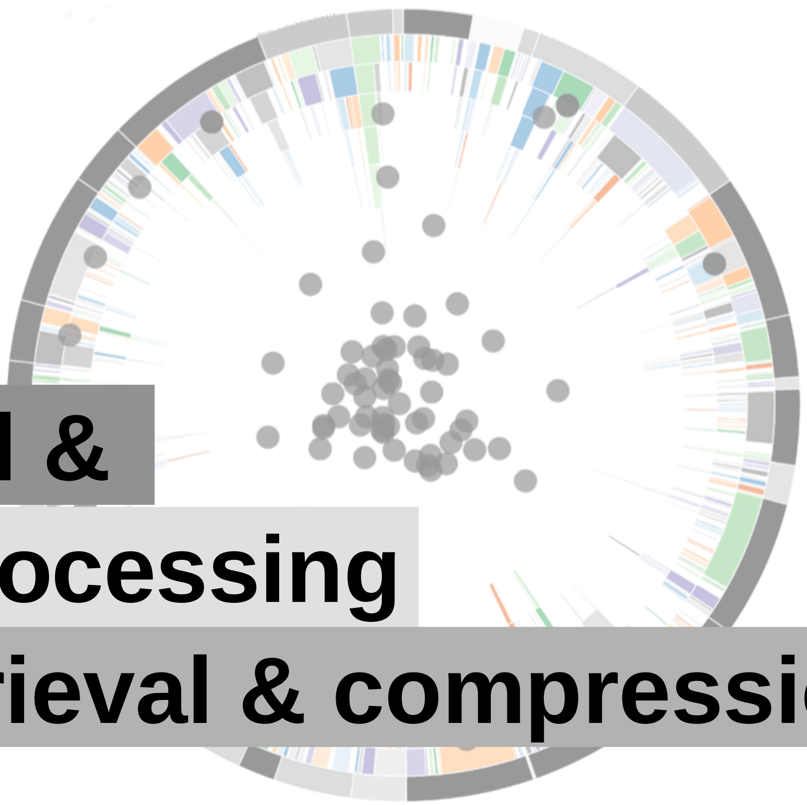
Cluster of Excellence
The Politics of Inequality

Steinbeis Transfer Center
Linguistic Data Analysis

UNIVERSITÄT PASSAU



Information Retrieval & Natural Language Processing
Week 5: Tolerant retrieval & compression

Annette Hautli-Janisz, Prof. Dr.

28 November 2024

Information Retrieval (IR): Today

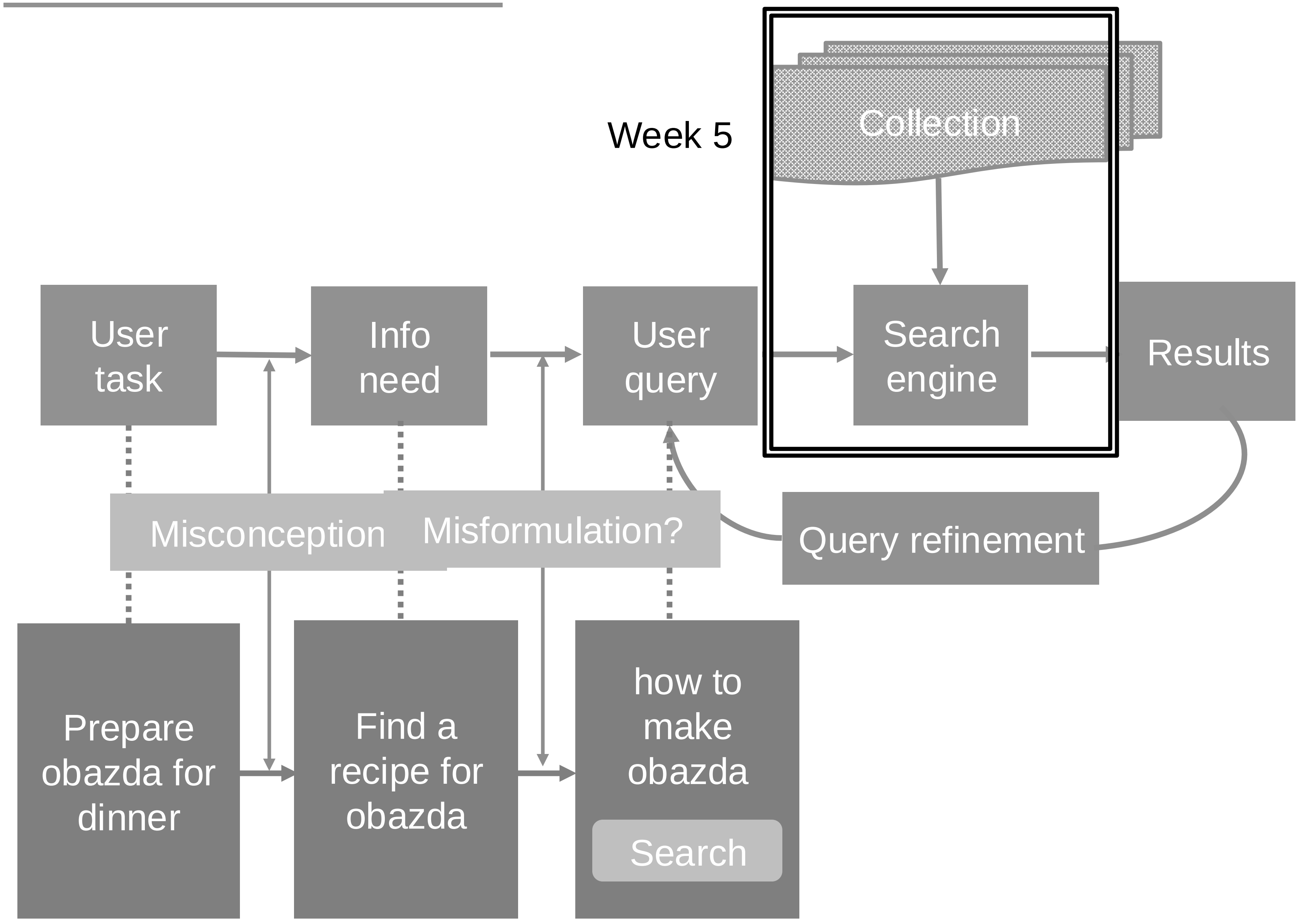
(IIR): Manning, Raghavan and Schütze, Introduction to IR, 2008

