ADFOCS 2004

Prabhakar Raghavan
Lecture 2

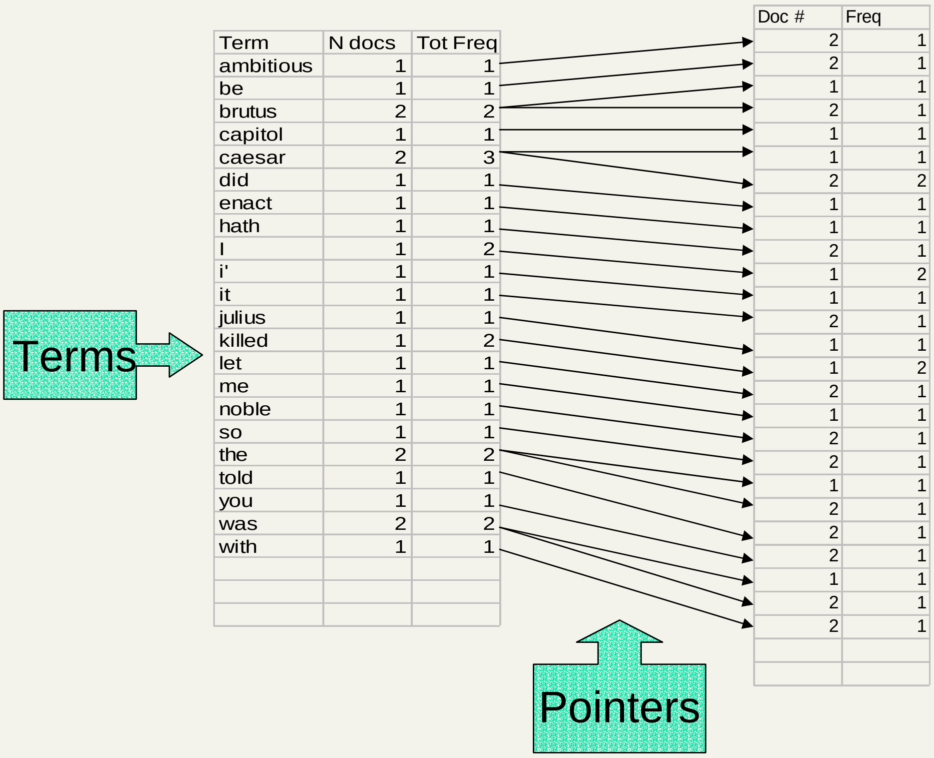
Corpus size for estimates

- Consider n = 1M documents, each with about 1K terms.
- Avg 6 bytes/term incl spaces/punctuation
  - 6GB of data.
- Say there are m = 500K *distinct* terms among these.

Don’t build the matrix

- 500K x 1M matrix has half-a-trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.
- So we devised the inverted index
  - Devised query processing for it
- Where do we pay in storage?

- Where do we pay in storage?



Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Storage analysis

- First will consider space for pointers
- Basic Boolean index only
  - Devise compression schemes
- Then will do the same for dictionary
- No analysis for positional indexes, etc.

Pointers: two conflicting forces

- A term like ***Calpurnia*** occurs in maybe one doc out of a million - would like to store this pointer using $\log_2$ 1M ~ 20 bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/pointer is too expensive.
  - Prefer 0/1 vector in this case.

Postings file entry

- Store list of docs containing a term in increasing order of doc id.
  - ***Brutus***: 33,47,154,159,202 …
- Consequence: suffices to store *gaps*.
  - 33,14,107,5,43 …
- Hope: most gaps encoded with far fewer than 20 bits.

Variable encoding

- For ***Calpurnia***, will use ~20 bits/gap entry.
- For ***the***, will use ~1 bit/gap entry.
- If the average gap for a term is G, want to use ~$\log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with ~ as few bits as needed for that integer.

γ codes for gap encoding

Length	Offset

- Represent a gap G as the pair <*length,offset*>
- *length* is in unary and uses $\lfloor \log_2 G \rfloor + 1$ bits to specify the length of the binary encoding of
- *offset* = $G - 2^{\lfloor \log_2 G \rfloor}$
- e.g., 9 represented as <1110,001>.
- Encoding G takes $2\lfloor \log_2 G \rfloor + 1$ bits.

