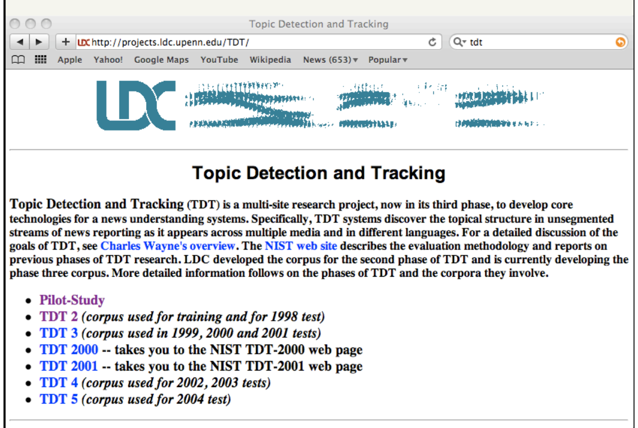
Index Compression

David Kauchak
cs458
Fall 2012
adapted from:
http://www.stanford.edu/class/cs276/handouts/lecture5-indexcompression.ppt

Administrative

- Assignment 1?
- Homework 2 out

- "What I did last summer" lunch talks today


Topic Detection and Tracking
http://projects.ldc.upenn.edu/TDT/

Topic Detection and Tracking

Topic Detection and Tracking (TDT) is a multi-site research project, now in its third phase, to develop core technologies for a news understanding systems. Specifically, TDT systems discover the topical structure in unsegmented streams of news reporting as it appears across multiple media and in different languages. For a detailed discussion of the goals of TDT, see Charles Wayne's overview. The NIST web site describes the evaluation methodology and reports on previous phases of TDT research. LDC developed the corpus for the second phase of TDT and is currently developing the phase three corpus. More detailed information follows on the phases of TDT and the corpora they involve.

- Pilot-Study
- TDT 2 *(corpus used for training and for 1998 test)*
- TDT 3 *(corpus used in 1999, 2000 and 2001 tests)*
- TDT 2000 -- takes you to the NIST TDT-2000 web page
- TDT 2001 -- takes you to the NIST TDT-2001 web page
- TDT 4 *(corpus used for 2002, 2003 tests)*
- TDT 5 *(corpus used for 2004 test)*

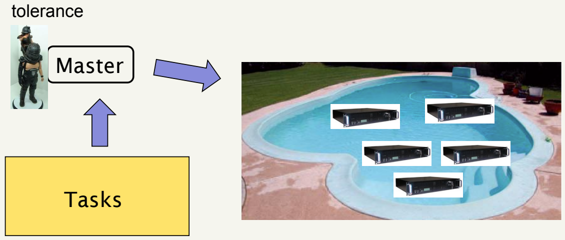
Distributed indexing

Maintain a *master* machine directing the indexing job

Break up indexing into sets of (parallel) tasks

Master machine assigns each task to an idle machine from a pool

Besides speed, one advantage of a distributed scheme is fault tolerance



Distributed indexing

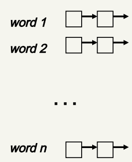
Quick refresh of the non-parallelized approach:

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



1. create term list 2. sort term list 3. create postings list

Distributed indexing

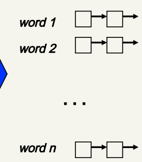
Split into smaller, parallelizable chunks

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



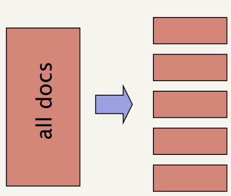
1. create term list 2. sort term list 3. create postings list

Parallel tasks

We will use two sets of parallel tasks

- Parsers (Step 1: create term list)
- Inverters (Steps 2-3: sort term list, create postings list)

split documents up for parsers



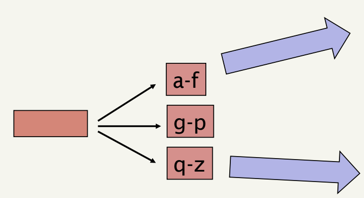
Parsers

Read a document at a time and emits (term, doc) pairs (Step 1)

Parser writes pairs into *j* partitions

Each partition is for a range of terms' first letters

- (e.g., ***a-f, g-p, q-z***) – here *j*=3.