Chapter 3: Dictionaries and tolerant retrieval

- How to deal with typos/alternative spellings
- Wildcard queries

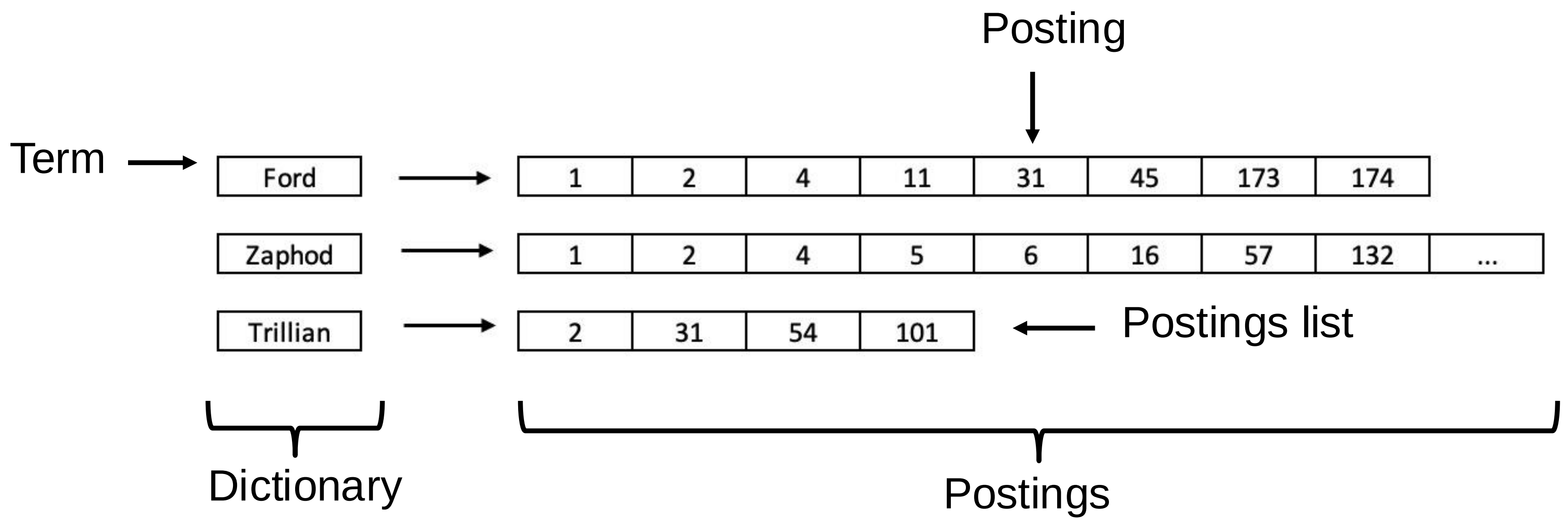
Chapter 5: Index compression

The classic search model



Inverted index

For each term *t* we store a list of documents that contain *t.*



Data structures for term look-up

Two main classes of data structures: hashes and trees

Some IR systems use hashes, some use trees.

Criteria for when to use hashes vs. trees:

- Is there a fixed number of terms or will it keep growing?
- What are the frequencies with which various keys will be accessed?
- How many terms are we likely to have?

Hashes

Each vocabulary term is hashed into an integer.

Try to avoid collisions.

At query time, do the following: hash query term, resolve collisions, locate entry in fixed-width array.

Pros:

- Lookup in a hash is faster than lookup in a tree.
- Lookup time is constant.

Hashes

Cons:

- No way to find minor variants (‘resume’ vs. ‘résumé’)
- No prefix search (all terms starting with ‘automat’)
- Need to rehash everything periodically if vocabulary keeps growing.

