

Text Indexing

CS635

Soumen Chakrabarti

(+ many slides from MRS book)

Abstract word and document model

- Define a **word** as any non-empty maximal sequence of characters from a restricted set
 - E.g. [a-zA-Z0-9]
 - Some languages do not have easy delimiters
- Set of all words found over all documents in corpus is the corpus **vocabulary**
- Can arbitrarily order words and number them
- Henceforth, word integer word ID
- First cut: document = **set** of word IDs
- Later, **bag** (multiset), finally, **sequence**

Word and document IDs

- For readability we will use string words instead of IDs in these slides
- Not essential to assign IDs to words in sorted order, can assign IDs using a counter as we encounter new words
 - Maintain a map (“dictionary”) from known words to allocated IDs
 - Later, will see how to compress this map
- Document IDs are completely arbitrary
 - Possible to assign doc IDs for better index compression

Toy corpus with two documents

d_1 my care is loss of care with old care done

d_2 your care is gain of care with new care won

Corpus

$d_1 = \{6, 1, 4, 5, 8, 1, 10, 9, 1, 2\}$

$d_2 = \{12, 1, 4, 3, 8, 1, 10, 7, 1, 11\}$

Document representation as sequence

$d_1 = \{1, 2, 4, 5, 6, 8, 9, 10\}$

$d_2 = \{1, 3, 4, 7, 8, 10, 11, 12\}$

Document representation as set

Vocabulary

1	care
2	done
3	gain
4	is
5	loss
6	my
7	new
8	of
9	old
10	with
11	won
12	your

Toy corpus as binary matrix

w_i	1	2	3	4	5	6	7	8	9	10	11	12
d_1	1	1	0	1	1	1	0	1	1	1	0	0
d_2	1	0	1	1	0	0	1	1	0	1	1	1

- Very sparse, most entries zero
 - 10^9 Web pages, each has 100 distinct words
 - Corpus vocabulary may be as large as 10^6
- When reading corpus, docs arrive one by one
- I.e., matrix is revealed a **row** at a time
- To run Boolean query, must probe by **columns**
- Must **transpose** matrix for fast query processing

Incidence vectors and Boolean queries

- Each term maps to a 0/1 vector
 - 1=care \rightarrow 11; 2=done \rightarrow 10; 7=new \rightarrow 01
- Examples with set representation:
 - Document/s containing “care” and “done”
 - $11 \wedge 10 = 10 \Rightarrow$ i.e. document 1
 - Document/s containing “care” but not “new”
 - $11 \wedge \neg 01 = 11 \wedge 10 = 10$ i.e. document 1
- Examples with sequence representation:
 - Document containing phrase “new care”
 - Need to keep track of positions where terms appeared
- Can compose into more complex queries
 - Has phrase “care with” but not “old”
- Widely used in legal and library search for decades

Inverted index

- For each term t , we must store a list of all documents that contain t .
 - Identify each by a **docID**, a document serial number
- Can we use fixed-size arrays for this?

Brutus

1	2	4	1	3	4	17	174
---	---	---	---	---	---	----	-----

Caesar

1	2	4	¹ 5	¹ 6	⁵ 1	³ 5	13
---	---	---	-------------------	-------------------	-------------------	-------------------	----

Calpurnia

2	31	54	101		⁶	⁷	²
---	----	----	-----	--	--------------	--------------	--------------

