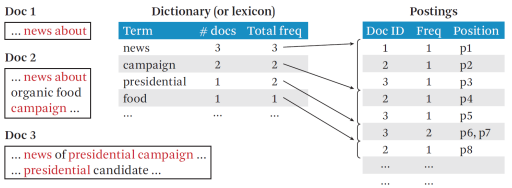


Figure 8.1 Inverted index postings and lexicon files.

- **Scan the raw document stream sequentially**. In tokenization, assign each document an ID. Tokenize each document to obtain term IDs, creating new term IDs as needed.

- While scanning documents, collect term counts for each term-document pair and **build an inverted index for a subset of documents in memory**. When we reach the limit of memory, write the incomplete inverted index into the disk.

- Continue this process to generate **many incomplete inverted indices**

- **Merge** all these indices in a pair-wise manner to produce a single sorted (by term ID) postings file.

- Once the postings file is created, **create the lexicon** by scanning through the postings file and assigning the offset values for each term ID.

sorting-based indexing

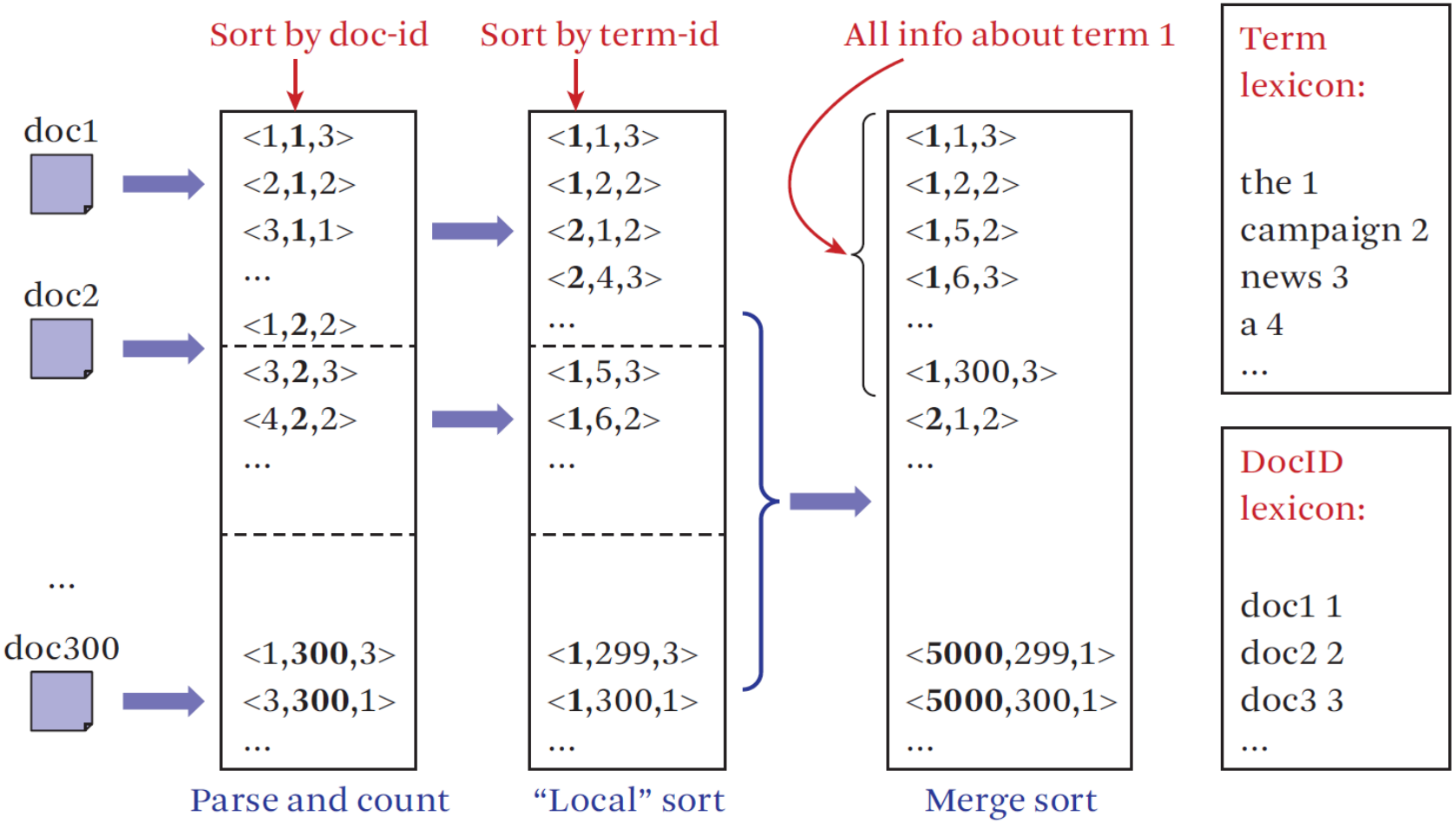


Figure 8.2 Sort-based inversion of inverted index chunks.