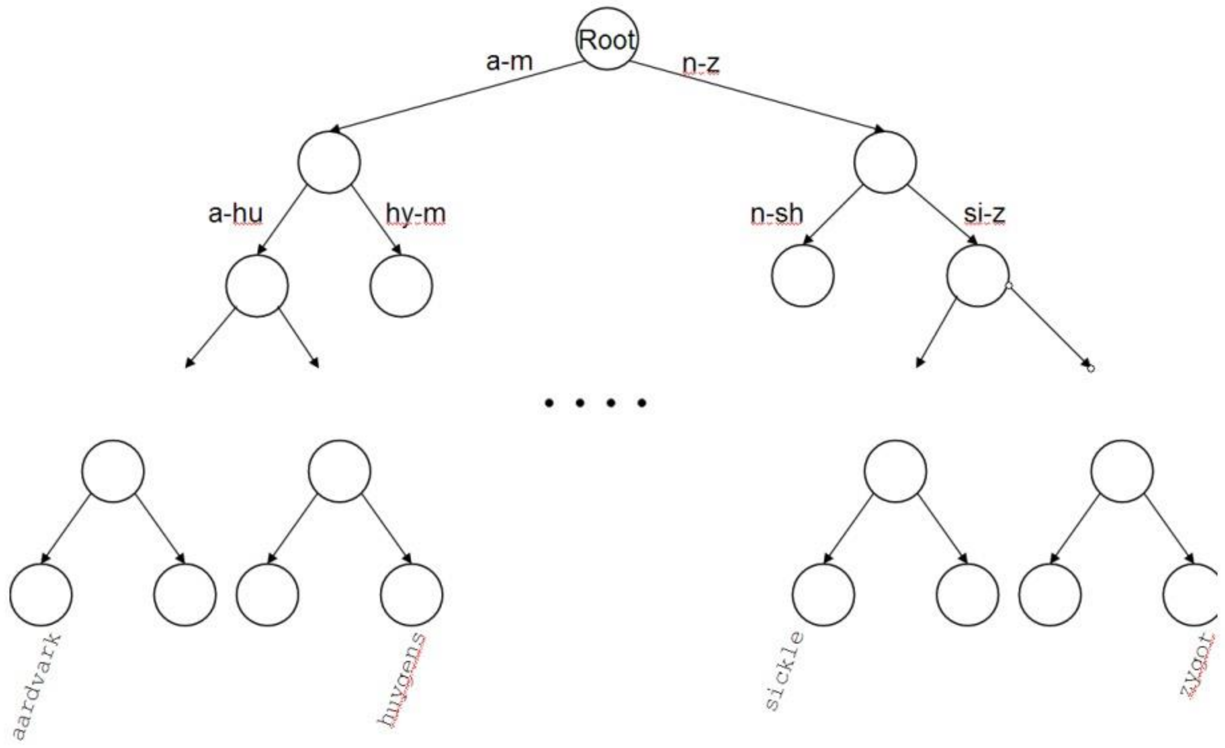
Trees

Trees solve the prefix problem (e.g., find all terms starting with ‘auto’)

B-trees: every internal node has a number of children in the interval [*a, b*], where *a, b* are appropriate positive integers, e.g., [2, 4].

Simplest tree: binary tree

Binary tree



► **Figure 3.1** A binary search tree. In this example the branch at the root partitions vocabulary terms into two subtrees, those whose first letter is between a and m, and the rest.

Wildcard queries

Trailing wildcard queries like *mon**: find all documents containing any term beginning with *mon*

With B-tree dictionary: find all terms *t* in the range of $mon \leq t < moo$

Leading wildcard queries with **mon*: find all docs containing any term ending with *mon*

- Maintain an additional tree for terms *backwards*
- e.g., *lemon* is represented by the path ‘root-n-o-m-e-l’
- Retrieve all terms *t* in the range $nom \leq t < non$

Result: A set of terms that are matches for the wildcard query

Wildcard queries

Example: *m**nchen*

Simple approach: We look up *m** and **nchen* in the backward B-tree and intersect the two sets of terms (this is expensive!)

Alternative: permuterm indexes

Permuterm → term mapping

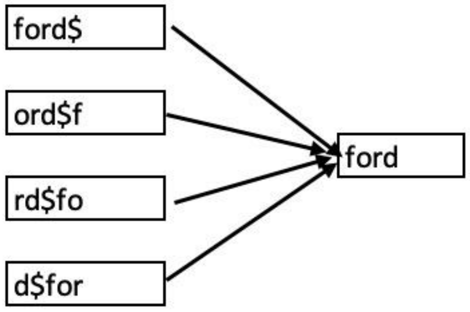
A permuterm index is a form of inverted index.

We use the special symbol ‘$’ to mark the end of a term: *ford* → *ford$*

Permuterm index: Rotate every wildcard query so that * occurs at the end.

Store each of these rotations in the dictionary, say, in a B-tree, and link them all to the original vocabulary term.

This is the set of rotated terms as the permuterm index:



Permuterm → term mapping

Consider the wildcard query *f*d.*

The key is to rotate such a wildcard query so that the * symbol appears at the end of the string → *d$f**

Queries:

- For *X*, look up ___
- For *X**, look up ___
- For **X*, look up ___
- For *X*Y*, look up ___

Processing a lookup in the permuterm index

Rotate query wildcard to the right.

Use B-tree lookup as before.

Problem: Permuterm more than quadruples the size of the dictionary compared to a regular B-tree (empirical estimation)

k-gram indexes

More space-efficient than permuterm index.

Enumerate all character *k*-grams (sequence of *k* characters) occurring in a term (2-grams are called bigrams).

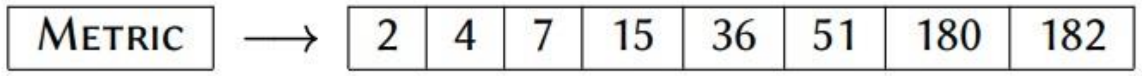
Example: From “*November is the cruelest month*” which bigrams do we get? (‘$’ is again a special word boundary symbol)

Maintain an inverted index from bigrams to the terms that contain the bigram.

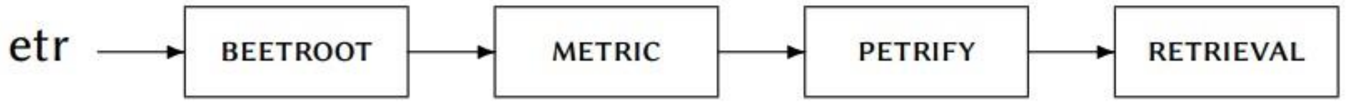
k-gram indexes

Note that now we have two different types of inverted indexes.