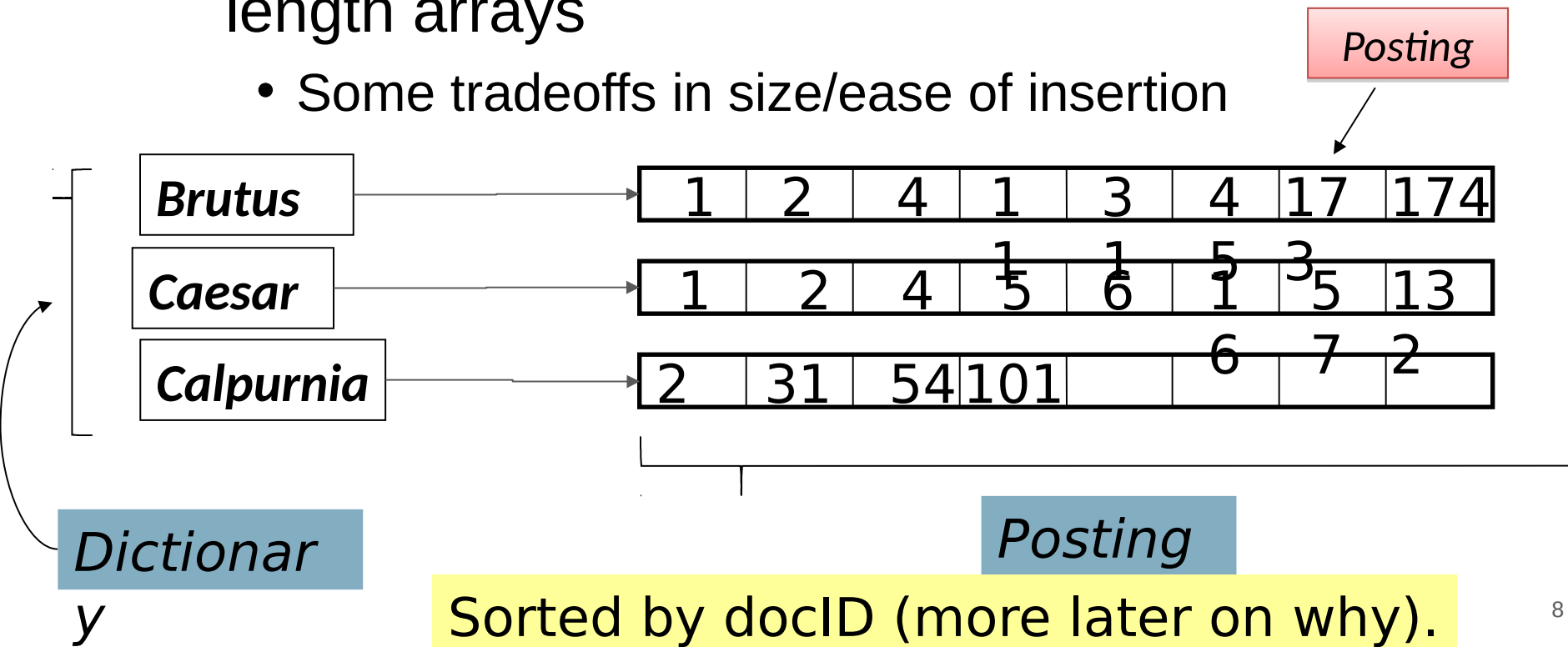
What happens if the word
Caesar is added to document
14?



Inverted index



- We need variable-size postings lists
 - On disk, a continuous run of postings is normal and best
 - In memory, can use linked lists or variable length arrays
 - Some tradeoffs in size/ease of insertion



Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

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	docID
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
	9

Indexer steps: Sort

- Sort by terms
- And then docID
- (Core indexing step)

Term	docID
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
was	2
ambitious	2



Term	docID
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

Dictionary and postings

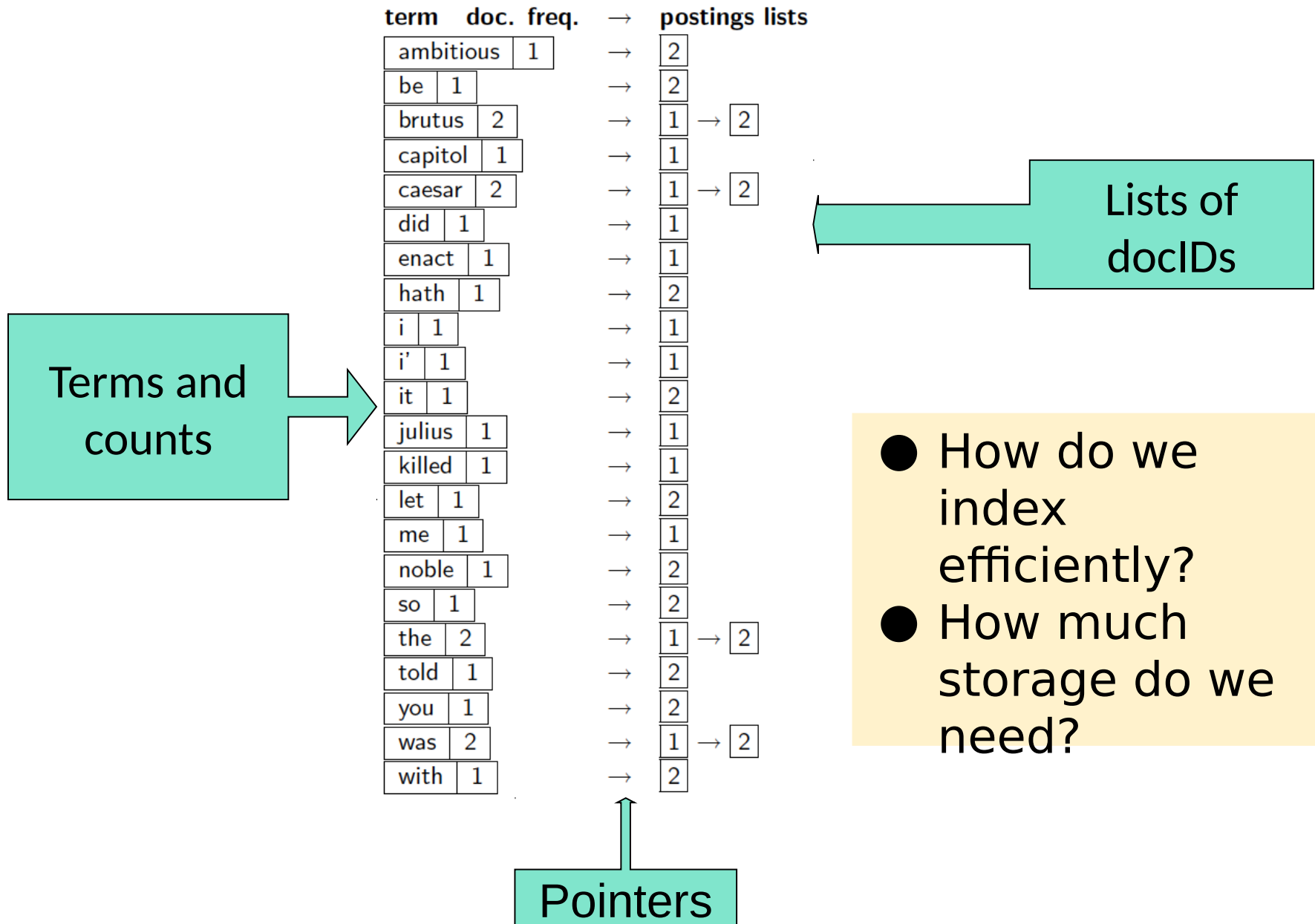
- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Doc. frequency information is added.