Exercise

- Given the following sequence of γ–coded gaps, reconstruct the postings sequence:

1110001110101011111101101111011

From these γ–decode and reconstruct gaps, then full postings.

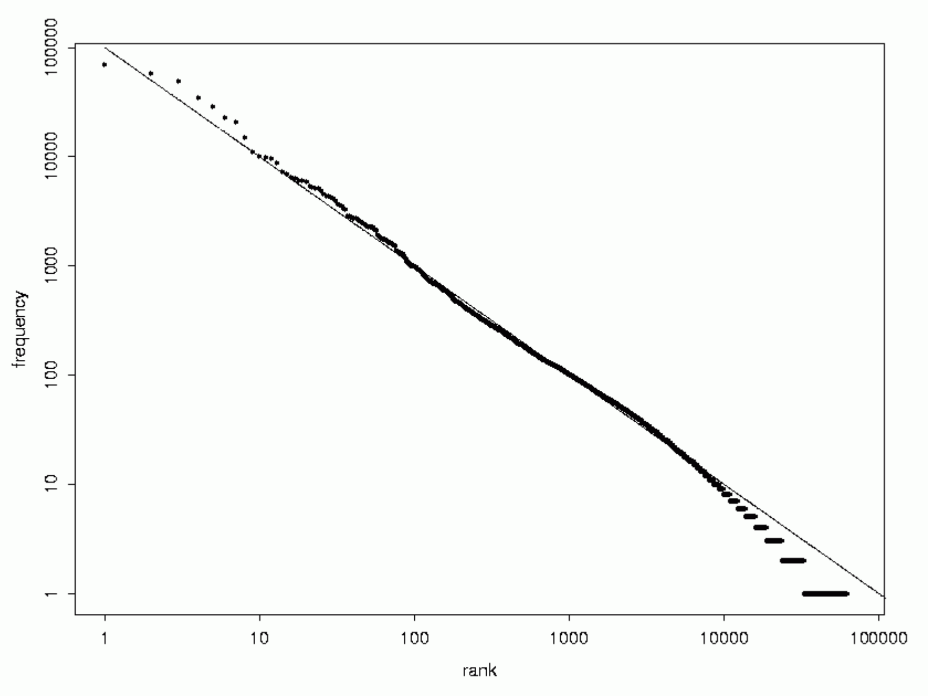
What we've just done

- Encoded each gap as tightly as possible, to within a factor of 2.
- For better tuning (and a simple analysis) - need a handle on the distribution of gap values.

Zipf's law

- The *k*th most frequent term has frequency proportional to *1/k*.
- Use this for a crude analysis of the space used by our postings file pointers.
  - Not yet ready for analysis of dictionary space.

Zipf’s law log-log plot



Rough analysis based on Zipf

- Most frequent term occurs in *n* docs
  - *n* gaps of 1 each.
- Second most frequent term in *n/2* docs
  - *n/2* gaps of 2 each …
- *k*th most frequent term in *n/k* docs
  - *n/k* gaps of *k* each - use $2\log_2 k + 1$ bits for each gap;
  - net of ~$(2n/k).\log_2 k$ bits for *k*th most frequent term.

Sum over k from 1 to m=500K

- Do this by breaking values of k into groups:
  group i consists of $2^{i-1} \leq k < 2^i$.
- Group i has 2^{i-1} components in the sum, each contributing at most $(2ni)/2^{i-1}$.
  - Recall n=1M
- Summing over i from 1 to 19, we get a net estimate of 340Mbits ~45MB for our index.

Work out calculation.

