Low-Level Details

Project for the *Information Retrieval* course

Gaia Saveri

18th December 2020

DSSC - UniTS

Outline

Introduction

The goals of this project are the following:

1. Implement a standard Boolean Information Retrieval System;
   - → **B+Tree** used as data structure for the dictionary;
   - → **Singly Linked List** used as data structure for the posting lists;

Introduction

The goals of this project are the following:

1. Implement a standard Boolean Information Retrieval System;
2. Implement a compressed representation for the dictionary;
   - → **Blocked Storage**: the set of consecutive terms is divided into blocks of size k;
   - → **Front coding** of each block: the fact that consecutive entries in an alphabetically sorted list share common prefixes is exploited;

Introduction

The goals of this project are the following:

1. Implement a standard Boolean Information Retrieval System;
2. Implement a compressed representation for the dictionary;
3. Compress the Posting file;
   - → **Gaps** between postings are stored, instead of entire DocID;
   - → Gaps are represented using **Variable Byte Coding**;

Introduction

The goals of this project are the following:

1. Implement a standard Boolean Information Retrieval System;
2. Implement a compressed representation for the dictionary;
3. Compress the Posting file;
4. Compare size and speed of different representations;

Introduction

The goals of this project are the following:

1. Implement a standard Boolean Information Retrieval System;
2. Implement a compressed representation for the dictionary;
3. Compress the Posting file;
4. Compare size and speed of different representations;

Disclaimer

The index is assumed to be **static**, i.e. if the collection changes, it has to be rebuilt from scratch.

Reuters-21578 Dataset

The corpus **Reuters-21578** is a set of financial dispatches emitted during the year 1987 by the Reuters agency.

- All documents are in English;
- The texts of this corpus have a journalistic style.

```
JANUARY HOUSING SALES DROP, REALTY GROUP SAYS
  Sales of previously owned homes
  dropped 14.5 pct in January to a seasonally adjusted annual
  rate of 3.47 mln units, the National Association of Realtors
  (NAR) said.
      But the December rate of 4.06 mln units had been the
  highest since the record 4.15 mln unit sales rate set in
  November 1978, the group said.
      "The drop in January is not surprising considering that a
  significant portion of December's near-record pace was made up
  of sellers seeking to get favorable capital gains treatment
  under the old tax laws," said the NAR's John Tuccillo.
```

Extract of a document in the corpus

Reuters-21578 ApteMod Collection

The **Reuters ApteMod** collection is a subset of the Reuters-21578. This is the corpus used for this project.

Dataset info:

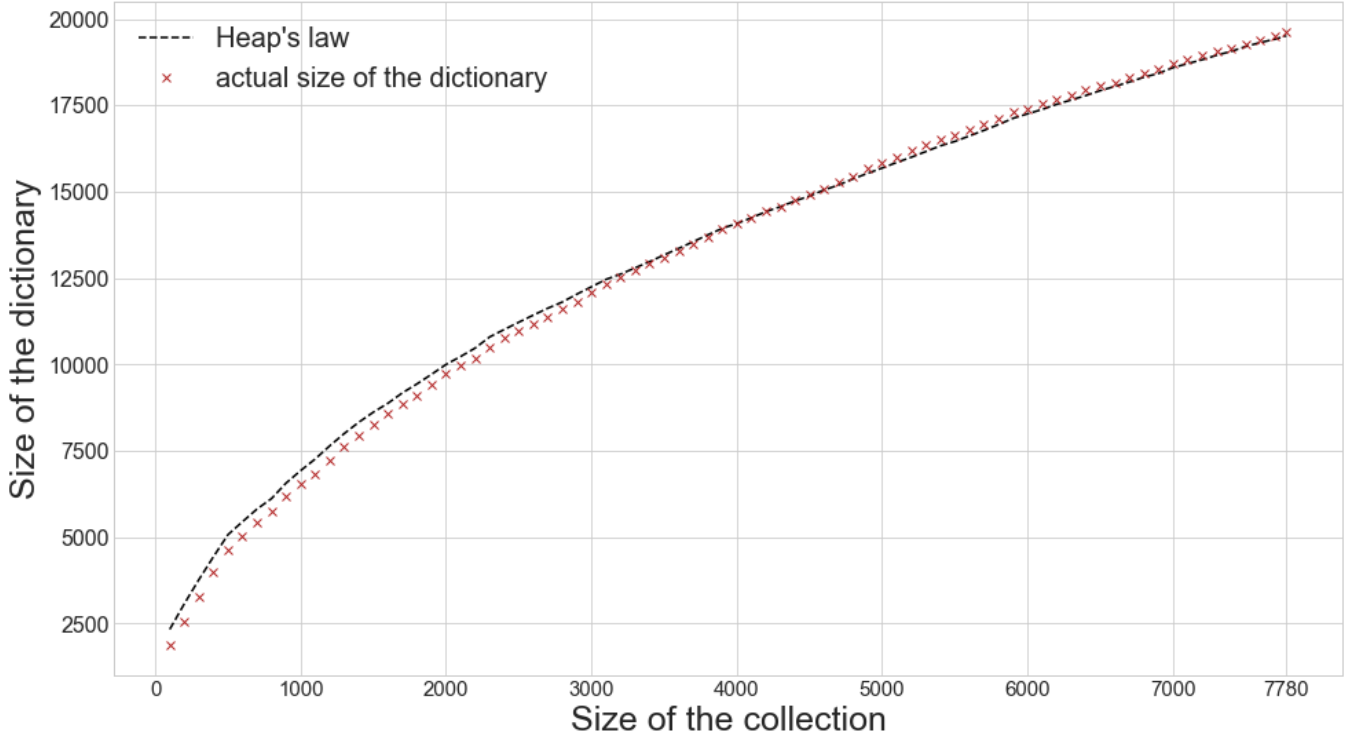
- 7780 documents from the Reuters financial newswire service;
- Each document is stored in a different file;
- A set of stop-words is provided;
- Term vocabulary of approximately 20000 words (after pre-processing).

Exploring the Dataset: Heap's Law

Heap's Law

It estimates vocabulary size M as a function of the collection size T (number of tokens) as:

$$M = k \cdot T^b$$

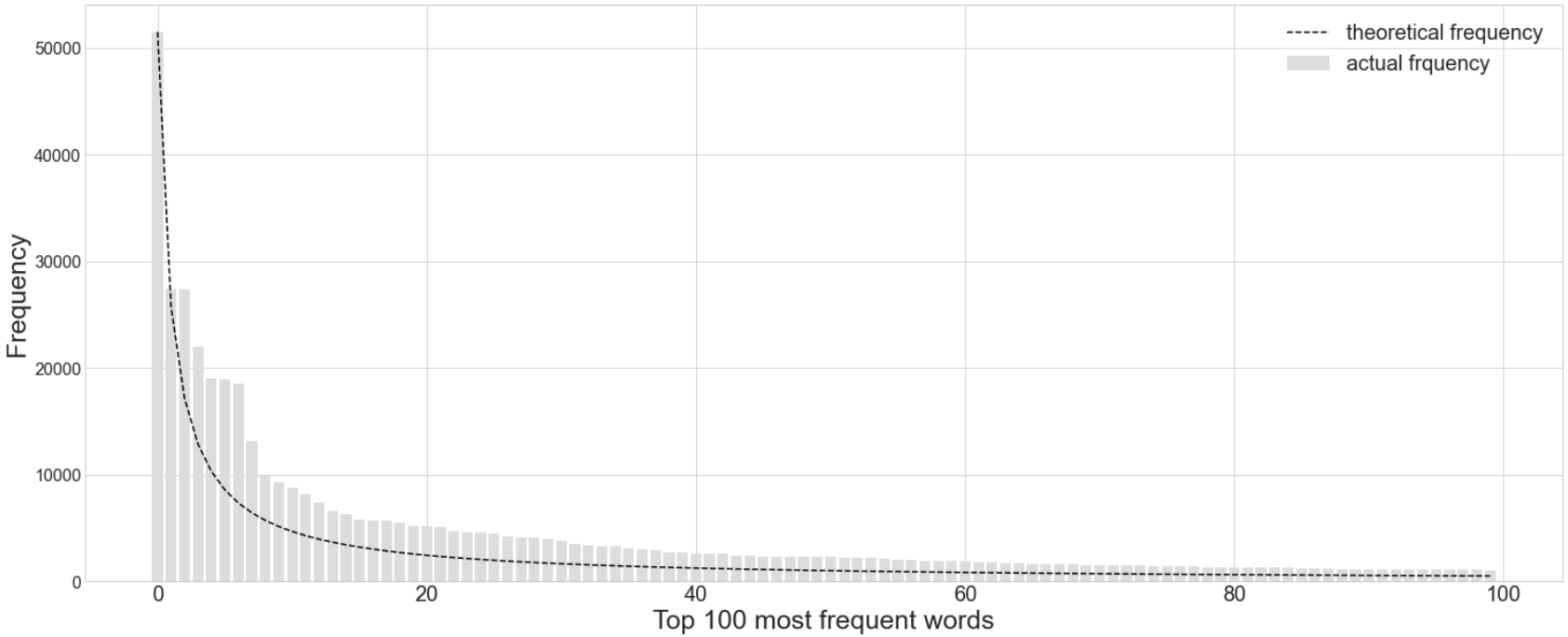


Heap's Law for used dataset, $b = 0.5$, $k = 21$

Exploring the Dataset: Zipf's Law

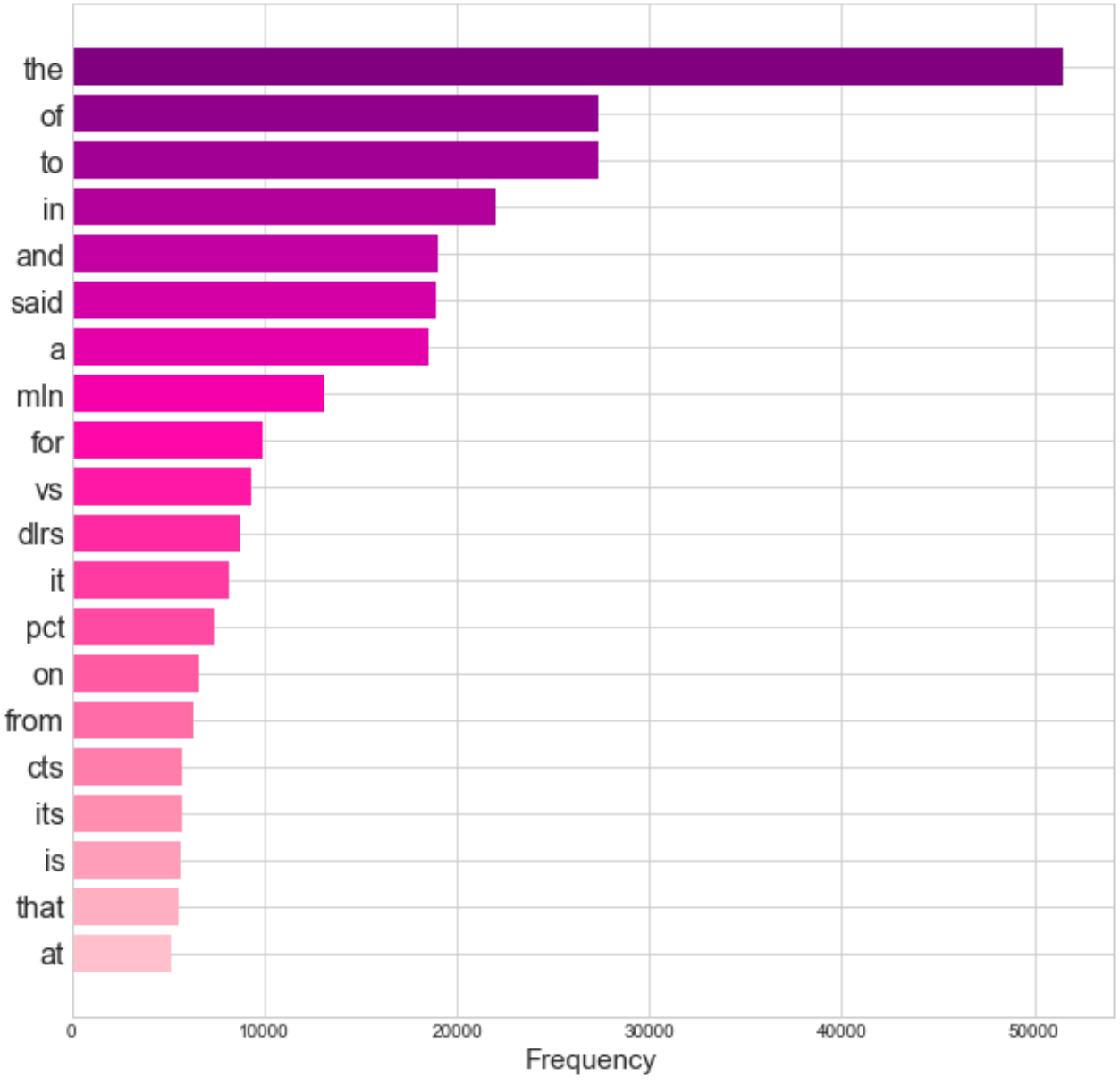
Zipf's Law

It models the distribution of terms in a collection as a power law: $cf_i \propto \frac{1}{i}$ being cf_i the collection frequency of the i-th most common term.

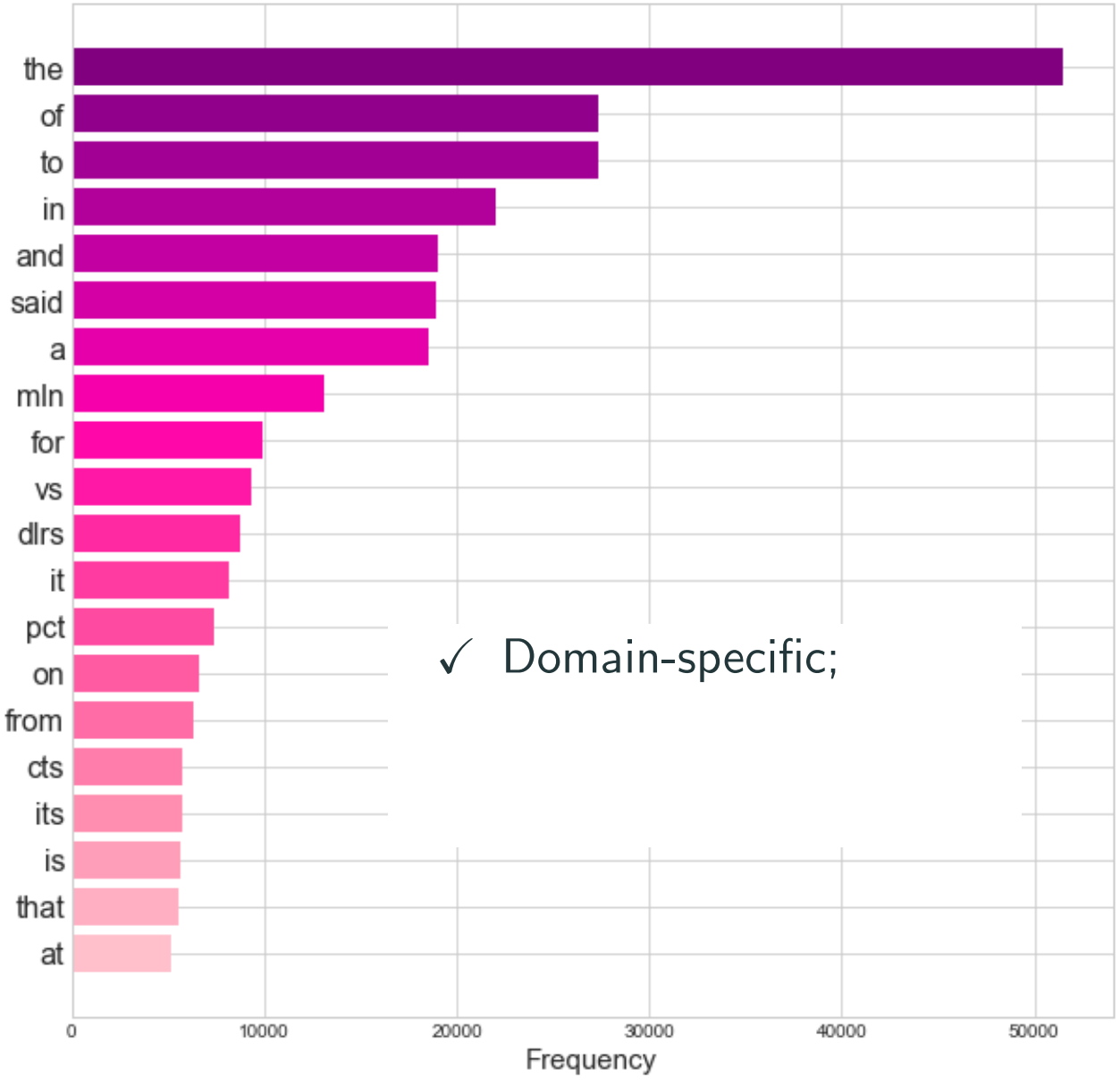


Zipf's Law for used dataset,
dashed line represents the function $cf_i = cf_1 \cdot \frac{1}{i}$