Why frequency?
Will discuss later.

Term	docID
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

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Where do we pay in storage?

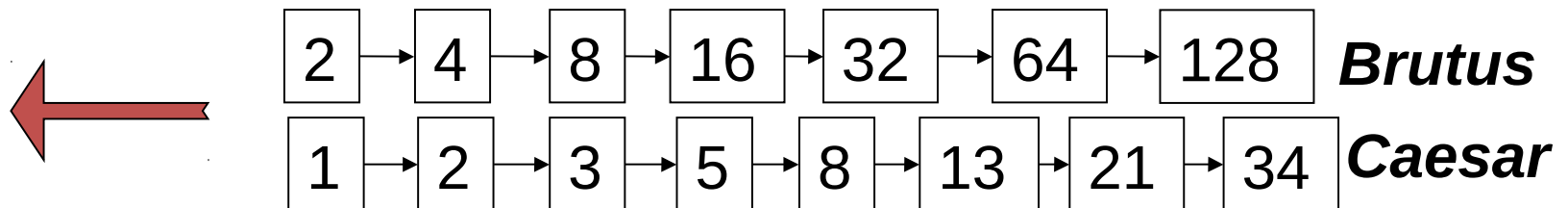


Query processing: AND

■ Consider processing the query:

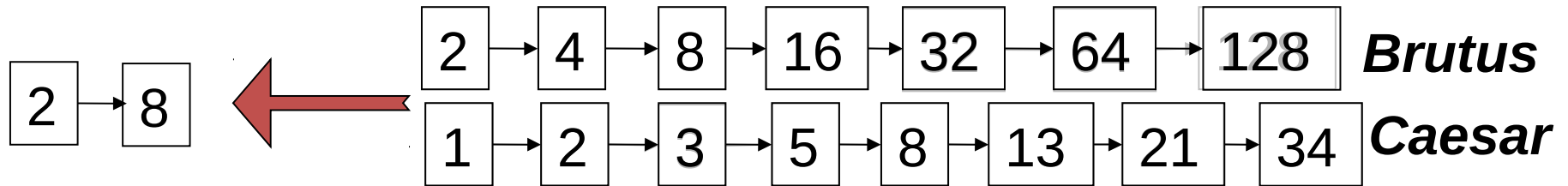
Brutus AND Caesar

- Locate ***Brutus*** in the Dictionary;
 - Retrieve its postings.
- Locate ***Caesar*** in the Dictionary;
 - Retrieve its postings.
- “Merge” the two postings:



The merge

Walk through the two postings simultaneously, in time linear in the total number of postings entries



If list lengths are x and y , merge takes $O(x+y)$ operations.

Crucial: postings sorted by docID.

Intersecting two postings lists (a “merge” algorithm)

INTERSECT(p_1, p_2)

```
1  answer  $\leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4      then ADD(answer,  $\text{docID}(p_1)$ )
5           $p_1 \leftarrow \text{next}(p_1)$ 
6           $p_2 \leftarrow \text{next}(p_2)$ 
7      else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
8          then  $p_1 \leftarrow \text{next}(p_1)$ 
9          else  $p_2 \leftarrow \text{next}(p_2)$ 
10 return answer
```

Boolean queries: Exact match

- The **Boolean retrieval model** is being able to ask a query that is a Boolean expression:
 - Boolean Queries use *AND*, *OR* and *NOT* to join query terms
 - Views each document as a set of words
 - Is precise: document matches condition or not.
 - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades.
- Many search systems you still use are Boolean:
 - Email, library catalog, Mac OS X Spotlight

Example: WestLaw <http://www.westlaw.com/>

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- Tens of terabytes of data; 700,000 users
- Majority of users *still* use boolean queries
- Example query:
 - What is the statute of limitations in cases involving the federal tort claims act?
 - LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM
 - /3 = within 3 words, /S = in same sentence

Boolean queries: More general merges

Exercise: Adapt the merge for the queries:

Brutus AND NOT Caesar

Brutus OR NOT Caesar

Can we still run through the merge in time $O(x+y)$?