forward index

Forward Index

Document	Words
Document 1	the,cow,says,moo
Document 2	the,cat,and,the,hat
Document 3	the,dish,ran,away,with,the,spoon

maps docs to terms

(records a term vector for each doc ID)

useful when doing <u>other operations aside from search</u> (e.g., clustering or classification). Using an inverted index to do this is not efficient at all

may be created in a similar way to the inverted index

scorer

many documents do not contain any of the query terms, their ranking score will be zero

with the inverted index, we can only score documents that match at least one query term (i.e., ignore docs with zero score)

Term-at-a-time Ranking

```
scores = {}        // score accumulator maps doc IDs to scores
for w ∈ q do
    for d, count ∈ Idx.fetch_docs(w) do
        scores[d] = scores[d] + score_term(count)
    end for
end for
return top k documents from scores
```

but the **size of the score accumulators** scores will be the size of the number of docs matching at least one term

while this is a huge improvement over all documents in the index, we can still make this data structure smaller…

Doc-at-a-time

instead of iterating through each document multiple times for each matched query term occurrence, **scores an entire document at once**

1) using the inverted index, initially gets a list of document IDs and postings data that need to be scored

2) as we score a complete document, it is added on **a priority queue**

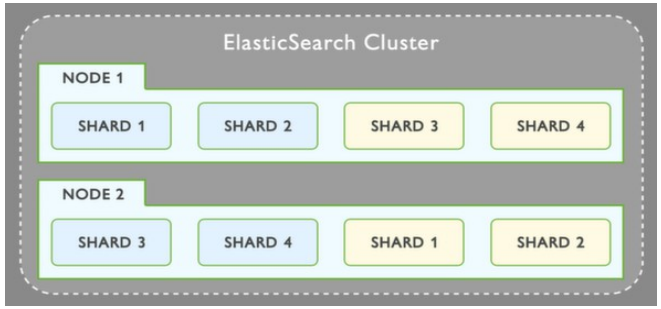
most searches are top-k searches, **only keeps the top k documents** at any one time

filtering documents

only returning documents that **meet a certain criteria.** For example, suppose we want to only return documents that were written within the past year.

filters can be as complex as desired (a Boolean function that takes a document and returns whether or not it should be returned)

index sharding



more than one inverted index for a particular search engine

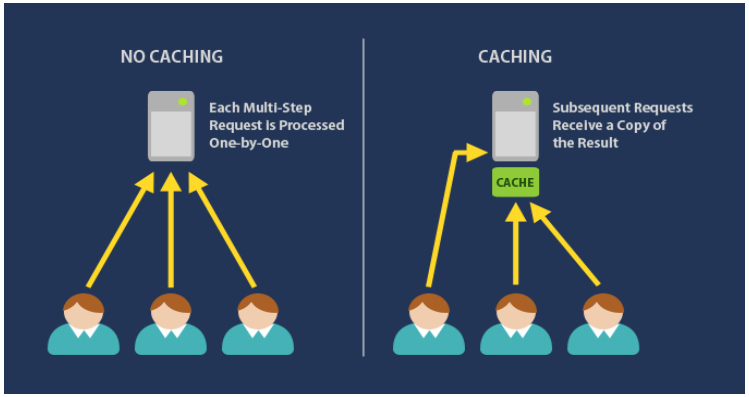
achieved by stopping the postings chunk merging process

e.g. have the number of shards equal to the number of threads (or **cluster nodes**) in our search system.

searches for matching terms in its respective shard, and the final search results are then **merged** together

map reduce paradigm: distributing the work and merging results

cache



the basic idea of a cache is to make **frequently accessed objects fast to acquire**

In SE, **the objects we access are postings lists**

Due to Zipf's law, we expect that <u>a relatively small number of postings lists will be accessed the majority of the time</u>

cache designs try that (1) the most-frequently used items are <u>fast to access</u>, and (2) the cache <u>doesn't get too large</u>

e.g. LRU