Caveats

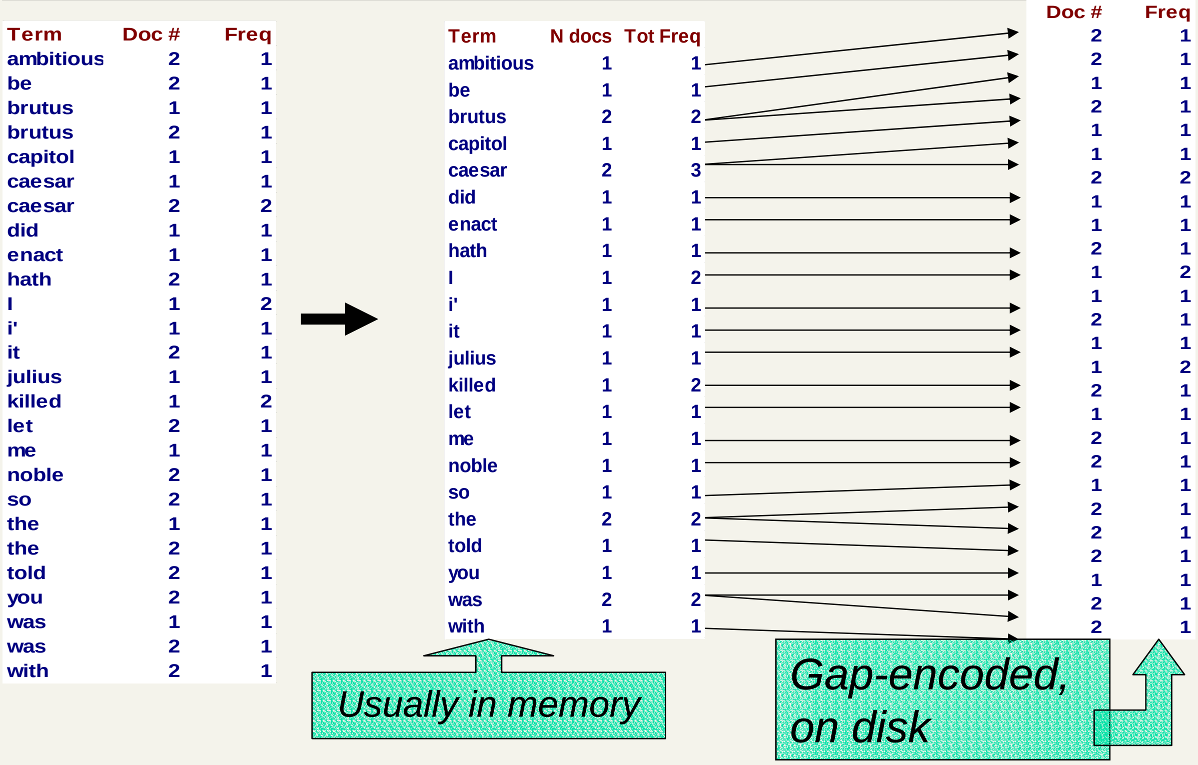
- This is not the entire space for our index:
  - does not account for dictionary storage;
  - as we get further, we'll store even more stuff in the index.
- Assumes Zipf's law applies to occurrence of terms in docs.
- All gaps for a term taken to be the same.
- Does not talk about query processing.

More practical caveat

- γ codes are neat but in reality, machines have word boundaries – 16, 32 bits etc
  - Compressing and manipulating at individual bit-granularity is overkill in practice
  - Slows down architecture
- In practice, simpler word-aligned compression (see Scholer reference) better

---

Dictionary and postings files



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

Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Usually in memory

Gap-encoded, on disk

Inverted index storage

Next up: Dictionary storage

- Dictionary in main memory, postings on disk
  - This is common, especially for something like a search engine where high throughput is essential, but can also store most of it on disk with small, in-memory index
- Tradeoffs between compression and query processing speed
  - Cascaded family of techniques

How big is the lexicon V?

- Grows (but more slowly) with corpus size
- Empirically okay model:

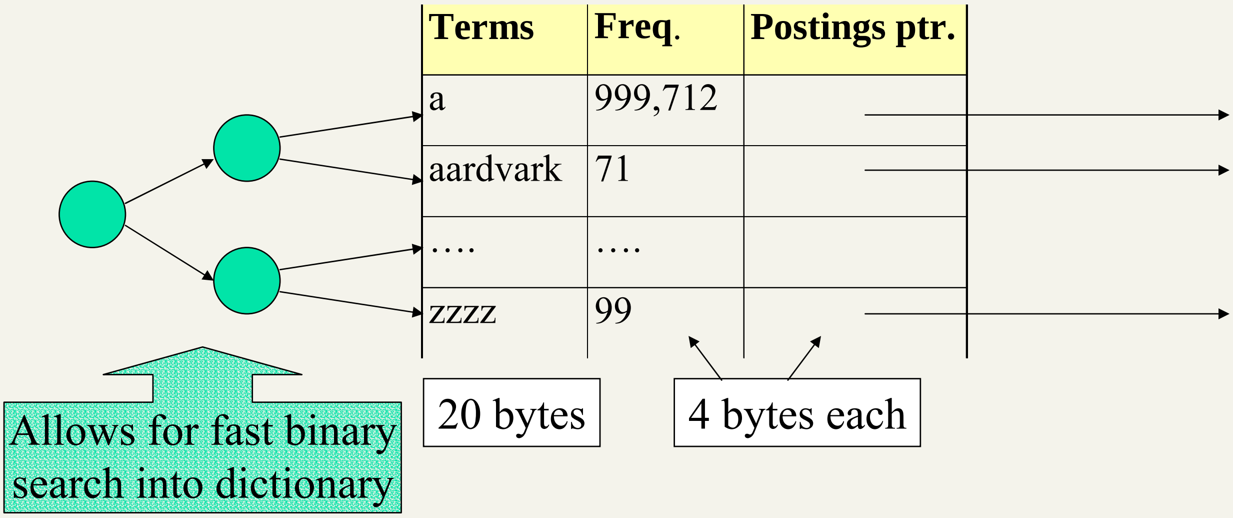
$$m = kN^b$$

Exercise: Can one derive this from Zipf's Law?

- where $b \approx 0.5$, $k \approx 30$–100; N = # tokens
- For instance TREC disks 1 and 2 (2 Gb; 750,000 newswire articles): ~ 500,000 terms
- V is decreased by case-folding, stemming
- Indexing all numbers could make it extremely large (so usually don't*)
- Spelling errors contribute a fair bit of size

Dictionary storage - first cut

- Array of fixed-width entries
  - 500,000 terms; 28 bytes/term = 14MB.



Terms	**Freq.**	**Postings ptr.**
a	999,712	
aardvark	71	
….	….	
zzzz	99	

Allows for fast binary search into dictionary

20 bytes

4 bytes each

Exercises

- Is binary search really a good idea?
- What are the alternatives?

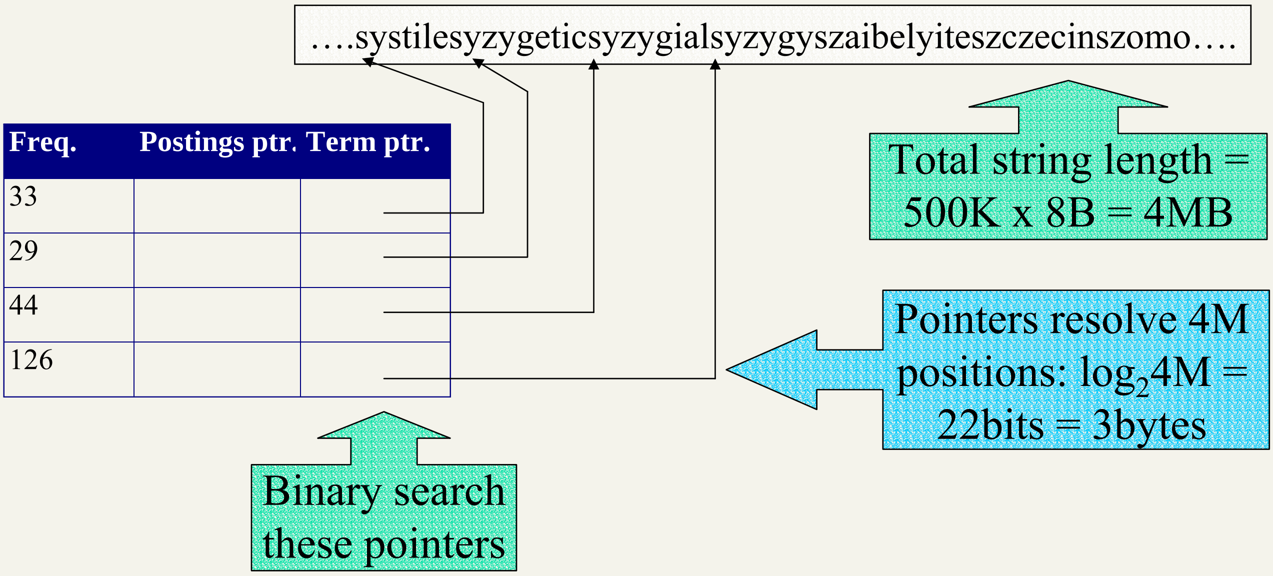
Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
  - And still can't handle *supercalifragilisticexpialidocious.*
- Written English averages ~4.5 characters.
  - Exercise: Why is/isn't this the number to use for estimating the dictionary size?
  - Short words dominate token counts.
- Average word in English: ~8 characters.