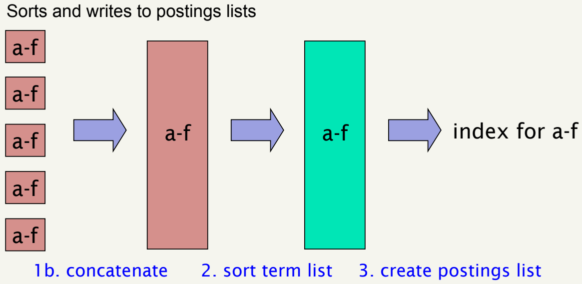
did	1
enact	1
caesar	1
capitol	1
brutus	1
be	2
caesar	2
brutus	2
caesar	2

was	1
the	1
so	2
with	2
told	2
was	2

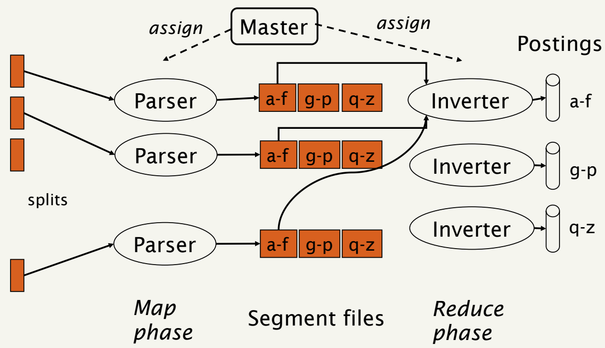
Inverters

Collects all (term, doc) pairs for one term-partition

Sorts and writes to postings lists



Data flow



MapReduce

MapReduce (Dean and Ghemawat 2004) is a robust and simple framework for distributed computing without having to write code for the distribution part

The Google indexing system (ca. 2002) consists of a number of phases, each implemented in MapReduce

MapReduce and similar type setups are hugely popular for web-scale development!

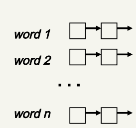
MapReduce

Index construction is just one phase
After indexing, we need to be ready to answer queries

There are two ways to we can partition the index:

- *Term-partitioned:* one machine handles a subrange of terms
- *Document-partitioned:* one machine handles a subrange of documents

Which do you think search engines use? Why?



Index compression

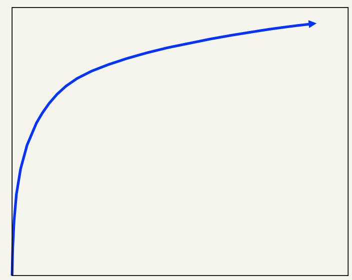
Compression techniques attempt to decrease the space required to store an index

What other benefits does compression have?

- Keep more stuff in memory (increases speed)
- Increase data transfer from disk to memory
  - [read compressed data and decompress] is faster than [read uncompressed data]
  - What does this assume?
    - Decompression algorithms are fast
    - True of the decompression algorithms we use

How does the vocabulary size grow with the size of the corpus?

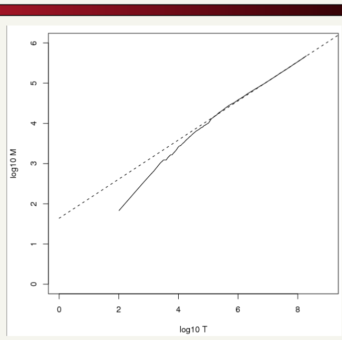
vocabulary size



number of documents

How does the vocabulary size grow with the size of the corpus?

log of the vocabulary size



log of the number of documents

Heaps' law

vocab size = k (tokens)^b

$V = k\, T^b$

Typical values:
$30 \leq k \leq 100$
$b \approx 0.5$

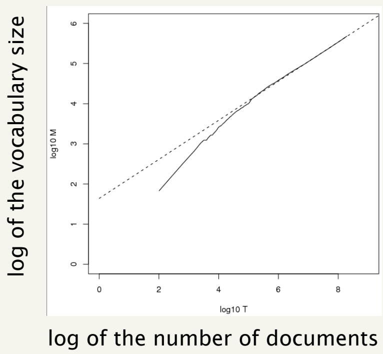
Does this explain the plot we saw before?

$\log V = \log k + b \log(T)$

What does this say about the vocabulary size as we increase the number of documents?

