05 – Indexing and Searching Very Large Texts

IA161 Advanced Techniques of Natural Language Processing

M. Jakubíček

NLP Centre, FI MU, Brno

October 16, 2019

Searching big text corpora

Corpus:

- positional attributes – word form, lemma, PoS tag, . . .
- structures and structure attributes – documents (e.g. with author, id, year, . . .), paragraph, sentence
- searching: Manatee/Bonito/Sketch Engine
- `http://corpora.fi.muni.cz`
- `https://app.sketchengine.eu`
- SQL unsuitable (independent rows)

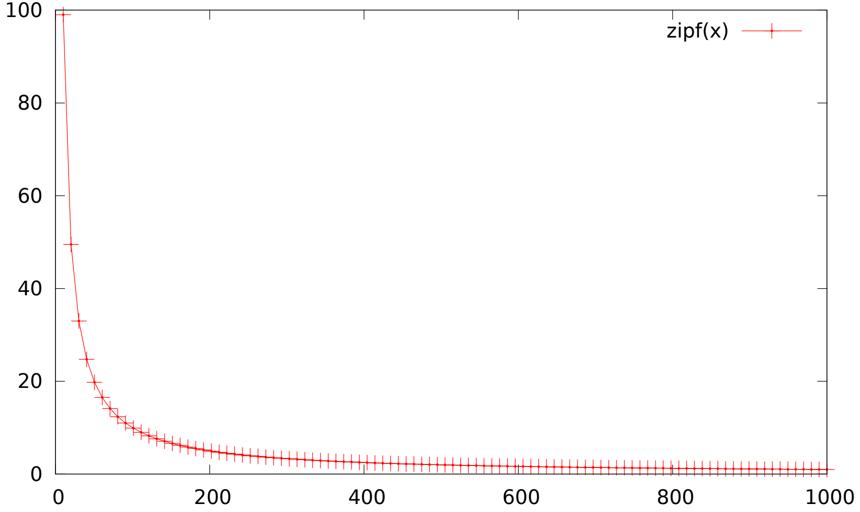
Searching big text corpora

- data too big to be stored in memory
- data too big to be search sequentially

⇒ preprocessing needed (indexing, alias corpus compilation)

- key decisions are:
  - trade off between compile-time (preprocessing) and run-time
  - trade off between in memory and off-memory processing

Zipf's law I



Zipf's law II

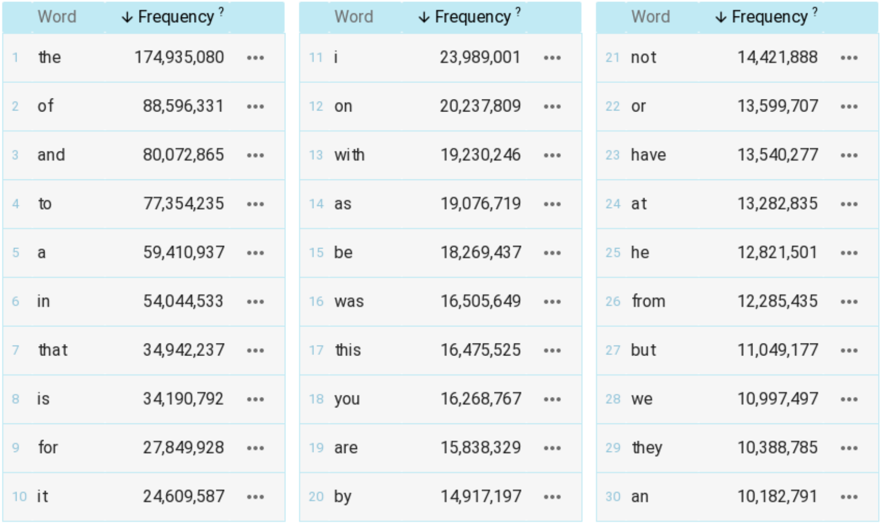
- may be simplified to inductive definition:

Zipf's law (simplified)

frequency of the *n*-th element $f_n \approx \frac{1}{n} \cdot f_1$

- $\Rightarrow$ frequency is inversely proportional to the rank according to frequency
- $\Rightarrow$ one needs really large corpora to capture all the variety of many language phenomena

Zipf's law III



	Word	↓ Frequency ?
1	the	174,935,080
2	of	88,596,331
3	and	80,072,865
4	to	77,354,235
5	a	59,410,937
6	in	54,044,533
7	that	34,942,237
8	is	34,190,792
9	for	27,849,928
10	it	24,609,587
11	i	23,989,001
12	on	20,237,809
13	with	19,230,246
14	as	19,076,719
15	be	18,269,437
16	was	16,505,649
17	this	16,475,525
18	you	16,268,767
19	are	15,838,329
20	by	14,917,197
21	not	14,421,888
22	or	13,599,707
23	have	13,540,277
24	at	13,282,835
25	he	12,821,501
26	from	12,285,435
27	but	11,049,177
28	we	10,997,497
29	they	10,388,785
30	an	10,182,791

enTenTen2008, 3.2G tokens

Zipf's law IV

About 1 billion words is enough to have enough evidence for single word units. But not for multiwords:

word	**Brown (1M)**	**BNC (100M)**	**enTenTen08 (2.7G)**	**enTenTen15 (15.7G)**
carbonation	0	5	429	2,817
weird phrase	0	0	14	34