Explain this.

Compressing the term list

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space.



….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

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

Total string length = 500K x 8B = 4MB

Pointers resolve 4M positions: $\log_2 4M$ = 22bits = 3bytes

Binary search these pointers

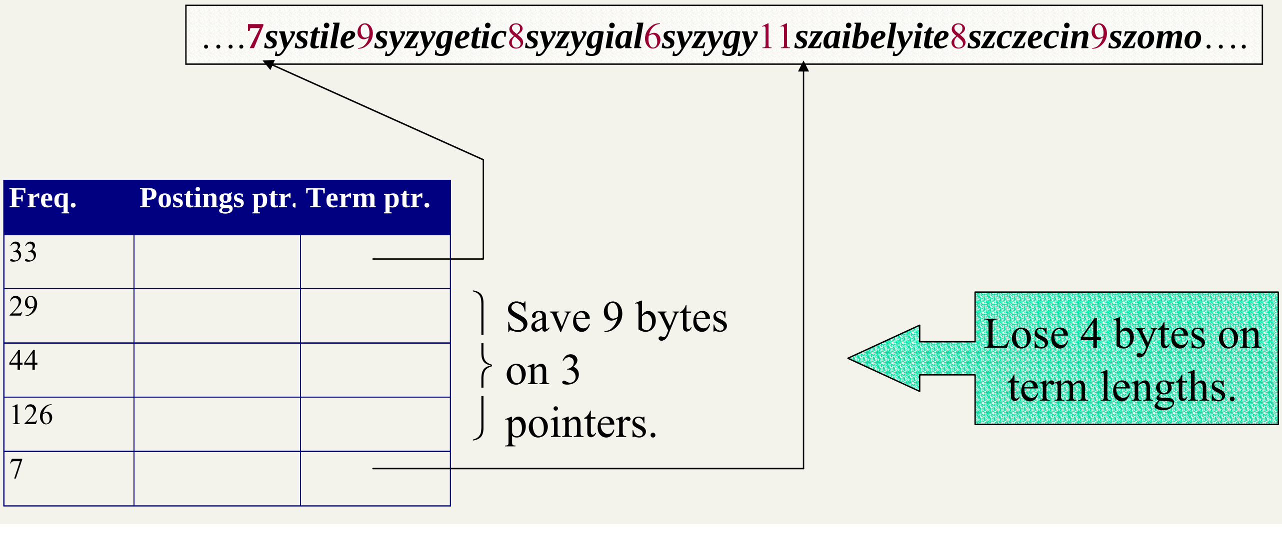
Total space for compressed list

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 500K terms ⇒ 9.5MB

} Now avg. 11 bytes/term, not 20.

Blocking

- Store pointers to every *k*th on term string.
  - Example below: *k*=4.
- Need to store term lengths (1 extra byte)



….7***systile***9***syzygetic***8***syzygial***6***syzygy***11***szaibelyite***8***szczecin***9***szomo***….

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*.

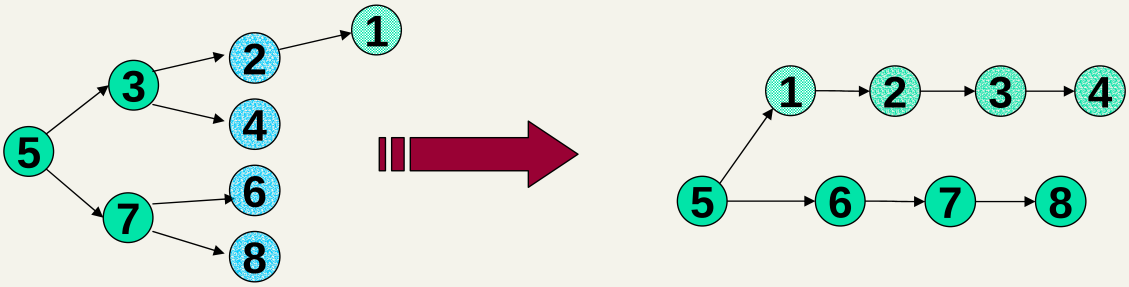
Why not go with larger *k*?