- there are almost always new words to be seen: increasing the number of documents increases the vocabulary size
- to get a linear increase in vocab size, need to add exponential number of documents

vocab growth vs. size of the corpus



$\log_{10} M = 0.49 \log_{10} T + 1.64$
is the best least squares fit.

$M = 10^{1.64} T^{0.49}$

$k = 10^{1.64} \approx 44$
b = 0.49.

Discussion

vocab size = k (tokens)^b
V = k T^b

Typical values:
$30 \leq k \leq 100$
$b \approx 0.5$

How do token normalization techniques and similar efforts like spelling correction interact with Heaps' law?

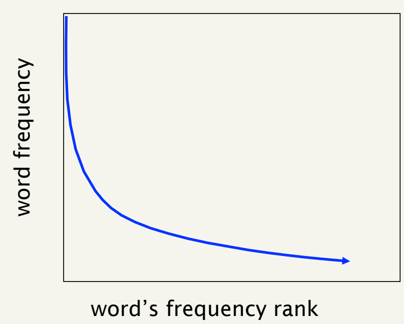
Heaps' law and compression

index compression is the task of reducing the memory requirement for storing the index

What implications does Heaps' law have for compression?

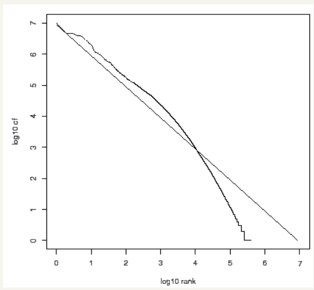
- Dictionary sizes will continue to increase
- Dictionaries can be very large

How does a word's frequency relate to it's frequency rank



How does a word's frequency relate to it's frequency rank

log of the frequency



log of the frequency rank

Zipf's law

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

Zipf's law: The *i* th most frequent term has frequency proportional to 1/*i*

$$\text{frequency}_i \propto c/i$$

where *c* is a constant

$$\log(\text{frequency}_i) \propto \log c - \log i$$

Consequences of Zipf's law

If the most frequent term (*the*) occurs cf_1 times, how often do the 2nd and 3rd most frequent occur?

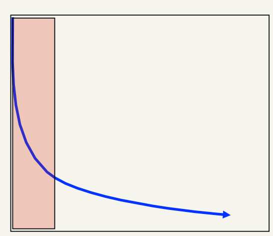
- then the second most frequent term (*of*) occurs $cf_1/2$ times
- the third most frequent term (*and*) occurs $cf_1/3$ times …

If we're counting the number of words in a given frequency range, lowering the frequency band linearly results in an exponential increase in the number of words

Zipf's law and compression

What implications does Zipf's law have for compression?

word frequency

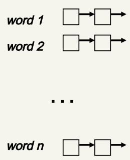


word's frequency rank

Some terms will occur **very** frequently in positional postings lists

Dealing with these well can drastically reduce the index size

Compresssing the inverted index



What do we need to store?

How are we storing it?

Compressing the inverted index

Two things to worry about:

dictionary:

- make it small enough to keep in main memory
- Search begins with the dictionary

postings:

- Reduce disk space needed, decrease time to read from disk
- Large search engines keep a significant part of postings in memory

Lossless vs. lossy compression

What is the difference between lossy and lossless compression techniques?

Lossless compression: All information is preserved

Lossy compression: Discard some information, but attempt to keep information that is relevant

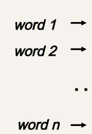
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.
- Prune postings entries that are unlikely to turn up in the top *k* list for any query

Where else have you seen lossy and lossless compresion techniques?

The dictionary

If I asked you to implement it right now, how would you do it?

How much memory would this use?



The dictionary

Array of fixed-width entries

~400K terms; 28 bytes/term = 11.2 MB.

Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	

20 bytes (assume 1byte chars)

4 bytes each (assuming 32-bit)

Fixed-width terms are wasteful

Any problems with this approach?

- Most of the bytes in the Term column are wasted – we allocate 20 bytes for 1 letter terms
  - And we still can't handle supercalifragilisticexpialidocious

Written English averages ~4.5 characters/word

- Is this the number to use for estimating the dictionary size?

Ave. dictionary word in English: ~8 characters

- Short words dominate token counts but not type average

Any ideas?