Building corpora

1. content definition (what will it be used for? how do I get texts?)
2. obtaining data (e.g. crawling)
3. data cleaning (spam, boilerplate, duplicates)
4. tokenization
5. sentence segmentation
6. further annotation (PoS tagging)
7. corpus indexing and analysis

Building corpora

1. content definition (what will it be used for? how do I get texts?)
2. obtaining data (e.g. crawling)
3. data cleaning (spam, boilerplate, duplicates)
4. tokenization
5. sentence segmentation
6. further annotation (PoS tagging)
7. corpus indexing and analysis

Corpus indexing

- text corpus is a database
- standard (=relational) database management systems are not suitable at all
    - text corpus does not have relational nature
- special database management systems needed

$\Rightarrow$ Manatee

Indexing corpora in Manatee

Key data structures for a positional attribute:

- lexicon
  - because operations on numbers are just so much faster than on strings
- corpus text
  - to iterate over positions
- inverted (reversed) index
  - to give fast access to positions for a given value

How to store integer numbers

- given Zipf's distribution: fixed-length storing very inefficient
- variable-length more complicated but yielding much smaller and quicker indices
- variable-length bit-wise universal Elias' codes: gamma, delta codes
- cf. Huffman coding

Indexing corpora in Manatee

Structures and operations:

- operations in between: string (`str`) – number (`id`) – position (`poss`)
- lexicon building: ⇒ word-to-id mapping ⇒ operations on numbers, not strings ⇒ `id2str, str2id`
- inverted index: `id2poss`
- corpus text: `pos2id`
- yields transitively also `pos2str, str2poss`

Searching corpora in Manatee

- key idea: operations on sorted forward-only streams of positions
- FastStream – single position stream
- RangeStream – stream of position pairs (structures: *from* position, *to* position)

CQL

- = Corpus Query Language (Christ and Schulze, 1994)
- positions and positional attributes: `[attr="value"]`
- structures and structural attributes: `<str attr="value">`
- example:

```
[word=".*ing" & tag="V.*"]
   <doc id="20[5-9].*"
```

- established a `within <str/>` query:

```
[tag="N.*"]+ within <s/>
```

  and alternative `meet/union` query:

```
(meet [lemma="take"] [tag="N.*"] -5 +5)
    (union (meet ...)  (meet ...))
```

CQL in Manatee/Bonito

- ehnancements and differences to the original CQL syntax
- `within <query>` and `containing <query>`
- `meet/union` (sub)query
- inequality comparisons
- frequency function

within/containing queries

- searching for particles:

  ```
  [tag="PR.*"] within [tag="V.*"] [tag="AT0"]?
  [tag="AJ0"]* [tag="(PR.?|N.*)"] [tag="PR.*"] within
  <s/>
  ```

- searching for a Czech idiom “hnout někomu žlučí” (“to get somebody's goat”):
  word-by-word translated as:
  hnout “move” [V, infinitive]
  někomu “somebody” [N, dative]
  žlučí “bile” [N, instrumental].

  ```
  <s/> containing [lemma="hnout"] containing
  [tag=".*c3.*"] containing [word="žlučí"]
  ```

`within`/`containing` queries

- structure boundaries: begin: `<str>`, whole structure: `<str/>`, end: `</str>`
- changes: `within <str>` not allowed anymore, use `within <str/>`

meet/union queries

- combined with regular query: `<s/>`
  ```
  containing (meet [lemma="have"] [tag="P.*"] -5 5)
  containing (meet [tag="N.*"] [lemma="blue"])
  ```
- changes: meet/union queries can be used on any position, they can contain labels and no MU keyword is required (and deprecated):
  `(meet 1:[] 2:[]) & 1.tag = 2.tag`

Inequality comparisons

- former comparisons allowed only equality and its negation: `[attr="value"] [attr!="value"]`
- inequality comparisons implemented: `[attr<="value"] [attr>="value"] [attr!<="value"] [attr!>="value"]`
- intended usage:
  `[tag="AJ.*"] [tag="NN.*"] within <doc year>="2009">`
- sophisticated comparison performed on the attribute value: `<doc id<="CC20101031B">` matches e.g. `BB20101031B, CC20091031B, CC20101030B CC20101031A`.