Exercise

- Estimate the space usage (and savings compared to 9.5MB) with blocking, for block sizes of *k* = *4, 8* and *16.*

Impact on search

- Binary search down to 4-term block;
- Then linear search through terms in block.
- 8 documents: binary tree ave. = 2.6 compares
- Blocks of 4 (binary tree), ave. = 3 compares



$= (1+2\cdot 2+4\cdot 3+4)/8$

$=(1+2\cdot 2+2\cdot 3+2\cdot 4+5)/8$

Exercise

- Estimate the impact on search performance (and slowdown compared to *k*=1) with blocking, for block sizes of *k = 4, 8* and *16.*

Total space

- By increasing *k*, we could cut the pointer space in the dictionary, at the expense of search time; space 9.5MB → ~8MB
- Adding in the 45MB for the postings, total 53MB for the simple Boolean inverted index

---

Some complicating factors

- Accented characters
  - Do we want to support accent-sensitive as well as accent-insensitive characters?
  - E.g., query ***resume*** expands to ***resume*** as well as ***résumé***
  - But the query ***résumé*** should be executed as only ***résumé***
  - Alternative, search application specifies
- If we store the accented as well as plain terms in the dictionary string, how can we support both query versions?

Index size

- Stemming/case folding cut
  - number of terms by ~40%
  - number of pointers by 10-20%
  - total space by ~30%
- Stop words
  - Rule of 30: ~30 words account for ~30% of all term occurrences in written text
  - Eliminating 150 commonest terms from indexing will cut almost 25% of space

---

Extreme compression (see *MG*)

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

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

→8{***automat***}***a***1◊***e***2◊***ic***3◊***ion***

Encodes ***automat***

Extra length beyond ***automat.***

Begins to resemble general string compression.

Extreme compression

- Using perfect hashing to store terms “within” their pointers
  - not good for vocabularies that change.
- Partition dictionary into pages
  - use B-tree on first terms of pages
  - pay a disk seek to grab each page
  - if we’re paying 1 disk seek anyway to get the postings, “only” another seek/query term.

Compression: Two alternatives

- Lossless compression: all information is preserved, but we try to encode it compactly
  - What IR people mostly do
- Lossy compression: discard some information
  - Using a stoplist can be thought of in this way
  - Techniques such as Latent Semantic Indexing (TH) can be viewed as lossy compression
  - One could prune from postings entries unlikely to turn up in the top *k* list for query on word
    - Especially applicable to web search with huge numbers of documents but short queries (e.g., Carmel et al. *SIGIR 2002)*

Top *k* lists

- Don’t store all postings entries for each term
    - Only the “best ones”
    - Which ones are the best ones?
- More on this subject later, when we get into ranking

Index construction

Index construction

- Thus far, considered index space
- What about index construction time?
- What strategies can we use with limited main memory?

Somewhat bigger corpus

- Number of docs = n = 40M
- Number of terms = m = 1M
- Use Zipf to estimate number of postings entries:
- $n + n/2 + n/3 + \ldots. + n/m \sim n \ln m$ = 560M entries
- No positional info yet

Check for yourself

Recall index construction

- Documents are parsed to extract words and these are saved with the Document ID.

Doc 1

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

Doc 2

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

Key step

- After all documents have been parsed the inverted file is sorted by terms

We focus on this sort step.

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

Index construction

- As we build up the index, cannot exploit compression tricks
  - parse docs one at a time. The final postings entry for any term is incomplete until the end.
  - (actually you can exploit compression, but this becomes a lot more complex)
- At 10-12 bytes per postings entry, demands several temporary gigabytes

---

System parameters for design

- Disk seek ~ 1 millisecond
- Block transfer from disk ~ 1 microsecond per byte (*following a seek*)
- All other ops ~ 10 microseconds
  - E.g., compare two postings entries and decide their merge order

Bottleneck

- Parse and build postings entries one doc at a time
- Now sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow

If every comparison took 1 disk seek, and *n* items could be sorted with $n\log_2 n$ comparisons, how long would this take?