The term-document inverted index for finding documents based on a query consisting of terms:



The k-gram index for finding terms based on a query *k*-grams:



***k*-gram indexes**

Query *mon** can now be run as: *$m AND mo AND on*

Gets us all terms with the prefix *mon*, but also many false positives like *moon.*

We must postfilter these terms against query.

Surviving terms are then looked up in term-document inverted index.

k-gram index versus permuterm index

- *k*-gram index is more space efficient
- permuterm index does not require postfiltering

Spelling correction

Two principal uses: Correcting documents being indexed or correcting user queries at query time.

Two different methods for spelling correction:

1. Isolated word spelling correction
   - Check each word on its own for misspelling
   - Will not catch typos resulting in correctly spelled words, e.g., *"an asteroid fell <u>form</u> the sky"*

2. Context-sensitive spelling correction
   - Look at surrounding words
   - Can correct *form/from* error above

Correcting documents vs. correcting queries

We’re not interested in interactive spelling correction of documents.

In IR we use document correction primarily for OCR’ed documents (OCR = Optical Character Recognition).

The general philosophy in IR is: don’t change the documents.

Spelling errors in queries are much more frequent.

Isolated word spelling correction

Premises:

1. There is a list of “correct words” from which the correct spellings come.
2. We have a way of computing the distance between a misspelled word and a correct word.

Simple algorithm: return the correct word that has the smallest distance to the misspelled word, e.g., *informaton* → *information*

For the list of correct words, we can use the vocabulary of all words that occur in our collection.

Why is that problematic?

Alternatives to using term vocabulary

A standard dictionary (Webster’s, OED etc.)

An industry-specific dictionary (for specialized IR systems)

The term vocabulary of the collection, appropriately weighted

Distance between misspelled and “correct” word

There are several alternatives:

- Edit distance and Levenshtein distance
- Weighted edit distance
- *k*-gram overlap

Edit distance

The edit distance between strings s_1 and s_2 is the minimum number of basic operations that convert s_1 to s_2.

Levenshtein: the basic operations are *insert, delete* and *replace*. Go through each character, compare, add 1 if operation is required.

What is the Levenshtein distance in these examples?

- *dog-do*:
- *cat-cart*:
- *cat-cut*:
- *cat-act*:
- *art-cart:*

k-grams for spelling correction

Enumerate all *k*-grams in the query term

Example:

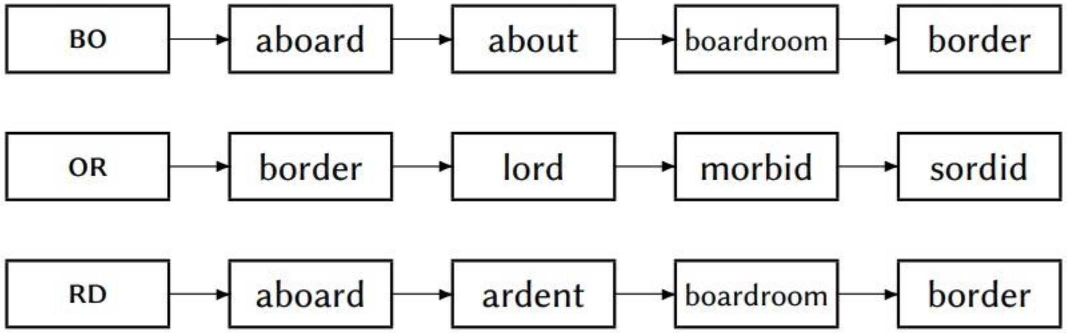
- bigram index, misspelled word: *bord*
- bigrams: *bo, or, rd*

Use the *k*-gram index to retrieve correct words that match query term *k*-grams

Threshold by number of matching *k*-grams (e.g., only vocabulary terms that differ by at most 3 *k*-grams)

k-grams for spelling correction

Suppose we want to retrieve vocabulary terms that contain at least two of the three bigrams of *bord:*



Problem: a term like ‘boardroom’ would get nominated, even though this correction is implausible.

Potential solution: the Jaccard coefficient.

k-grams for spelling correction

What is the Jaccard coefficient between the bigrams of *bord/boardroom* and *bord/aboard*?

General issues in spelling correction

User interface:

- automatic versus suggested correction
- *Did you mean?* only works for one suggestion
- What about multiple possible corrections?
- Tradeoff: simple vs. powerful UI

Cost:

- Spelling correction is potentially expensive
- Avoid running on every query?
- Maybe just on queries that match few documents
- Guess: Spelling correction of major search engines if efficient enough to be run on every query.

Information Retrieval (IR): Today

(IIR): Manning, Raghavan and Schütze, Introduction to IR, 2008

Chapter 3: Dictionaries and tolerant retrieval

- How to deal with typos/alternative spellings
- Wildcard queries

Chapter 5: Index compression

Dictionary compression

Why compression (in general)?

- Use less disk space → save money; give users more space
- Keep more stuff in memory → increases speed
- Increase speed of data transfer from disk to memory ([read compressed data + decompress] is faster than [read uncompressed data])
- Premise: decompression algorithms are fast.

Dictionary compression

Why compression for inverted indexes?