What can we achieve?

Merging

What about an arbitrary Boolean formula?

***(Brutus OR Caesar) AND NOT
(Antony OR Cleopatra)***

- Can we always merge in “linear” time?
 - Linear in what?
- Can we do better?

Query optimization

- What is the best order for query processing?
- Consider a query that is an *AND* of n terms.
- For each of the n terms, get its postings, then *AND* them together.

Brutus	2	4	8	1	3	6	12	
Caesar	1	2	3	5	8	1	2	3
Calpurnia	13	16				6	1	4

Query: **Brutus AND Calpurnia AND Caesar**

Query optimization example

- Process in order of increasing freq:
 - *start with smallest set, then keep cutting further.*

This is why we kept
document freq. in
dictionary

Brutus	2	4	8	1	3	6	12	
Caesar	1	2	3	6	2	4	8	
Calpurnia	13	16				6	1	4

Execute the query as (***Calpurnia AND Brutus***) AND ***Caesar***.

More general optimization

- e.g., (*madding OR crowd*) AND (*ignoble OR strife*)
- Get doc. freq.'s for all terms.
- Estimate the size of each *OR* by the sum of its doc. freq.'s (conservative).
- Process in increasing order of *OR* sizes.

Exercise



- Recommend a query processing order for

*(tangerine OR trees) AND
(marmalade OR skies) AND
(kaleidoscope OR eyes)*

Term	Freq
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

Transposing matrix = indexing

w	1	2	3	4	5	6	7	8	9	10	11	12
d_1	1	1	0	1	1	1	0	1	1	1	0	0
d_2	1	0	1	1	0	0	1	1	0	1	1	1

- When reading corpus, read a **row** at a time
- To run Boolean query, must probe by **columns**
- Must **transpose** matrix for fast query processing
- Matrix too large to fit in RAM

Indexing — Hardware basics

- Access to data in memory is ***much*** faster than access to data on disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
- Block sizes: 8KB to 256 KB.

Hardware basics

- Servers used in IR systems now typically have tens to hundreds of GB of main memory
- Available disk space is several (2–3) orders of magnitude larger, tens of terabytes on a typical server
- Hardware fault tolerance is very expensive: It's much cheaper to use many regular machines rather than one fault tolerant machine.

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

Scaling index construction

- In-memory index construction does not scale
 - Can't fit entire collection into memory, sort, then write back
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about . . .
- Memory, disk, speed, etc.

Sort-based index construction

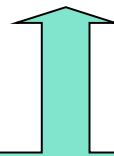
- As we build the index, we parse docs one at a time
- Final postings for any term incomplete until end
- At 12 bytes per non-positional postings entry (*term, doc, freq*), demands a lot of space for large collections (32-bit docids may not suffice)
- Say vocab size is 100,000,000
 - ... can do this in memory in 2009, but typical collections are much larger. E.g., the *New York Times* provides an index of >150 years of newswire
- Must store intermediate results on disk

Sort using disk as “memory”?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: 100,000,000 records on disk is too slow – too many disk seeks.
- We need an *external* sorting algorithm.

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 – must sort 100M records



If every comparison took 2 disk seeks, and N items could be sorted with $N \log_2 N$ comparisons, how long would this take?

BSBI: Blocked sort-based Indexing (Sorting with fewer disk seeks)

- 12-byte (4+4+4) records (term, doc, freq)
- These are generated as we parse docs
- Must now sort 100M such 12-byte records by term
- Define a *block* ~10M such records
 - Can easily fit a couple blocks into memory
 - Will have 10 such blocks to start with
- Basic idea of algorithm:
 - Accumulate postings for each block, sort, write to disk
 - Then merge the blocks into one long sorted order