Sorting with fewer disk seeks

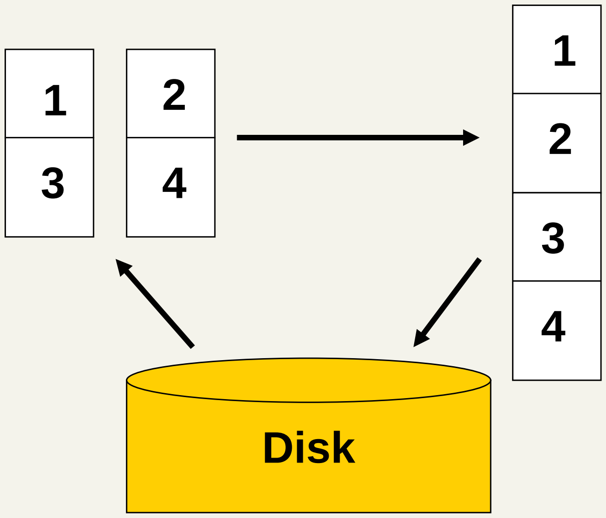
- 12-byte (4+4+4) records *(term, doc, freq).*
- These are generated as we parse docs.
- Must now sort 560M such 12-byte records by *term*.
- Define a <u>Block</u> = 10M such records
  - can "easily" fit a couple into memory.
- Will sort within blocks first, then merge the blocks into one long sorted order.

Sorting 56 blocks of 10M records

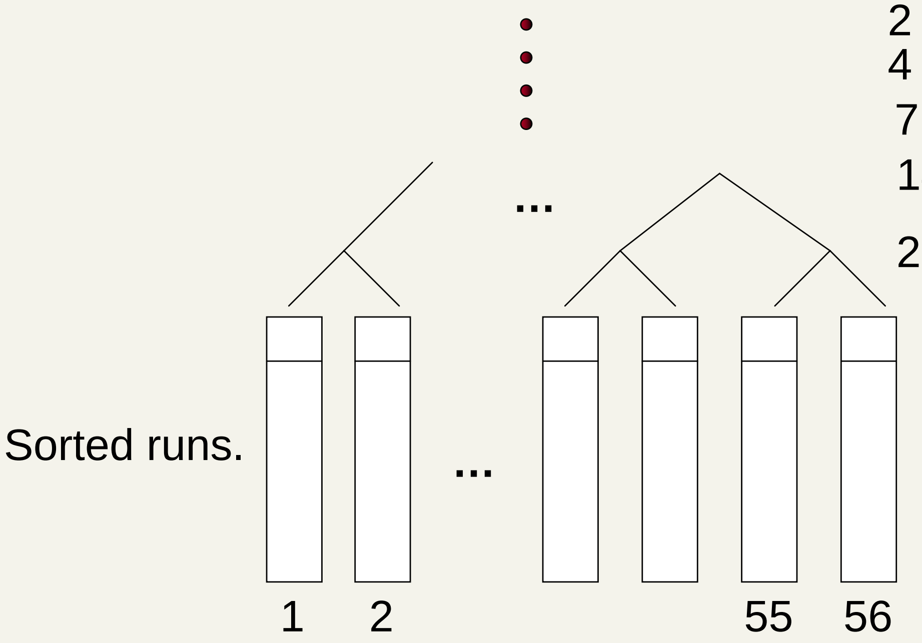
- First, read each block and sort within:
  - Quicksort takes about 2 x (10M ln 10M) steps
- *Exercise: estimate total time to read each block from disk and and quicksort it.*
- 56 times this estimate - gives us 56 sorted *runs* of 10M records each.
- Need 2 copies of data on disk, throughout.

Merging 56 sorted runs

- Merge tree of $\log_2 56$ ~ 6 layers.
- During each layer, read into memory runs in blocks of 10M, merge, write back.



Merge tree



1 runs … ?
2 runs … ?
4 runs … ?
7 runs, 80M/run
14 runs, 40M/run
28 runs, 20M/run

Merging 56 runs

- Time estimate for disk transfer:
- 6 x (56runs x 120MB x 10^{-6}sec) x 2 ~ 22hrs.