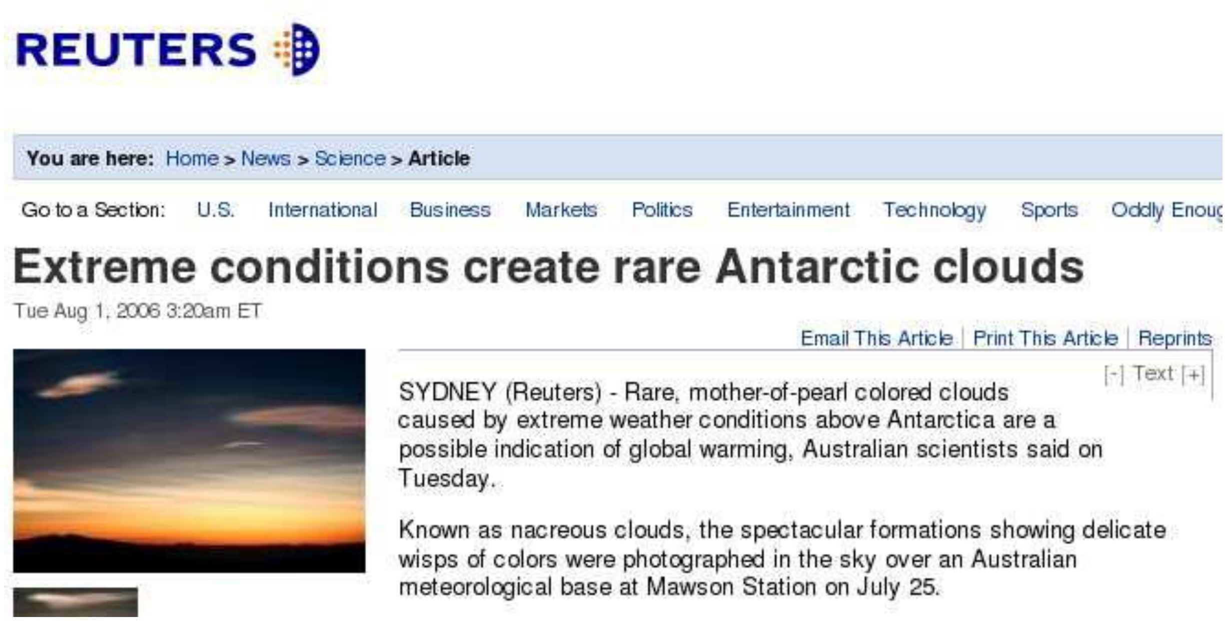
- Dictionary:
  - Make it small enough to keep in main memory
  - Make it so small that you can keep some postings lists in main memory, too

- Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
  - large search engines keep a significant part of the postings in memory

Example: The Reuters RCV1 collection

As an example for applying scalable index construction algorithms: one year of Reuters newswire.

The collection is not really large enough, but it is publicly available and is a plausible example.



REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

Example: The Reuters RCV1 collection

symbol	statistic	value
N	documents	800,000
L	avg # of tokens/doc	200
M	terms (= word types)	~400,000
	avg. # of bytes/token (w/ spaces & punctuation)	6
	avg. # of bytes/token (w/o spaces & punctuation)	4.5
	avg. # of bytes/term	7.5
	non-positional postings	100,000,000

What’s the type/token ratio in RCV1?

Lossless vs. lossy compression

Lossless compression: All information is preserved. (This is what we mostly do in IR)

Lossy compression: Discard some information.

Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.

Vocabulary size vs. collection size

How big is the term vocabulary? (That is, how many distinct words are there?)

Can we assume an upper bound? Not really. In practice, the vocabulary will keep growing with the collection size.

Two laws:

- Heaps’ law: vocabulary size in collections
- Zipf’s law: relative frequencies of terms

Vocabulary size vs. collection size

Heaps' law: $M = kT^b$

M = vocabulary size

T = number of tokens in the collection

$30 \leq k \leq 100$

$b \approx 0.5$

In a log-log plot of vocabulary size *M* vs. *T*, Heaps' law predicts a line with a slope about 0.5.

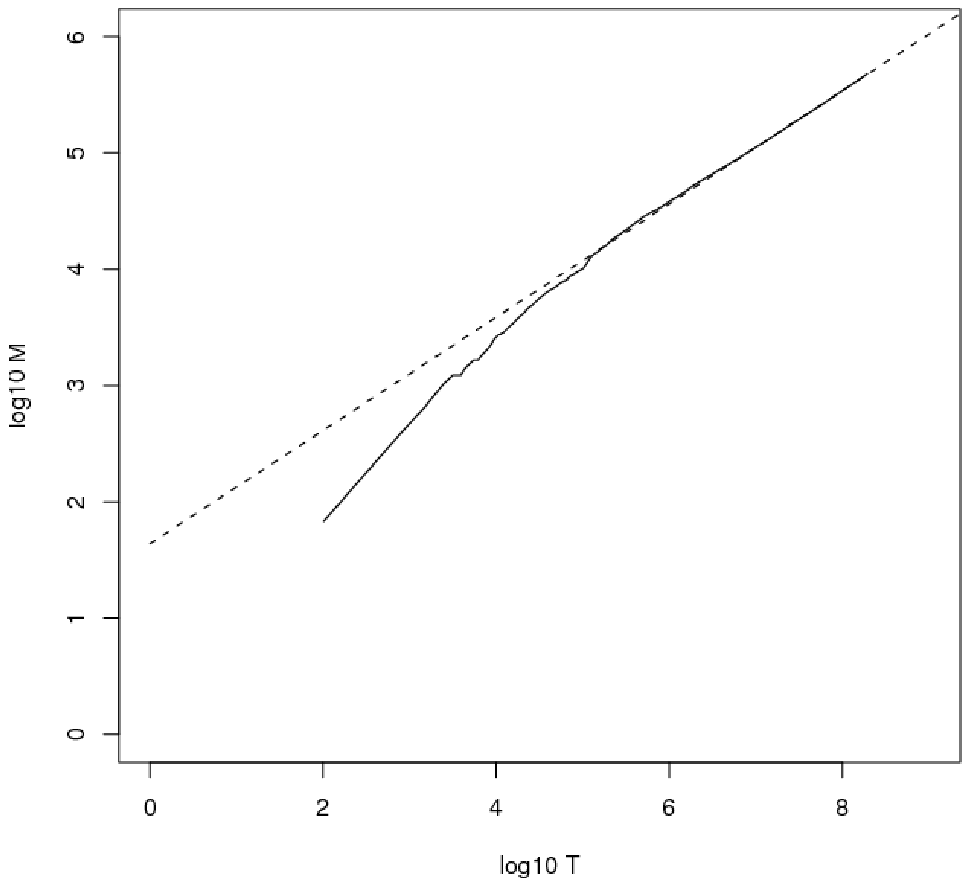
- It is the simplest possible (linear) relationship between the two in log-log space.
- An empirical finding ("empirical law").