Exploring the Dataset: top 20 most frequent words

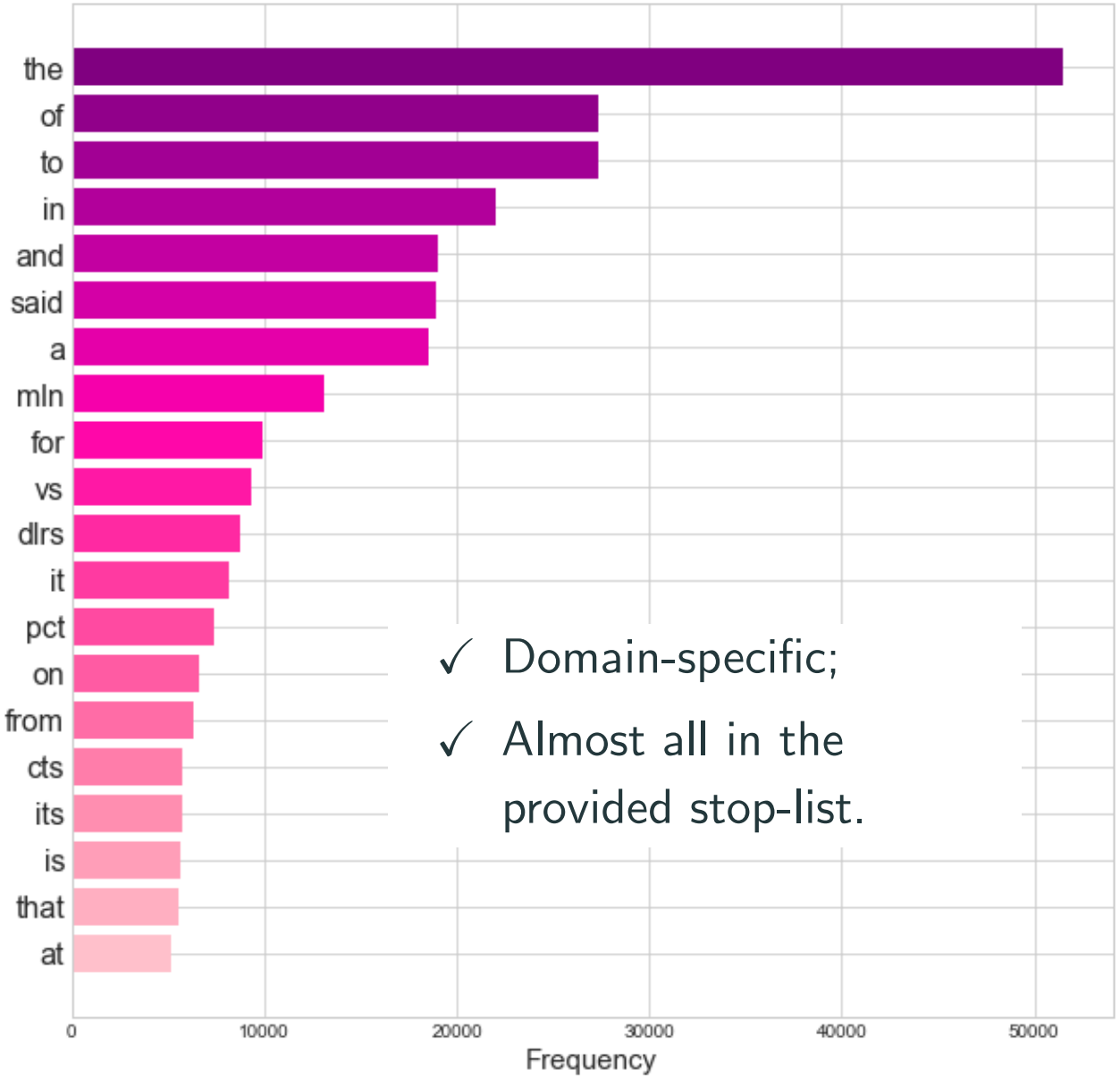


Exploring the Dataset: top 20 most frequent words



✓ Domain-specific;

Exploring the Dataset: top 20 most frequent words



- ✓ Domain-specific;
- ✓ Almost all in the provided stop-list.

Outline

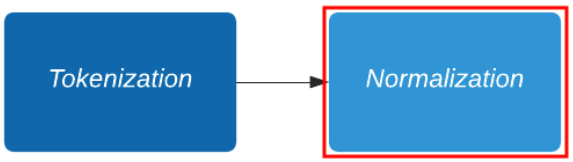
Preprocessing pipeline

The following steps are applied to each document:

- Each document is read in order to retrieve a list of words (tokens);

Preprocessing pipeline

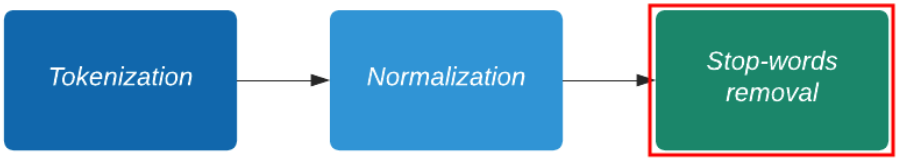
The following steps are applied to each document:



- The following are performed to the previously obtained list of words:
  1. Case-folding: all words to lower-case;
  2. Non-textual characters are removed;

Preprocessing pipeline

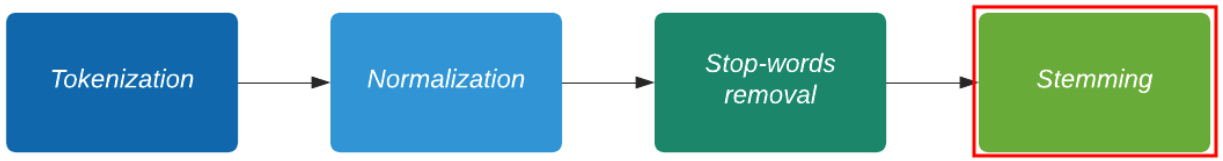
The following steps are applied to each document:



- Stop-words are detached using the stop-list provided with the dataset;

Preprocessing pipeline

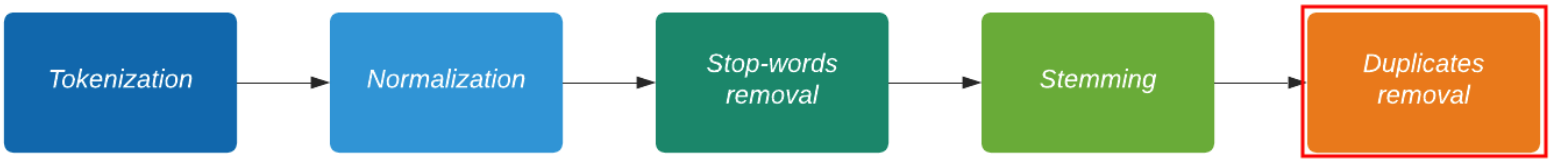
The following steps are applied to each document:



- This step aims at transforming derivationally related forms of a word to a common base form: the *Porter Stemmer* is used, which is a process for removing the commoner morphological and inflexional endings from words in English;

Preprocessing pipeline

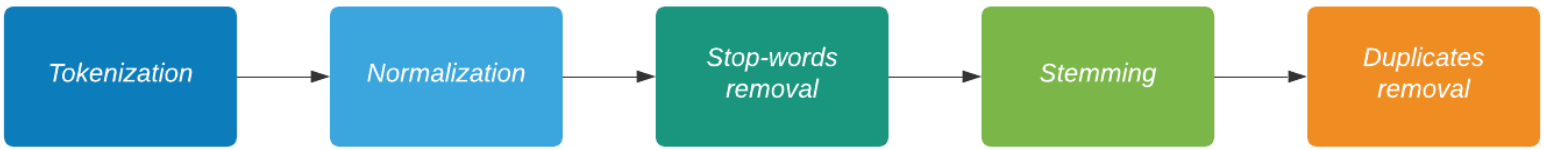
The following steps are applied to each document:



- Only one occurrence for each token is retained;

Preprocessing pipeline

The following steps are applied to each document:



→ Now we are left with normalized types that will be inserted into the dictionary.