postings
to be merged

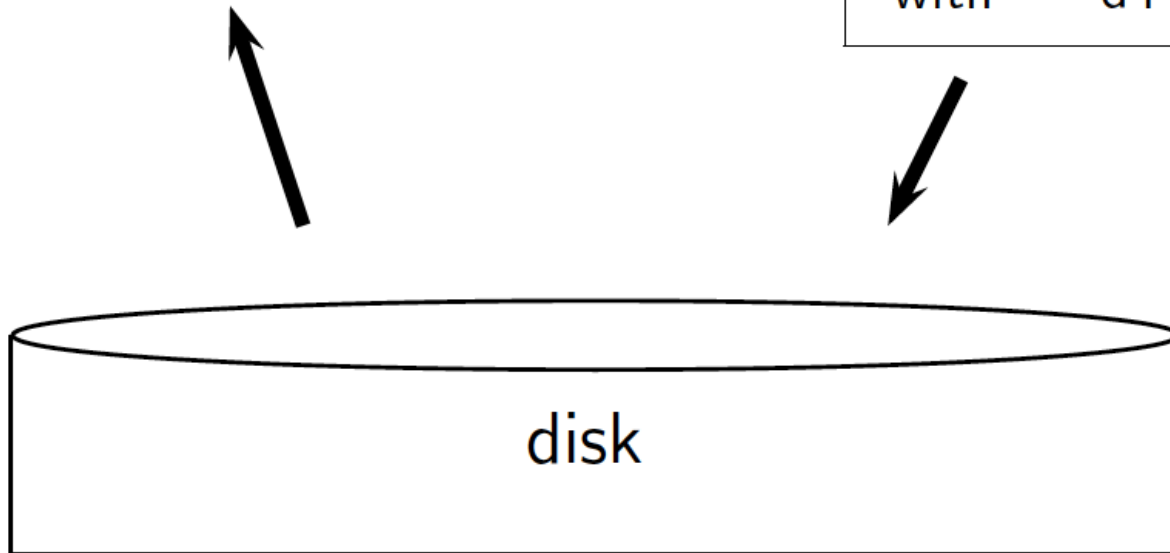
brutus	d3
caesar	d4
noble	d3
with	d4

brutus	d2
caesar	d1
julius	d1
killed	d2



brutus	d2
brutus	d3
caesar	d1
caesar	d4
julius	d1
killed	d2
noble	d3
with	d4

merged
postings



Sorting 10 blocks of 10M records

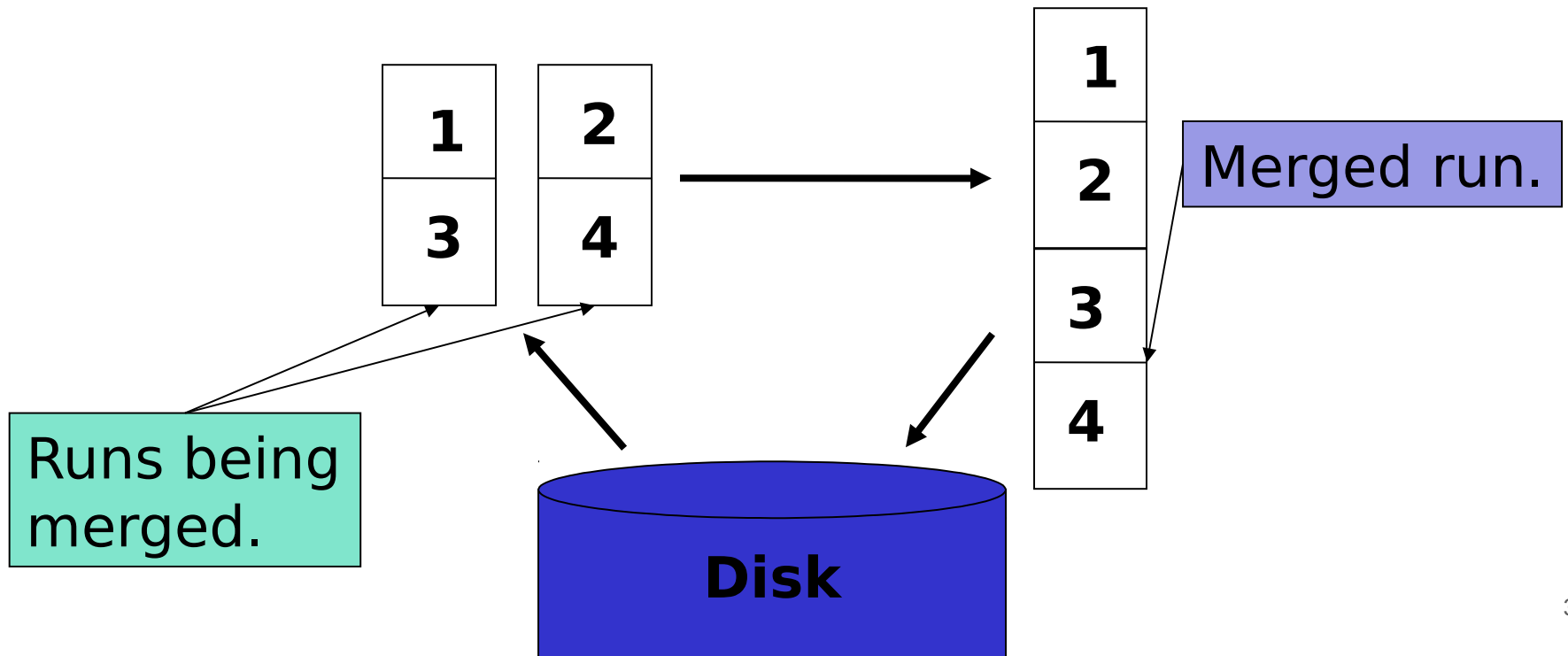
- First, read each block and sort within:
 - Quicksort takes $2N \ln N$ expected steps
 - In our case $2 \times (10M \ln 10M)$ steps
- *Exercise: estimate total time to read each block from disk and and quicksort it.*
- 10 times this estimate – gives us 10 sorted runs of 10M records each.
- Done straightforwardly, need 2 copies of data on disk
 - But can optimize this