Vocabulary size vs. collection size

For RCV1, the dashed line is the best least squares fit.

Good empirical fit for Reuters RCV1.

For the first 1,000,020 tokens, law predicts 38,323 terms, actually: 38,365 terms.



Vocabulary size vs. collection size

Zipf's law: The i^{th} most frequent term has frequency proportional to $1/i$.

Intuition: In natural language, there are a few very frequent terms and very many very rare terms.

$cf_i \propto 1/i = K/i$ (with K as a normalizing constant)

Vocabulary size vs. collection size

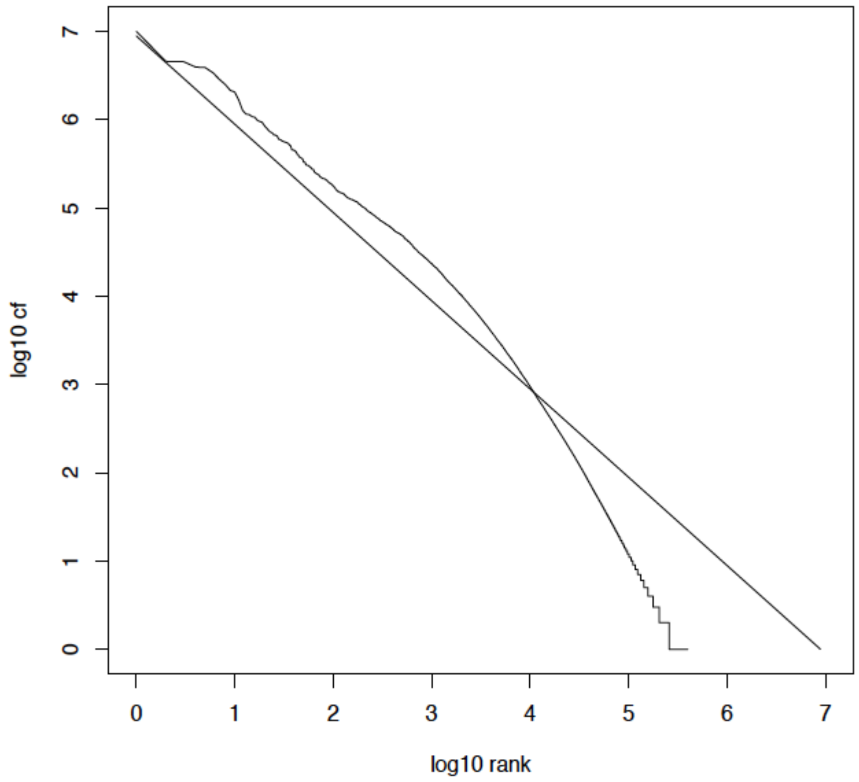
If the most frequent term (*the*) occurs cf_1 times

- then the second most frequent term (*of*) occurs $\mathrm{cf}_1/2$ times
- the third most frequent term (*and*) occurs $\mathrm{cf}_1/3$ times …

Equivalent: $\mathrm{cf}_i = K/i$ where K is a normalizing factor, so

- $\log \mathrm{cf}_i = \log K - \log i$
- Linear relationship between $\log \mathrm{cf}_i$ and $\log i$

Another power law relationship!



Compression

Why compress the dictionary?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have little memory
- Even if the dictionary is not in memory, we want it to be small for a fast search startup time

→ Dictionary compression is important.

Dictionary storage – naïve version

Array of fixed-width entries

~400,000 entries,
28bytes/term (term = 20 bytes; doc freq = 4 bytes; pointer = 4 bytes)

→ 11.2 MB for storing the dictionary in this way

term	doc freq	pointer to postings list
a	656,265	→
aachen	65	→
...	...	→
zulu	221	→

Recap, CS101: bits and bytes.