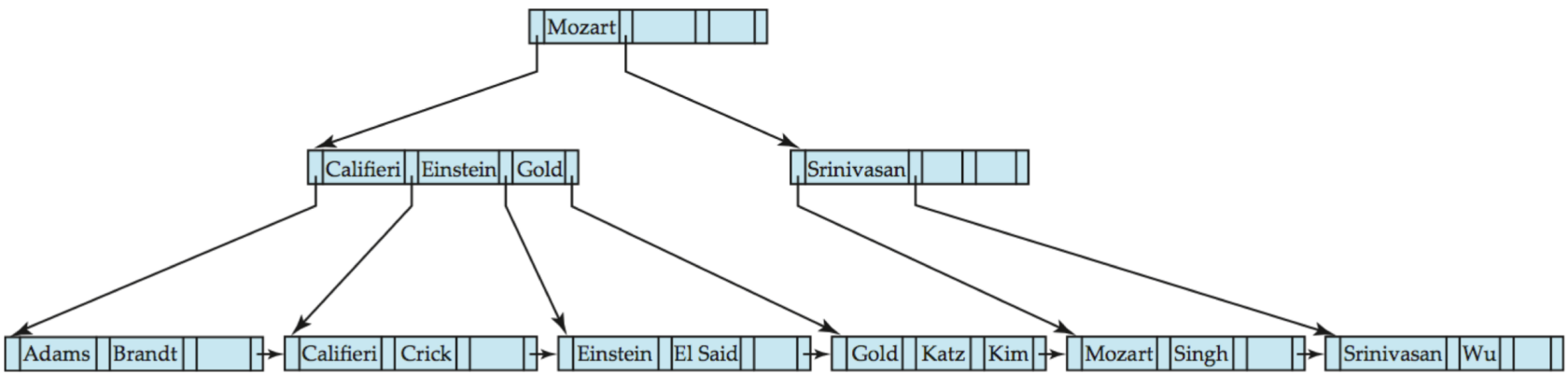
Outline

Search Structure for the Dictionary

B+Tree

A *B+Tree* of order k is a self-balancing tree in which:

- each *internal node* contains up to k children nodes, thus stores up to $k-1$ keys (used as placeholders to guide the search);
- each *leaf node* contains up to k keys, and one (pointer to) value for each key. Leaves are linked together;
- the *root node* is either a leaf or it has at least two children.



Example of B+Tree [1]

[1] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts - Seventh Edition, McGraw-Hill, 2019.

B+Tree basic operations: search

Searching for a key in the B+Tree always starts at the root node and moves downwards until it reaches a leaf node.

The algorithm proceeds as follows:

1. Start from the root;
2. At each internal node, binary search the keys to figure out which pointer to follow;
3. At leaf node, binary search the keys to check for the presence of the searched key.

$\rightarrow$ Complexity is $O(log_{\lceil \frac{k}{2} \rceil}(N))$, being $N = \#$ of keys in the tree.

B+Tree basic operations: insert

Inserting a key in the B+Tree always starts at the leaves level. Eventually the tree will grow at the root.

The algorithm proceeds as follows:

1. Find the leaf node in which the key would appear;
2. Insert the key in the correct leaf:
   - If there is enough space, done;
   - Otherwise the leaf has to be split: redistribute keys evenly and copy up the middle key (fix also pointers between parents and children and between sibling leaves);
3. This can propagate recursively to other nodes:
   - When splitting non-leaf nodes, entries are redistributed evenly, the middle key is pushed up.

$\rightarrow$ Complexity is $O(log_{\lceil \frac{k}{2} \rceil}(N))$, being $N = \#$ of keys in the tree.

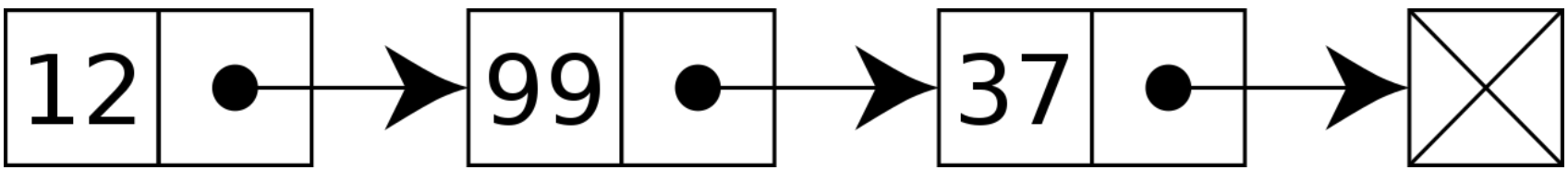
Data Structure for the Posting Lists

Singly Linked List

A *Singly Linked List* is a collection of nodes which together represent a sequence. Each node contains:

- a value;
- a pointer to the next node in the sequence.

Unlike conventional arrays, the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because the data items need not to be stored contiguously in memory.



Example of Linked List.[2]

[2] https://en.wikipedia.org/wiki/Linked_list

Building the Inverted Index

The inverted index is composed by a B+Tree containing all the terms in the vocabulary, having in the leaves all the words and the pointers to the linked lists where posting lists are stored.

In order to build it, we proceed as:

1. Read one document from the collection and pre-process it;

Building the Inverted Index

The inverted index is composed by a B+Tree containing all the terms in the vocabulary, having in the leaves all the words and the pointers to the linked lists where posting lists are stored.

In order to build it, we proceed as:

1. Read one document from the collection and pre-process it;
2. Assign to the document a unique (serial) DocID;

Building the Inverted Index

The inverted index is composed by a B+Tree containing all the terms in the vocabulary, having in the leaves all the words and the pointers to the linked lists where posting lists are stored.

In order to build it, we proceed as:

1. Read one document from the collection and pre-process it;
2. Assign to the document a unique (serial) DocID;
3. For each term in the document, insert in the index the key-value pair $(term, docID)$:
   - if the term is not present, insert both the term and the posting;
   - if the term is already present, find the term and add to its posting list the current DocID;

Building the Inverted Index

The inverted index is composed by a B+Tree containing all the terms in the vocabulary, having in the leaves all the words and the pointers to the linked lists where posting lists are stored.

In order to build it, we proceed as:

1. Read one document from the collection and pre-process it;
2. Assign to the document a unique (serial) DocID;
3. For each term in the document, insert in the index the key-value pair $(term, docID)$:
   - if the term is not present, insert both the term and the posting;
   - if the term is already present, find the term and add to its posting list the current DocID;
4. At the end of the index construction, the leaves of the tree will contain all the terms, sorted alphabetically, and pointers to sorted posting lists.

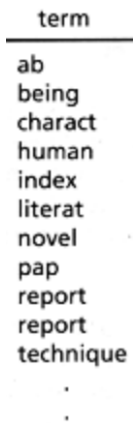
Outline

Model Baseline

So far all computations were done in memory:

→ one of the reasons behind compressing the dictionary is to have faster transfer of data from disk to memory;

→ we need a new model baseline to compare the speed of retrieving terms from disk!