Store the dictionary as one long string

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

Gets ride of wasted space

If the average word is 8 characters, what is our savings over the 20 byte representation?

- Theoretically, 60%
- Any issues?

Dictionary-as-a-String

Store dictionary as a (long) string of characters:

- Pointer to next word shows end of current word

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		

Total string length = 400K x 8B = 3.2MB

Pointers resolve 3.2M positions: $\log_2 3.2M$ = 22bits = 3bytes

How much memory to store the pointers?

Space for dictionary as a string

Fixed-width

- 20 bytes per term = 8 MB

As a string

- 5.6 MB (3.2 for dictionary and 2.4 for pointers)

30% reduction!

Still a long way from 60%. Any way we can store less pointers?

Blocking

Store pointers to every *k*th term string

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		

What else do we need?

Blocking

Store pointers to every *k*th term string

- Example below: k = 4

Need to store term lengths (1 extra byte)

….*7systile9syzygetic8syzygial6syzygy11szaibelyite8szczecin9szomo*….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

Net

Where we used 3 bytes/pointer without blocking

- 3 x 4 = 12 bytes for *k*=4 pointers,

now we use 3+4=7 bytes for 4 pointers.

Shaved another ~0.5MB; can save more with larger *k*.

Why not go with larger *k*?

Dictionary search without blocking

How would we search for a dictionary entry?

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

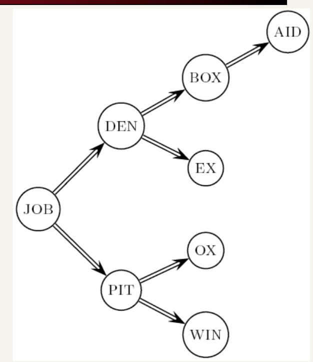
Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		

Dictionary search without blocking

Binary search

Assuming each dictionary term is equally likely in query (not really so in practice!), average number of comparisons = ?

(1 + 2*2+4*3+4)/8 = 2.6



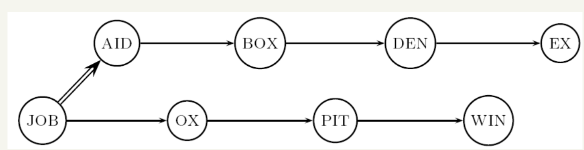
Dictionary search with blocking

What about with blocking?

….*7systile9syzygetic8syzygial6syzygy11szaibelyite8szczecin9szomo*….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		

Dictionary search with blocking



Binary search down to 4-term block

- Then linear search through terms in block.

Blocks of 4 (binary tree), avg. = ?

$(1+2\cdot2+2\cdot3+2\cdot4+5)/8 = 3$ compares

More improvements…

8***automata***8***automate***9***automatic***10***automation***

We're storing the words in sorted order

Any way that we could further compress this block?

Front coding

Front-coding:

Sorted words commonly have long common prefixes – store differences only (for last *k-1* in a block of *k*)

8***automata***8***automate***9***automatic***10***automation***

→8***automat****a1◊e2◊ic3◊ion*

Encodes ***automat***

Extra length beyond ***automat***

Begins to resemble general string compression

RCV1 dictionary compression

Technique	Size in MB
Fixed width	11.2
String with pointers to every term	7.6
Blocking *k* = 4	7.1
Blocking + front coding	5.9

Postings compression

The postings file is much larger than the dictionary, by a factor of at least 10

A posting for our purposes is a docID

Regardless of our postings list data structure, we need to store all of the docIDs

For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers

Alternatively, we can use $\log_2 800{,}000 \approx 20$ bits per docID

Postings: two conflicting forces

Where is most of the storage going?

Frequent terms will occur in most of the documents and require a lot of space

A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.

- Prefer 0/1 bitmap vector in this case

A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2$ 1M ~ 20 bits.

Postings file entry

We store the list of docs containing a term in increasing order of docID.

- ***computer***: $\boxed{33,47}$,154,159,202 …

Is there another way we could store this sorted data?