Dictionary storage – naïve version

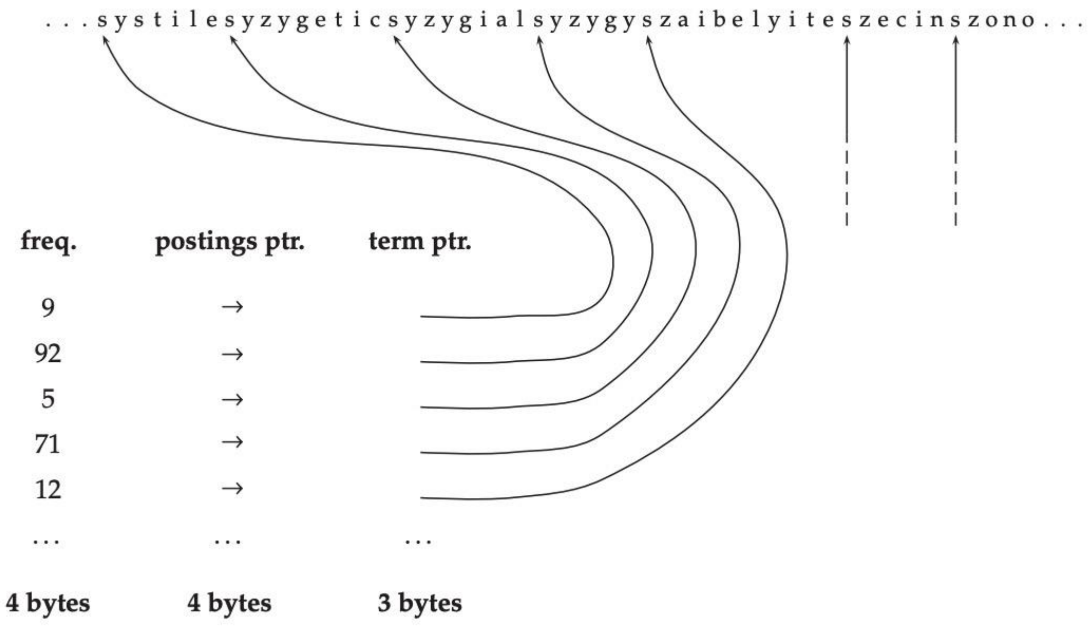
Fixed-width terms are wasteful.

- Most of the bytes in the term column are wasted – we allot 20 bytes for 1 letter terms (and we still can’t handle *supercalifragilisticexpialidocious*...)
- Written English averages ~4.5 characters/word
- Avg. dictionary word in English: ~8 characters – use this!

Dictionary compression – dictionary-as-a-string

One way to overcome the fixed-width storage problem: Store the dictionary as one string.

Pointers to mark term boundaries.


. . . s y s t i l e s y z y g e t i c s y z y g i a l s y z y g y s z a i b e l y i t e s z e c i n s z o n o . . .
freq.
postings ptr.
term ptr.
9
92
5
71
12
4 bytes
4 bytes
3 bytes

► **Figure 5.4** Dictionary-as-a-string storage. Pointers mark the end of the preceding term and the beginning of the next. For example, the first three terms in this example are systile, syzygetic, and syzygial.

Dictionary compression – dictionary-as-a-string

Saves up to 60% of space compared to fixed-width-storage:

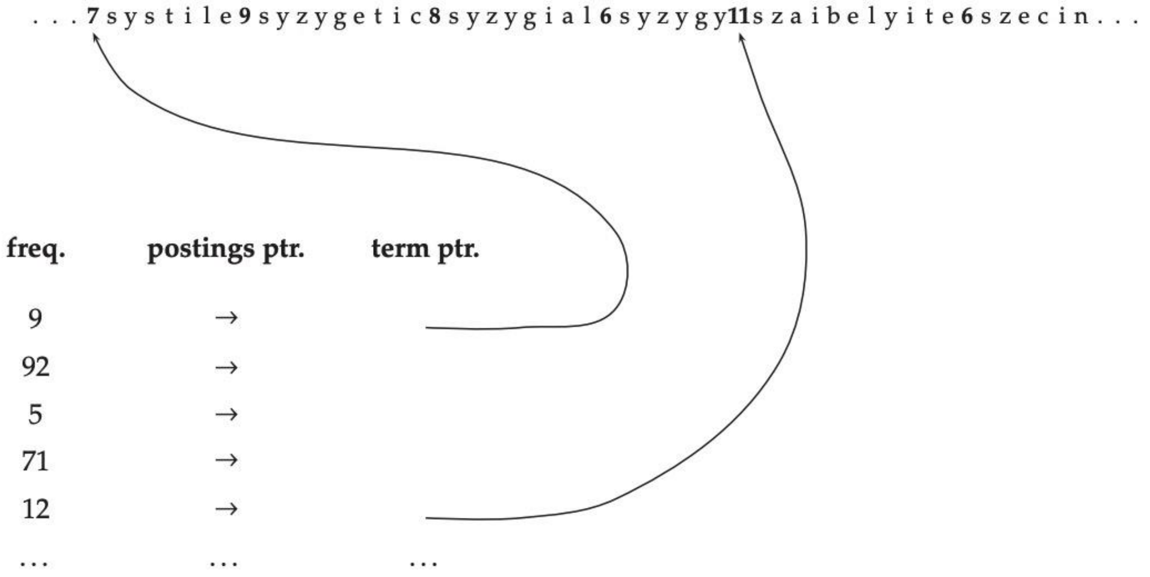
- 4 bytes per term to store doc freq
- 4 bytes per term for pointer to postings
- 3 bytes per term to store term boundary pointer (explanation of why in the book)
- avg. 8 bytes per term in term string

→ 400K terms x 19 bytes = 7.6MB (compared to 11.2MB for fixed width)

Dictionary compression – blocked storage

Further compression possible by grouping terms in the string onto blocks of size *k* and keeping a term pointer only for the first term of each block.

Store the length of the term in the string as an additional byte at the beginning of the term.


. . . 7 s y s t i l e 9 s y z y g e t i c 8 s y z y g i a l 6 s y z y g y 11 s z a i b e l y i t e 6 s z e c i n . . .
freq.
postings ptr.
term ptr.
9
92
5
71
12
. . .
. . .
. . .

► **Figure 5.5** Blocked storage with four terms per block. The first block consists of systile, syzygetic, syzygial, and syzygy with lengths of seven, nine, eight, and six characters, respectively. Each term is preceded by a byte encoding its length that indicates how many bytes to skip to reach subsequent terms.