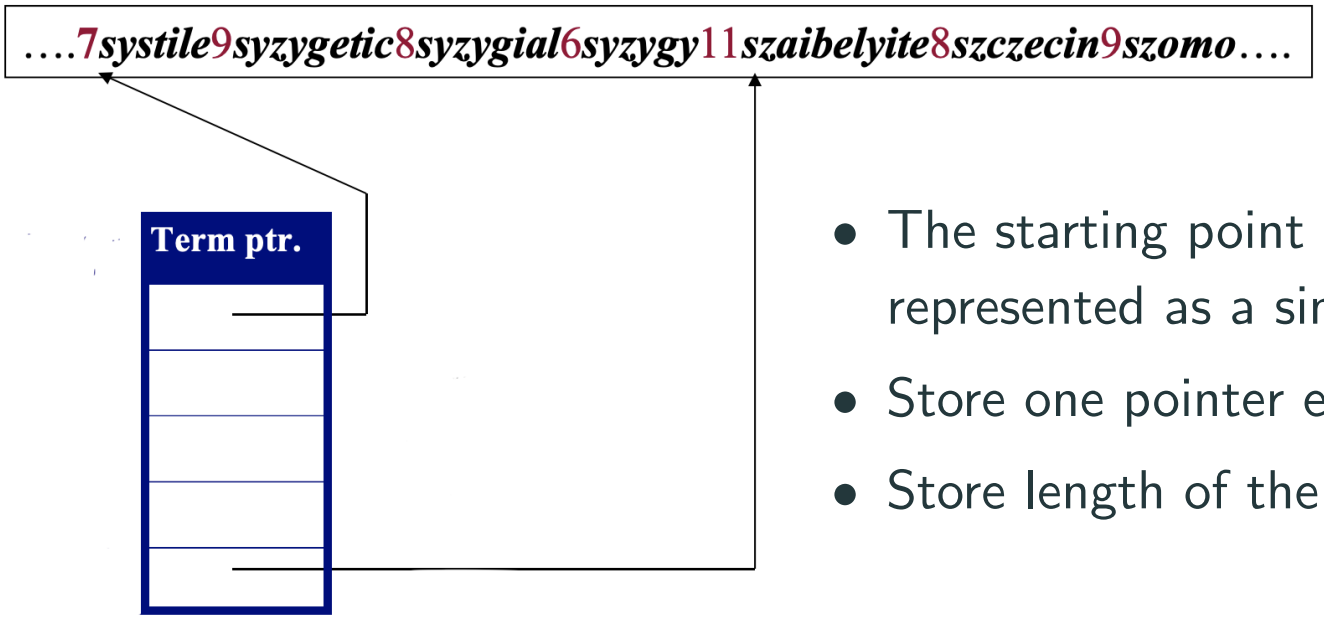


File containing the dictionary[3].

- start from the dictionary as produced by the B+tree;
- mmap the file so that access is as simple as dereferencing a pointer;
- keep offsets to the beginning of each term;
- terms can be retrieved in logarithmic time by binary searching the file;

[3] http://orion.lcg.ufrj.br/Dr.Dobbs/books/book5/chap03.htm

Compression: Blocked Storage

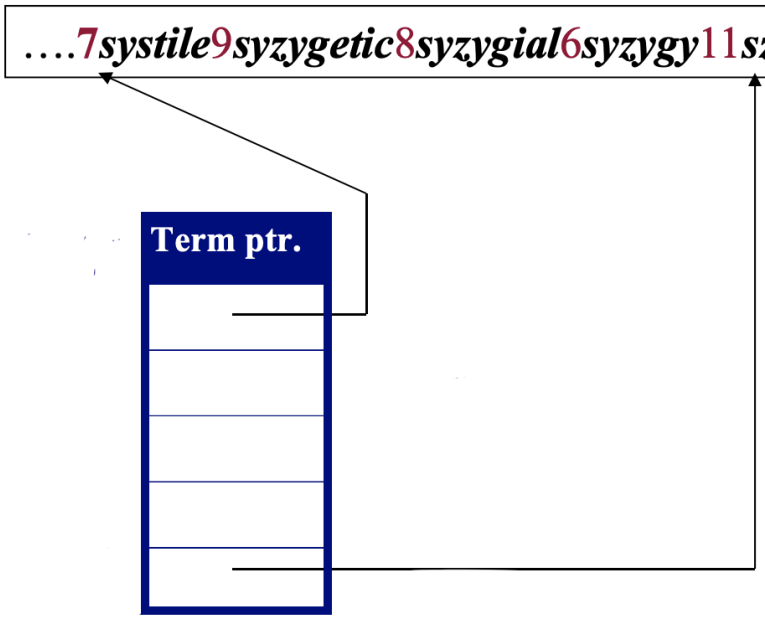


- The starting point is the dictionary represented as a single string;
- Store one pointer every k terms;
- Store length of the words (1 extra byte);

Blocked storage, 4 terms per block[4].

[4] https://www.cs.ucy.ac.cy/courses/EPL660/lectures/lecture5-compression.pdf

Compression: Blocked Storage



- The starting point is the dictionary represented as a single string;
- Store one pointer every k terms;
- Store length of the words (1 extra byte);

Blocked storage, 4 terms per block[4].

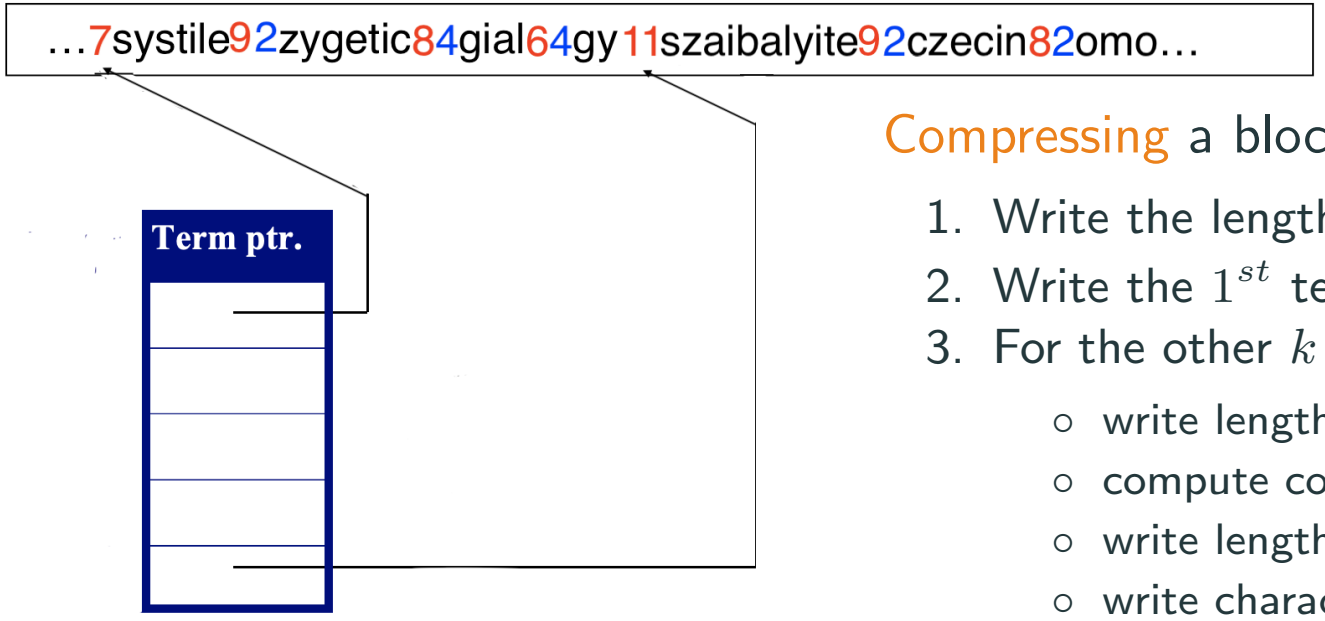
$\rightarrow$ Only 1 pointer every k terms is inserted in the inverted index.

[4] https://www.cs.ucy.ac.cy/courses/EPL660/lectures/lecture5-compression.pdf

Compression: Blocked Storage + Front Coding

Sorted words commonly have a common prefix.