Fixed string comparisons

- normally the CQL values are regular expressions
- sometimes this is not desirable (batch processing needs escaping of metacharacters)
- new == and !== operator introduced for fixed strings comparison
- no escaping needed except for '"' and '\'
- examples: ".", "$", " " matches a single dot, dollar sign and tilda, respectively, "\n" matches a backslash followed by the character n,

Frequency function

- a frequency constraint allowed in the global conditions part of CQL:
  `1:[tag="PP.*"] 2:[tag="NN.*"] & f(1.word) > 10`

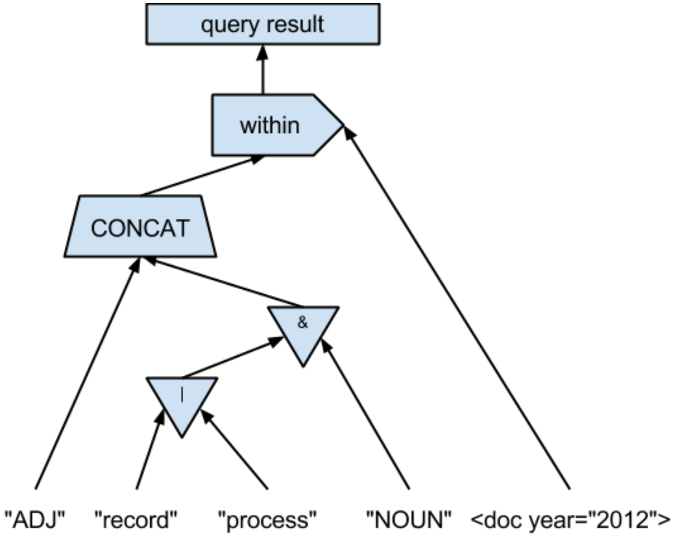
Performance evaluation

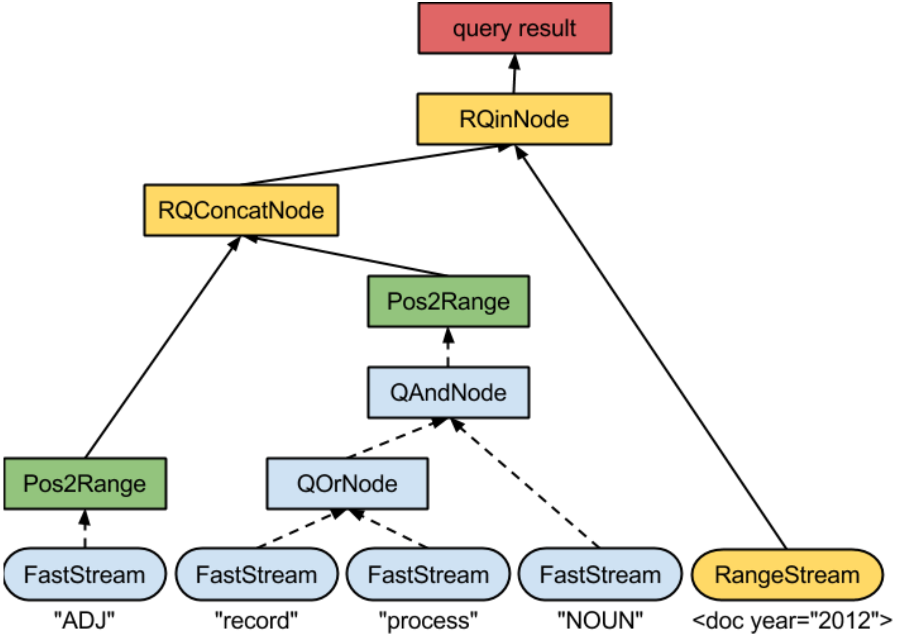
Table: Query performance evaluation – corpora legend: ○ BNC (110M tokens), • BiWeC (version with 9.5G tokens), ∗ Czes (1.2G tokens)

query	# of results	time (m:s)
○ `[lemma="time"]`	179,321	0.07
○ `[lemma="t.*"]`	14,660,881	3.12
○ Ex: particles	1,219,973	33.36
• Ex: particles	97,671,485	32:26.48
∗ Ex: idioms	66	1:6.86
○ Ex: meet/union	3	8.47
• Ex: meet/union	1457	7:13.12

CQL query evaluation

Example: `[tag="ADJ"] [(word="record" | word="process") & tag="NOUN"] within <doc year="2012"/>`




query result
RQinNode
RQConcatNode
Pos2Range
Pos2Range
QAndNode
QOrNode
FastStream
"ADJ"
FastStream
"record"
FastStream
"process"
FastStream
"NOUN"
RangeStream
<doc year="2012">

Conclusions

- special database management systems for processing text corpora needed
- trade-offs between compile-time and run-time, in-memory and off-memory
- CQL
- Manatee

Assignment

https://pastebin.com/Lt0xbhjZ