BSBINDEXCONSTRUCTION()

```
1   $n \leftarrow 0$ 
2  while (all documents have not been processed)
3  do  $n \leftarrow n + 1$ 
4       $block \leftarrow \text{PARSENEXTBLOCK}()$ 
5       $\text{BSBI-INVERT}(block)$ 
6       $\text{WRITEBLOCKTODISK}(block, f_n)$ 
7   $\text{MERGEBLOCKS}(f_1, \dots, f_n; f_{\text{merged}})$ 
```

How to merge the sorted runs?

- Can do binary merges, with a merge tree of $\log_2 10 = 4$ layers.
- During each layer, read into memory runs in blocks of 10M, merge, write back.
- Generalize to multiway merges for i/o efficiency

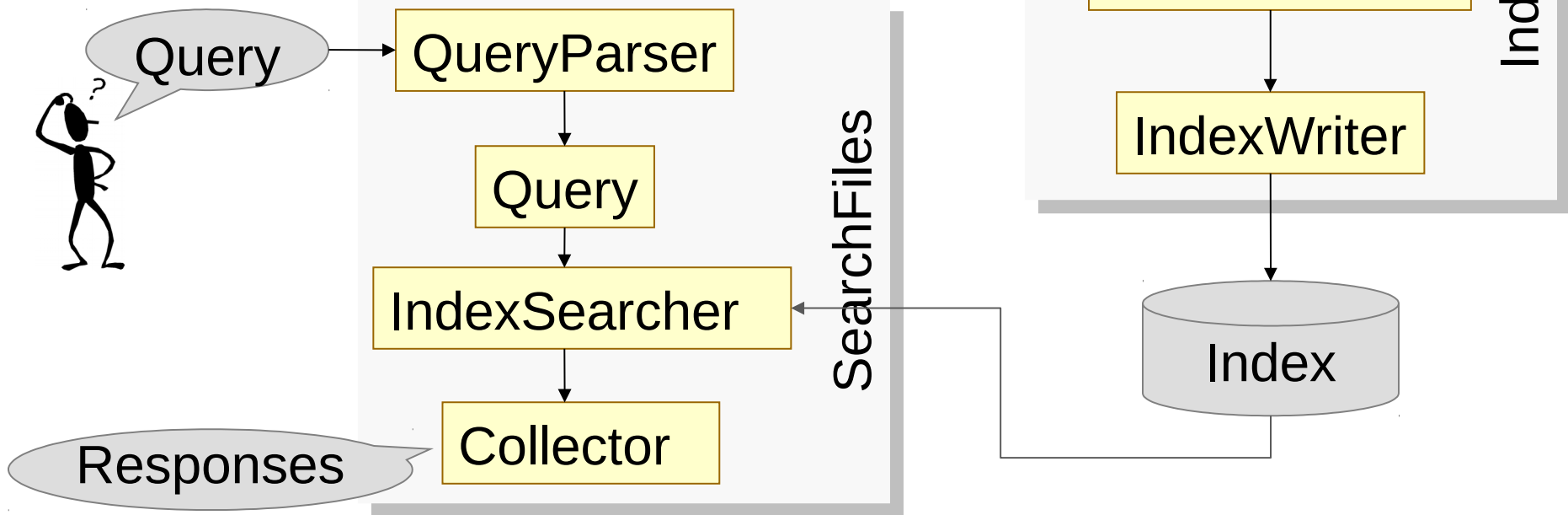


A few practical issues

- Two things in each index 'shard'
 - Map from string tokens to (begin, end) of posting lists
 - Posting lists themselves
- Vocabulary is different in different shards because terms assigned IDs as they are encountered in respecting corpus partitions
- Index merge needs to take extra care about vocabulary/dictionary merging

Lucene, hands-on

- `java -cp $CLASSPATH org.apache.lucene.demo.IndexFiles -index /path/to/index/dir /path/to/files/to/index`
- `java org.apache.lucene.demo.SearchFiles`



SPIMI:

Single-pass in-memory indexing

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don't sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

SPIMI-Invert

```
SPIMI-INVERT(token_stream)
1  output_file = NEWFILE()
2  dictionary = NEWHASH()
3  while (free memory available)
4  do token  $\leftarrow$  next(token_stream)
5      if term(token)  $\notin$  dictionary
6          then postings_list = ADDTODICTIONARY(dictionary, term(token))
7          else postings_list = GETPOSTINGSLIST(dictionary, term(token))
8      if full(postings_list)
9          then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11 sorted_terms  $\leftarrow$  SORTTERMS(dictionary)
12 WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13 return output_file
```

■ Merging of blocks is analogous to BSBI.