The idea behind *front coding* is to exploit the common prefix and store only the differences.

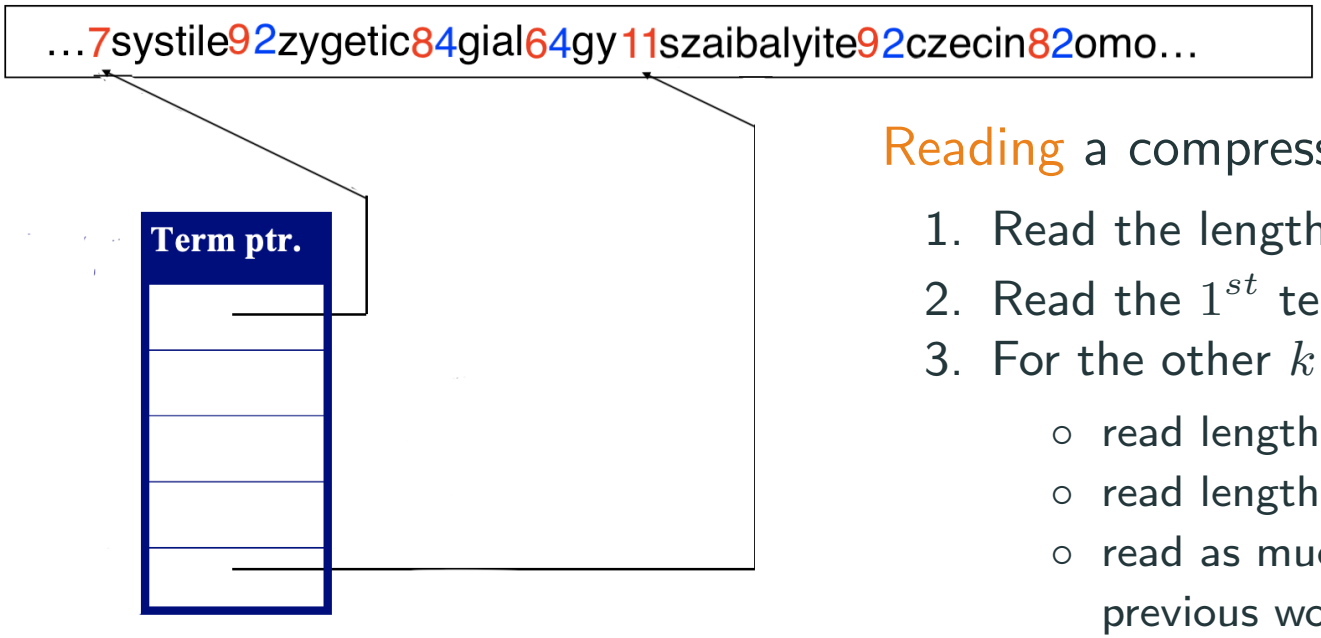


Compressing a block of k terms:

1. Write the length of 1^{st} the term (1 byte);
2. Write the 1^{st} term of the block;
3. For the other $k-1$ terms:
   - write length of the term (1 byte)
   - compute common prefix;
   - write length of the prefix (1 byte);
   - write characteristic suffix.

Blocked storage + front coding, 4 terms per block.

Blocked Storage + Front Coding Cont'd



Reading a compressed block of k terms:

1. Read the length of 1^{st} the term (1 byte);
2. Read the 1^{st} term of the block;
3. For the other $k-1$ terms:
   - read length of the term (1 byte)
   - read length of the shared prefix;
   - read as much as necessary from the previous word;
   - read characteristic suffix.

Blocked storage + front coding, 4 terms per block.

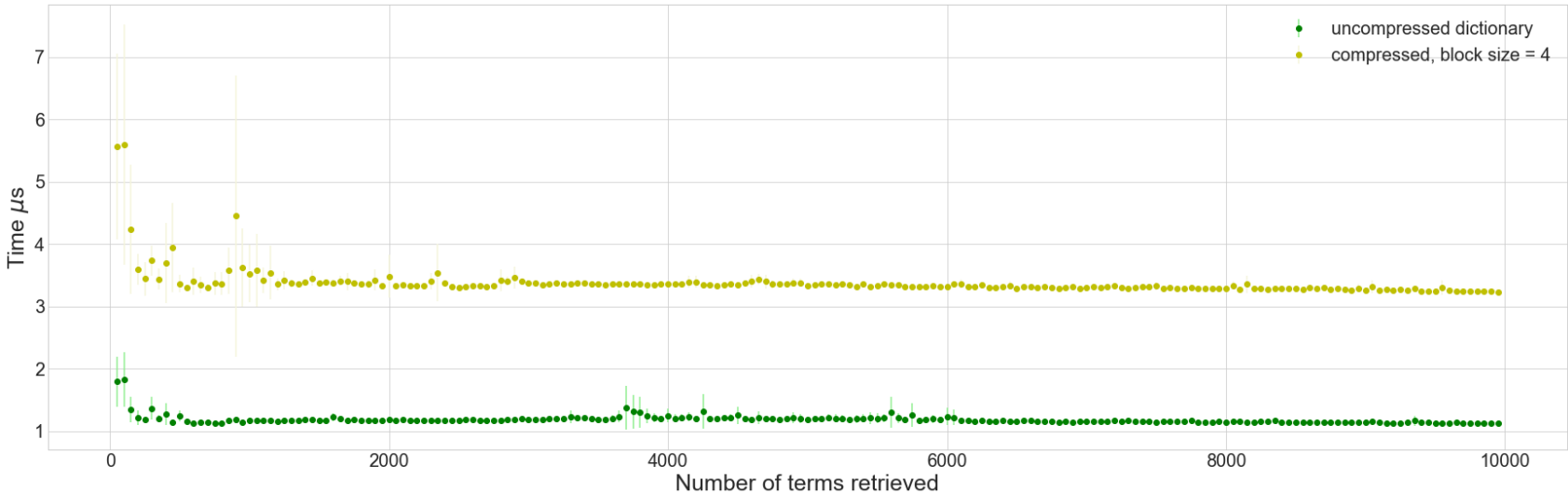
Compressed Dictionary: searching

In order to lookup for a term in the compressed dictionary, proceed as follows:

1. Locate the block in which the term should be contained.
   This is achieved by binary searching the blocks:
   - $\rightarrow$ read the first term of the current block and the first term of the next block.
2. Linear scan the selected block.

Dictionary: speed comparison

Retrieving a term: compressed vs uncompressed dictionary

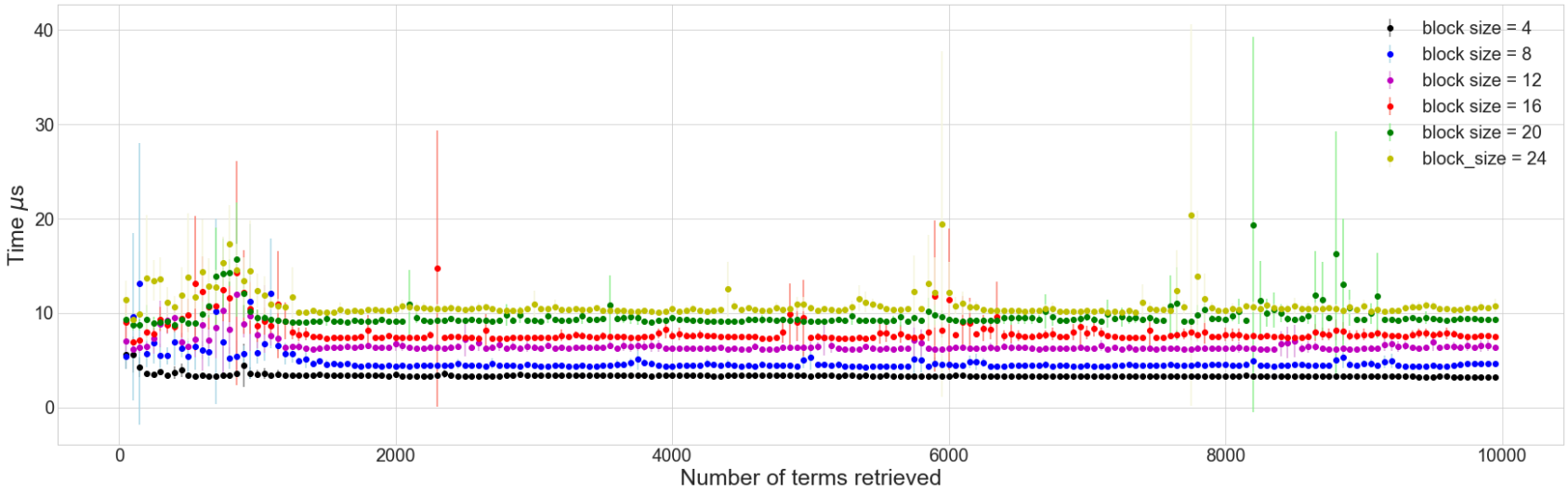


$\rightarrow$ Searching in the compressed dictionary is slower;

$\rightarrow$ Term retrieval performance is still at microsecond level.