Disk block transfer time. Why is this an Overestimate?

Work out how these transfers are staged, and the total time for *merging*.

Exercise - fill in this table

	Step	Time
1	56 initial quicksorts of 10M records each	
2	Read 2 sorted blocks for merging, write back	
3	Merge 2 sorted blocks	
4	Add (2) + (3) = time to read/merge/write	
5	56 times (4) = total merge time	

?

Large memory indexing

- Suppose instead that we had 16GB of memory for the above indexing task.
- *Exercise: how much time to index?*
- *Repeat with a couple of values of n, m.*
- In practice, spidering interlaced with indexing.
  - Spidering bottlenecked by WAN speed and many other factors - more on this later.

Improvements on basic merge

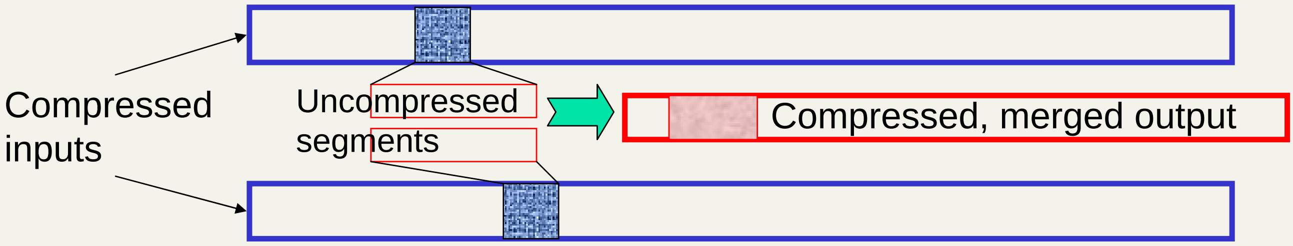
- Compressed temporary files
    - compress terms in temporary dictionary runs
- How do we merge compressed runs to generate a compressed run?
    - Given two γ-encoded runs, merge them into a new γ-encoded run
    - To do this, first γ-decode a run into a sequence of gaps, then actual records:
    - 33,14,107,5… → 33, 47, 154, 159
    - 13,12,109,5… → 13, 25, 134, 139

Merging compressed runs

- Now merge:
    - 13, 25, 33, 47, 134, 139, 154, 159
- Now generate new gap sequence
    - 13,12,8,14,87,5,15,5
- Finish by γ-encoding the gap sequence
- But what was the point of all this?
    - If we were to uncompress the entire run in memory, we save no memory
    - How do we gain anything?

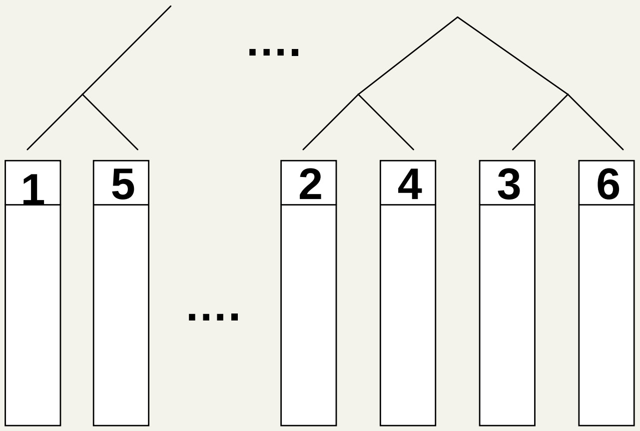
“Zipper” uncompress/decompress

- When merging two runs, bring their γ-encoded versions into memory
- Do NOT uncompress the entire gap sequence at once – only a small segment at a time
  - Merge the uncompressed segments
  - Compress merged segments again



Improving on binary merge tree

- Merge more than 2 runs at a time
  - Merge *k>2* runs at a time for a shallower tree
  - maintain heap of candidates from each run



Dynamic indexing

- Docs come in over time
  - postings updates for terms already in dictionary
  - new terms added to dictionary
- Docs get deleted

Simplest approach

- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index

Resources

- MG 3.3, 3.4, 4.2, 5
- F. Scholer, H.E. Williams and J. Zobel. Compression of Inverted Indexes For Fast Query Evaluation. Proc. ACM-SIGIR 2002.
- http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top