Store *gaps*: $\boxed{33,14}$,107,5,43 …

- 14 = 47-33
- 107 = 154 – 47
- 5 = 159 - 154

Fixed-width

	encoding	postings list									
THE	docIDs	...		283042		283043		283044		283045	...
	gaps				1		1		1		...
COMPUTER	docIDs	...		283047		283154		283159		283202	...
	gaps				107		5		43		...
ARACHNOCENTRIC	docIDs	252000		500100							
	gaps	252000	248100								

How many bits do we need to encode the gaps?

Does this buy us anything?

Variable length encoding

Aim:

- For ***arachnocentric***, we will use ~20 bits/gap entry
- For ***the***, we will use ~1 bit/gap entry

Key challenge: encode every integer (gap) with as few bits as needed for that integer

1, 5, 5000, 1, 1524723, …

for smaller integers, use fewer bits
for larger integers, use more bits

Variable length coding

1, 5, 5000, 1, 1124 ...

1, 101, 1001110001, 1, 10001100101 ...

Fixed width:

$\underbrace{0000000001}\underbrace{0000000101}\underbrace{1001110001}$ ...

every 10 bits

Variable width:

11011001111000111000110010 1 ...

?

Variable Byte (VB) codes

Rather than use 20 bits, i.e. record gaps with the smallest number of bytes to store the gap

1, 101, 1001110001

$\underbrace{00000001}_{\text{1 byte}}$, $\underbrace{00000101}_{\text{1 byte}}$, $\underbrace{00000010\ 01110001}_{\text{2 bytes}}$

00000001000001010000001001110001

?

VB codes

Reserve the first bit of each byte as the continuation bit

If the bit is 1, then we're at the end of the bytes for the gap

If the bit is 0, there are more bytes to read

1, 101, 1001110001

10000001 10000101 00000100 11110001

For each byte used, how many bits of the gap are we storing?

Example

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Postings stored as the byte concatenation
000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.

Other variable codes

Instead of bytes, we can also use a different "unit of alignment": 32 bits (words), 16 bits, 4 bits (nibbles) etc.

What are the pros/cons of a smaller/larger unit of alignment?

- Larger units waste less space on continuation bits (1 of 32 vs. 1 of 8)
- Smaller unites waste less space on encoding smaller number, e.g. to encode '1' we waste (6 bits vs. 30 bits)

More codes

10000001100001010000000100 11110001

Still seems wasteful

What is the major challenge for these variable length codes?

We need to know the length of the number!

Idea: Encode the length of the number so that we know how many bits to read

Gamma codes

Represent a gap as a pair *length* and *offset*

offset is *G* in binary, with the leading bit cut off

- 13 → 1101 → 101
- 17 → 10001 → 0001
- 50 → 110010 → 10010

length is the length of offset

- 13 (offset 101), it is 3
- 17 (offset 0001), it is 4
- 50 (offset 10010), it is 5

Encoding the length

We've stated *what* the length is, but not *how* to encode it

What is a requirement of our length encoding?

- Lengths will have variable length (e.g. 3, 4, 5 bits)
- We must be able to decode it without any ambiguity

Any ideas?

Unary code

- Encode a number *n* as *n* 1's, followed by a 0, to mark the end of it
- 5 → 111110
- 12 → 1111111111110

Gamma code examples

number	length	offset	γ-code
0			
1			
2			
3			
4			
9			
13			
24			
511			
1025			

Gamma code examples

number	length	offset	γ-code
0			none
1	0		0
2	10	0	10,0
3	10	1	10,1
4	110	00	110,00
9	1110	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001

Gamma code properties

Uniquely prefix-decodable, like VB

All gamma codes have an odd number of bits

What is the fewest number of bits we could expect to express a gap (without any other knowledge of the other gaps)?

- $\log_2$ (gap)

How many bits do gamma codes use?

- $2 \lfloor \log_2 (\text{gap}) \rfloor + 1$ bits
- Almost within a factor of 2 of best possible

Gamma seldom used in practice

Machines have word boundaries – 8, 16, 32 bits

Compressing and manipulating at individual bit-granularity will slow down query processing

Variable byte alignment is potentially more efficient

Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

RCV1 compression

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, k = 4	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ–encoded	101.0

Index compression summary

We can now create an index for highly efficient Boolean retrieval that is very space efficient

Only 4% of the total size of the collection

Only 10-15% of the total size of the text in the collection

However, we've ignored positional information

Hence, space savings are less for indexes used in practice

- But techniques substantially the same