Dictionary compression – blocked storage

Blocking net gains (example for block size *k* = 4):

Without blocking: 3 bytes/pointer to term boundary

With blocking: We can eliminate *k* – 1 term pointers, but we need an additional *k* bytes for storing the length of each term.

- For *k* = 4, we save (*k* – 1) x 3 = 9 bytes for term pointers
- Add *k* = 4 bytes for term lengths

→ space reduced by 5 bytes per four-term block, or a total of 400,000 x ¼ x 5 bytes = 0.5MB

→ 7.1MB storage with blocked storage

Postings file compression

The postings file (or postings list) is much larger than the dictionary, factor of at least 10, often over 100 times larger.

symbol	statistic	value
N	documents	800,000
L	avg # of tokens/doc	200
M	terms (= word types)	~400,000
	avg. # of bytes/token (w/ spaces & punctuation)	6
	avg. # of bytes/token (w/o spaces & punctuation)	4.5
	avg. # of bytes/term	7.5
	non-positional postings	100,000,000

Postings file compression

Key desideratum: store each posting compactly.

A posting for our purposes is a docID

Example inverted list (docIDs only):

3, 17, 21, 24, 34, 38, 45, ..., 11876, 11899, 11913, ...

Numbers small in the beginning and large in the end, using an integer requires 4 bytes or even 8 bytes per id (in the case of web search)

Postings file compression

One solution: gap encoding.

Only store differences from one docID to the next:

+3, +14, +4, +3, +10, +4, +7, ..., +12, +23, +14, ...

Works as long as we process the lists from left to right.

Now we have a sequence of mostly (but not always) small numbers... how do we store these in little space?

Binary representation: We can write number x in binary using ($\log_2 x + 1$) bits

Postings file compression

x	binary	number of bits
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3

This encoding is optimal.

So why not just (gap-)encode like this and concatenate:

+3, +14, +4, ... → 11, 1110, 100, ... → 111110100...

What’s the problem now?

Postings file compression

Prefix-free codes, definition.

- Decode bit sequence from last slide: 111110100

  This could be: +3, +14, +4

  Could also be: +7, +6, +4

  Or: +3, +3, +2, +4

Problem: we have no way to tell where one code ends and the next code begins.

Or: some codes are prefixes of other codes.

One solution: v-byte encoding (more details in the book, p. 88ff)

V-byte encoding

Small deltas (between postings) more frequent than large ones.

→ The core idea of v-byte encoding is to use fewer bytes for smaller numbers (aka. variable number of bytes depending how large your number is)

1 byte: 0 ... 2^7
2 bytes: 2^7 ... 2^{14}
3 bytes: 2^{14} ... 2^{21}
4 bytes: 2^{21} ... 2^{28}

V-byte encoding

Instead of using 8 bits in a byte,

- we use the 7 bits in a byte to represent the number itself
- the high bit has a special function: records whether this byte is the last byte of an encoding (high bit = 1) or whether you need more bytes (high bit = 0)

  - “6” → 10000110
  - “127” → 11111111
  - “128” → 0000000110000000

V-byte encoding

For decoding,

- identify the boundaries of numbers by looking at the high bits
- throw away the high bits and read off the remaining representation

Example string:

10000001 00000001 10000010 10000010

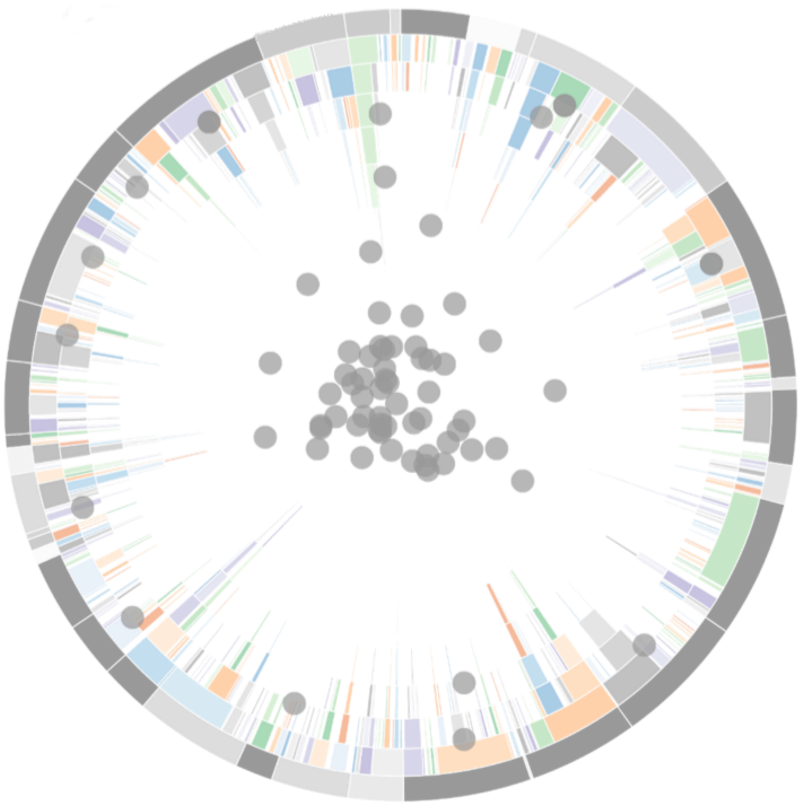
Which numbers are encoded by this string, using v-byte encoding?

Cluster of Excellence
The Politics of Inequality

Steinbeis Transfer Center
Linguistic Data Analysis

UNIVERSITÄT PASSAU

Thank you.
Questions?
Comments?



Annette Hautli-Janisz, Prof. Dr.
cornlp-teaching@uni-passau.de