Recap

- Document as set, bag, sequence of terms
- Corpus as term-document matrix
- Indexing = transposing $(d,t) \Rightarrow (t,d)$
- Term \Rightarrow posting list (= sparse list of doc IDs)
- Matrix too large to store in RAM
- Cast as external merge sort
 - Process blocks of docs in RAM
 - Merge per-block indices
- Compress the posting list and dictionary

Lucene and MG4J demo → exercises

PyLucene
available!

- Download [code](#), build and run. Print document scores.
- Using synthetic docs and queries, tease out salient properties of the scoring functions.
- Run MG4J and try query *network embedding* and note result highlighting.
- Extend to multiple fields. Learn multi-field query format.
- You are given a dict mapping a term to a set of synonyms. Modify the text scanner to emit all synonyms at the same token offset as the original word.
- You are given a dict from terms (cat, dog, hippo) to hypernyms (mammal). Modify the text scanner to emit all hypernyms at the same token offset as the original word.
- Test above two mods with synthetic corpus and queries.

Deeper dive if you wish

- (Do not use Lucene, SOLR, etc.)
- Familiarize with a few key classes of MG4J (InputBitStream, OutputBitStream)
- Some classes we can provide (ExternalMergeSort)
- Write a very basic indexing program
 - What is the size of the index?
- Extend to distributed map-reduce version?
- And a basic Boolean query processor

Positional queries

- Examples with set representation:
 - Document/s containing “care” and “done”
 - Document/s containing “care” but not “old”
- Examples with sequence representation:
 - Document containing phrase “new care”
 - ... “care” within 4 words of “won”
- Can build more complex clauses
 - Has phrase “care with” but not “old”
- Was state of the art in library catalog and legal search (small corpus) for a long time

Reusing the index we built

- Relax a phrase or proximity query to AND
- “new york” \square “new” AND “york”
- Not all docs that pass the AND filter will have the phrase
- To filter, must read the document
- Random seek, very slow
- Solution: in the posting list, retain not only the doc ID, but also the word offset where the word occurred

Toy corpus with two documents

d_1 my₀ care₁ is₂ loss₃ of₄ care₅ with₆ old₇ care₈ done₉

d_2 your₀ care₁ is₂ gain₃of₄ care₅ with₆ new₇ care₈ won₉

Documents as word sequences

Vocabulary	1	care	→	$d_1:1,5,8; d_2:1,5,8$
	2	done		
	3	gain	→	$d_1: 9$
	4	is		
	5	loss		
	6	my		
	7	new		
	8	of		
	9	old		
	10	with		
	11	won		
	12	your		

Positional posting list: a doc info block followed by position list

- Positional postings can speed up query processing
- But consumes much more index space
- Can (mostly) reconstruct document (except for discarded bits like case, punct)

Why compression (in general)?

- Use less disk space
 - Saves a little money
- 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
 - True of the decompression algorithms we use

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.
 - Compression lets you keep more in memory

■ We will devise various IR-specific compression schemes

Index compression

BRUTUS →

1	2	4	11	31	45	173	174
---	---	---	----	----	----	-----	-----

CAESAR →

1	2	4	5	6	16	57	132	...
---	---	---	---	---	----	----	-----	-----

CALPURNIA →

2	31	54	101
---	----	----	-----

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID (for starters)
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
 - Web corpus → 100 G ID space
- Alternatively, we can use $\log_2 800,000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

Postings: two conflicting forces

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

Postings file entry

- We store the list of docs containing a term in increasing order of docID.
 - **computer**: 33,47,154,159,202 ...
- Consequence: it suffices to store *gaps*.
 - 33,14,107,5,43 ...
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.
- Heads we win, tails they lose
 - Either a word is rare, few gaps to encode
 - Or many small gaps, each needs few bits

Three postings entries

	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				

Variable length encoding

■ Aim:

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

■ If the average gap for a term is G , we want to use $\sim \log_2 G$ bits/gap entry.

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

■ This requires a *variable length encoding*

■ Variable length codes achieve this by using short codes for small numbers

Variable Byte (VB) codes

- For a gap value G , we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store G and dedicate 1 bit in it to be a continuation bit c
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode G 's lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for the other bytes $c = 0$.

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 unit codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles).
- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.
- Variable byte codes:
 - Used by many commercial/research systems
 - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches (vs. bit-level codes, which we look at next).
- There is also recent work on word-aligned codes that pack a variable number of gaps into one word
- Be sure to view Jeff Dean talk at WSDM 2009

Unary code

- Represent n as n 1s with a final 0.

- Unary code for 3 is 1110.