Dictionary: speed comparison cont'd

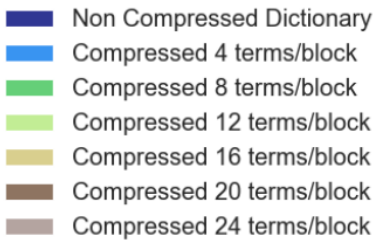
Retrieving a term: different block-sizes

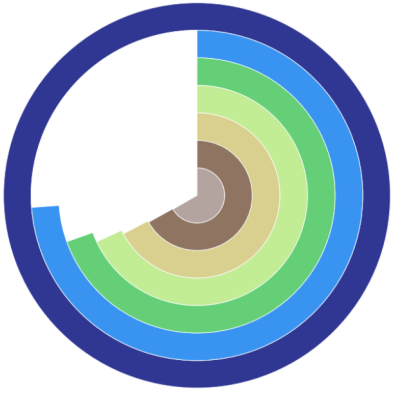


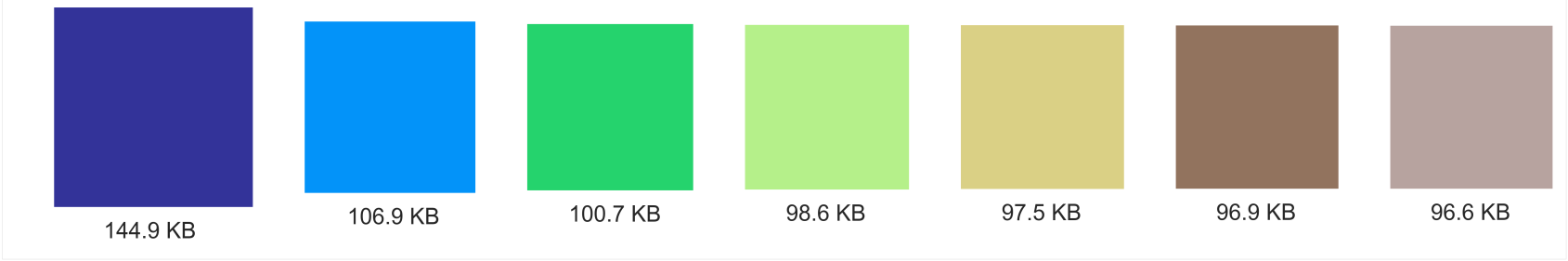
$\rightarrow$ The bigger the block-size, the slower the retrieval.

Dictionary: size comparison

→ The bigger the block-size, the smaller the dictionary size.



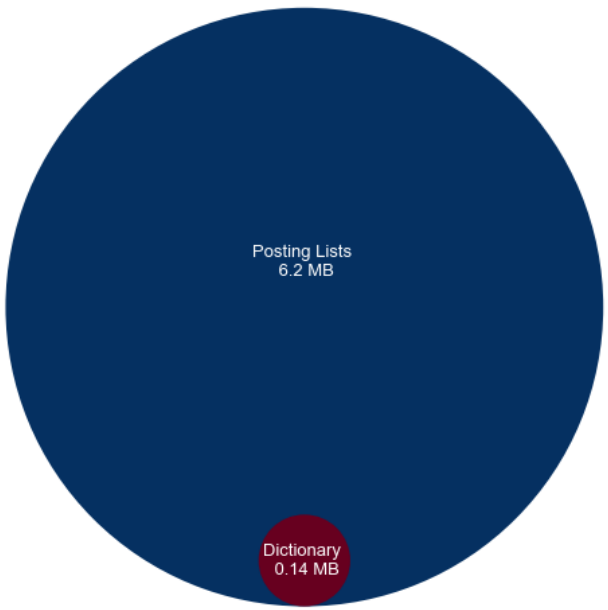




Outline

Model baseline

Consider the postings file, as produced by the B+tree.
In order to retrieve the posting list associated to a term:



1. mmap the file and keep offsets to the beginning of each posting list;
2. by construction, the posting list associated to the i-th term of the dictionary will start at the i-th line of the postings file;
3. take as input the line-index of the term (the posting list is always retrieved after locating the term);
4. read the posting list into a linked list.

Compression: Variable Byte Codes

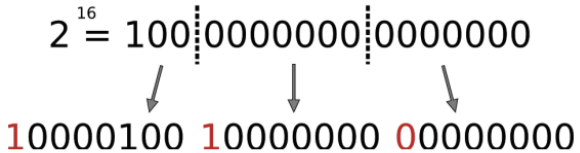
Instead of considering postings, consider gaps between postings.

Postings for frequent terms are close together:

$\rightarrow$ gaps for the most frequent terms are mostly equal to 1;

$\rightarrow$ gaps for rare terms have the same order of magnitude of the DocID;

For an efficient representation of the posting file, we need a *variable byte coding* method that uses fewer bits for short gaps.



The binary representation of the gaps is partitioned into groups of 7-bits.
A flag bit (*continuation bit*) is appended to each group to indicate whether the group is the last (0) or not (1) of the representation.

Variable Byte Code: implementation

Encoding: use an integral number of bytes to encode a gap.

- ✓ An integer is encoded into a vector of unsigned char (1 byte each);
- ✓ Bit-wise operations are used to set the continuation bit.

```
void VBencoder(std::vector<unsigned char>& coded,
               int number){
  while(number > 0x7f){
    auto x = (number & 0x7f);
    x |= 0x80;
    coded.push_back(x);
    number >>= 7;
  }
  coded.push_back(number & 0x7f);
}
```

Reading compressed postings

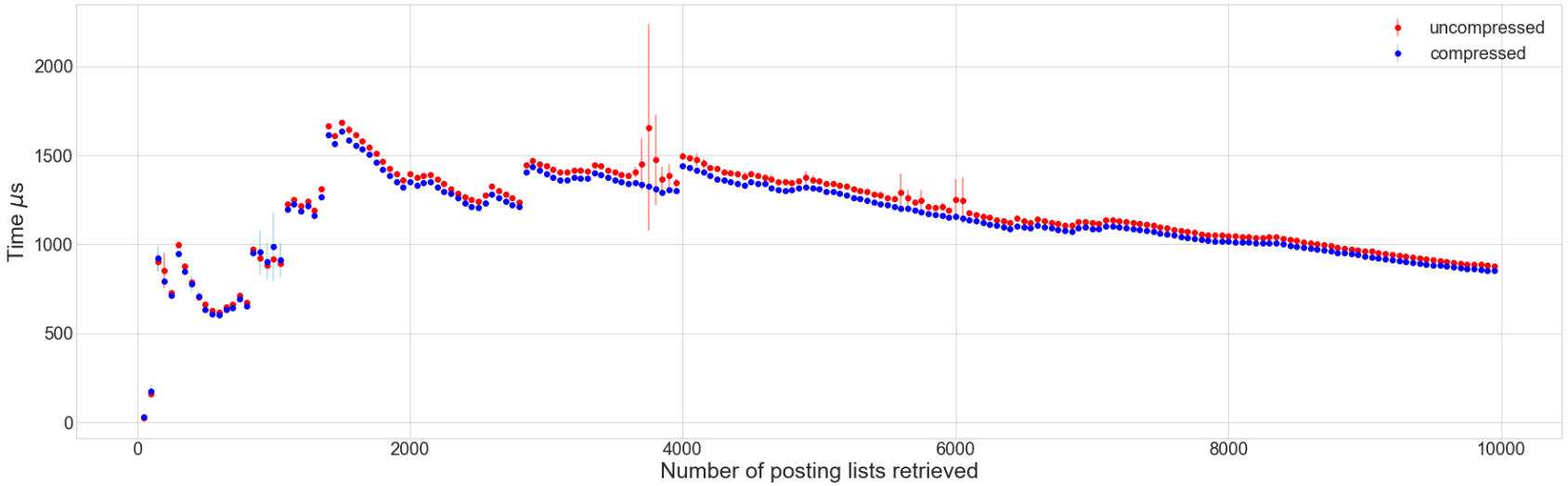
Decoding: scan the byte sequence until a byte whose value is smaller than 128 is found.

```
int VBdecoder(unsigned char* &ptr){
  int shift = 0;
  int number = 0;
  int currentByte = *ptr;
  number |= (currentByte & 0x7f);
  shift += 7;
  ptr++;
  while(currentByte & 0x80){
    currentByte = *ptr;
    number |= (currentByte & 0x7f) << shift;
    shift += 7;
    ptr++;
  }
  return number;
}
```

- ✓ The compressed posting list is read from disk, mmapping the file;
- ✓ Read the first byte;
- ✓ Pointer is advanced as long as the continuation bit is 1.

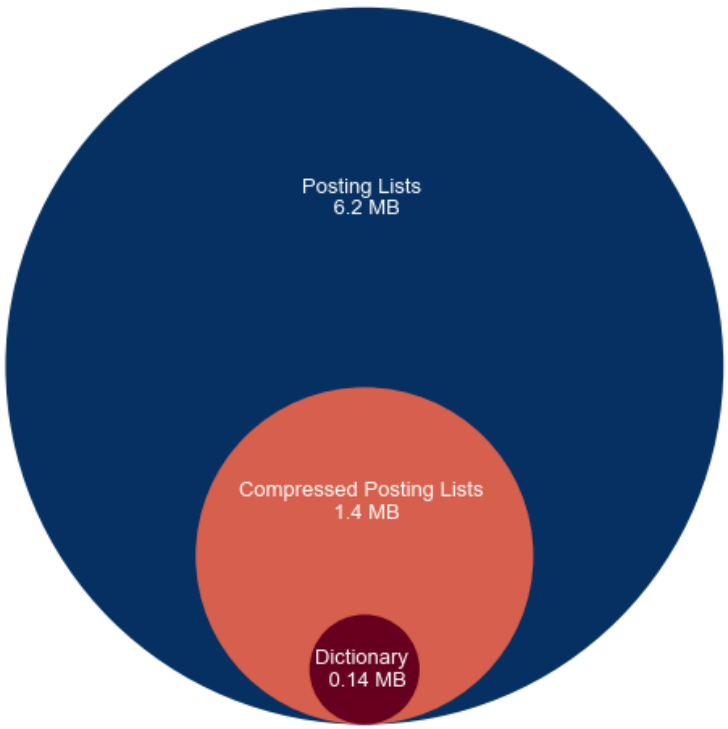
Posting Lists: speed comparison

Retrieving a posting list: compressed vs uncompressed representation



$\rightarrow$ Retrieving a compressed posting list is slightly faster than retrieving a non-compressed one.

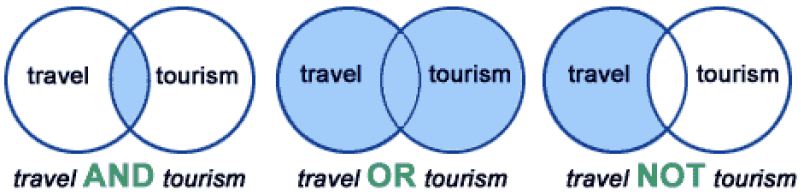
Posting Lists: size comparison



- The size of the compressed posting file is $\approx \frac{1}{5}$ of the uncompressed one;
- Still posting file is bigger than dictionary.

Outline

Answering Queries



Queries without sub-expressions[5].

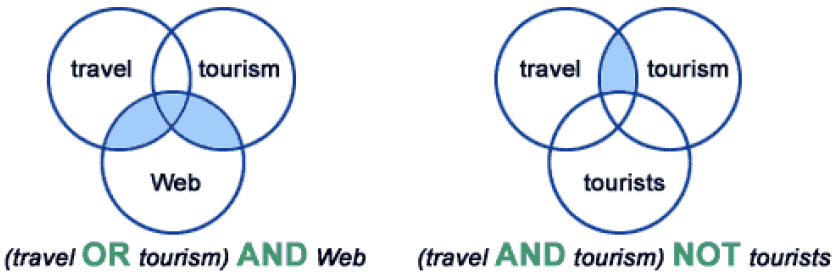
- Locate terms in the dictionary;
- Retrieve their posting lists;
- Perform operations between them.

$\rightarrow$ Complexity $O(n + m)$, with $n =$ length of the posting list of $term_1$, $m =$ length of the posting list of $term_2$.

Normalization and *stemming* steps of the pre-processing pipeline are applied to query terms.

[5] https://www.pinterest.com/pin/238550111489498643/

Answering Queries Cont'd



Queries with sub-expressions[6].

- Locate terms in the dictionary;
- Retrieve their posting lists;
- Evaluate and temporarily store the answers for intermediate expressions;

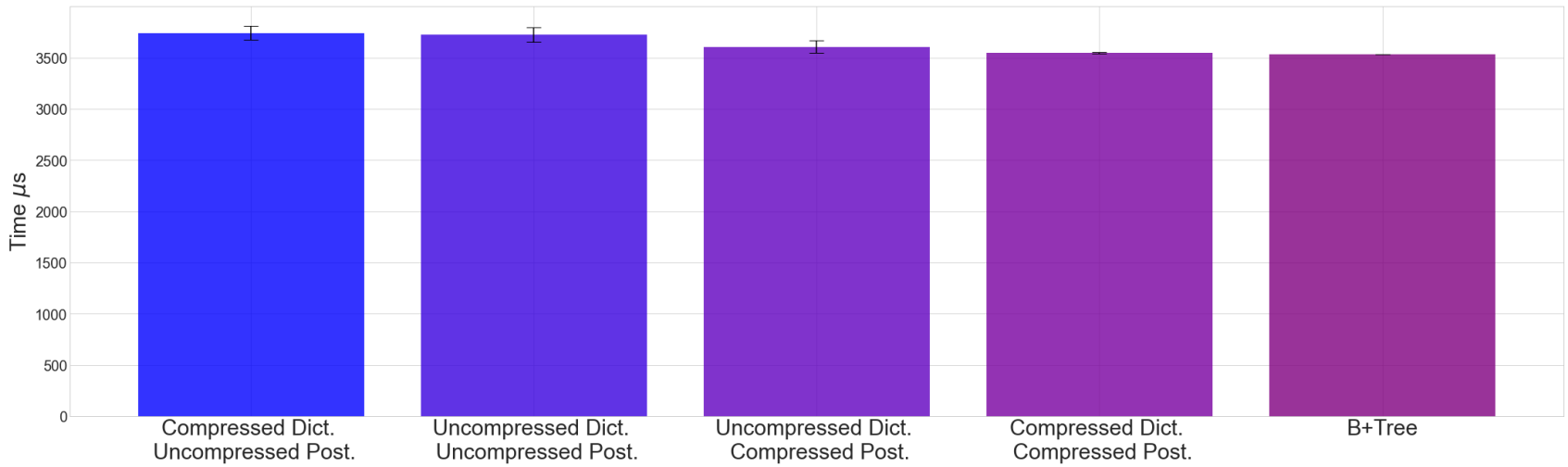
$\rightarrow$ Complexity $O(M)$, with $M = \#$ documents in the collection.

Normalization and *stemming* steps of the pre-processing pipeline are applied to query terms.

[6] https://www.pinterest.com/pin/238550111489498643/

Answering Queries: speed comparison

Considering only queries without sub-expressions:



Thank you for your attention!