- Unary code for 40 is

110

- Unary code for 80 is:

111

110

- This doesn't look promising, but....

Gamma codes

- We can compress better with bit-level codes
 - The Gamma code is the best known of these.
- Represent a gap G as a pair *length* and *offset*
- *offset* is G in binary, with the leading bit cut off
 - For example $13 \rightarrow 1101 \rightarrow 101$
- *length* is the length of offset
 - For 13 (offset 101), this is 3.
- We encode *length* with *unary code*: 1110.
- Gamma code of 13 is the concatenation of *length* and *offset*: 1110101

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

- G is encoded using $2 \lfloor \log G \rfloor + 1$ bits
 - Length of offset is $\lfloor \log G \rfloor$ bits
 - Length of length is $\lfloor \log G \rfloor + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$
- Gamma code is uniquely prefix-decodable, like VB
- Can be used for any distribution, even if not the best fit
- Parameter-free

Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32, 64 bits
 - Operations that cross word boundaries are slower
- Compressing and manipulating at the granularity of bits can be slow
- Variable byte encoding is aligned and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

RCV1 posting 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

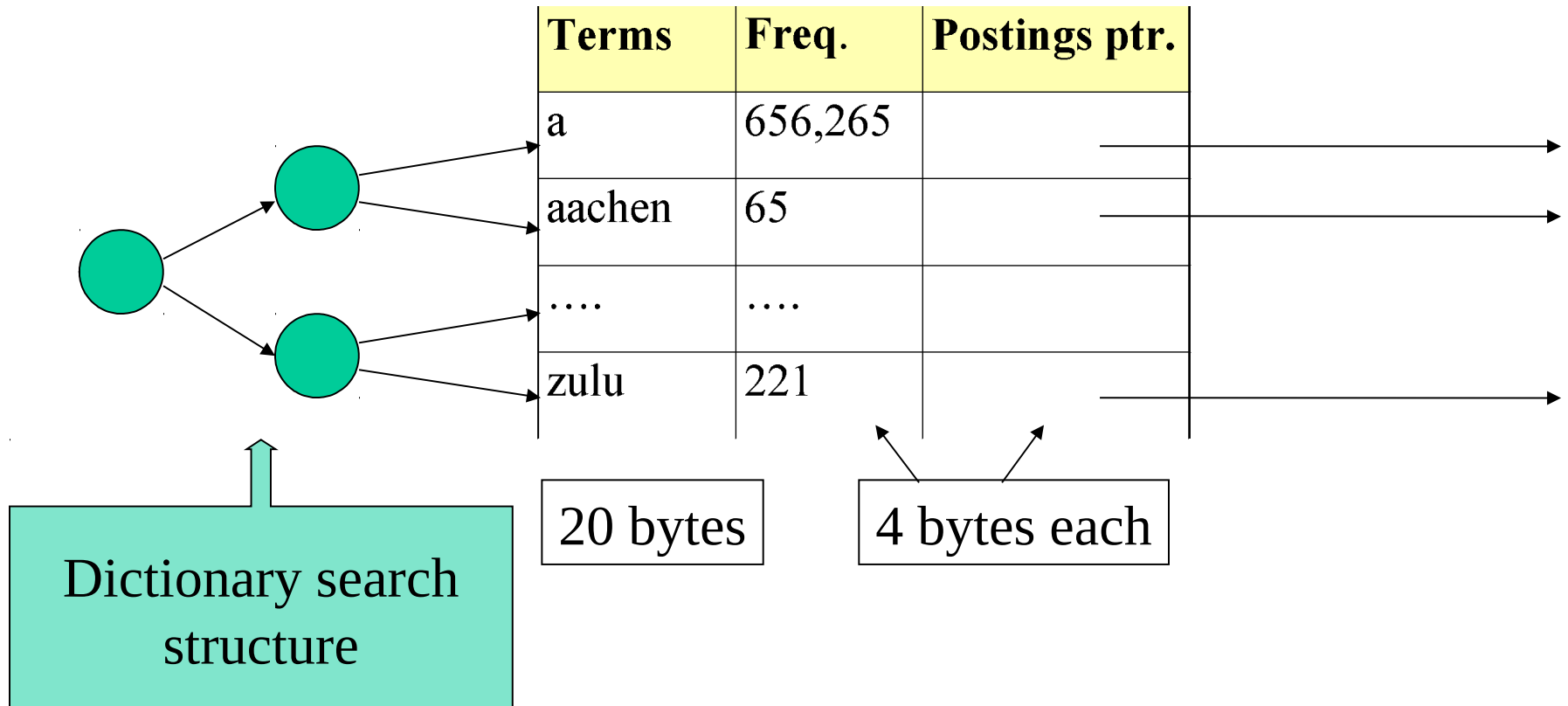
Next: dictionary compression

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

Dictionary storage - first cut

■ Array of fixed-width entries

- ~400,000 terms; 28 bytes/term = 11.2 MB.

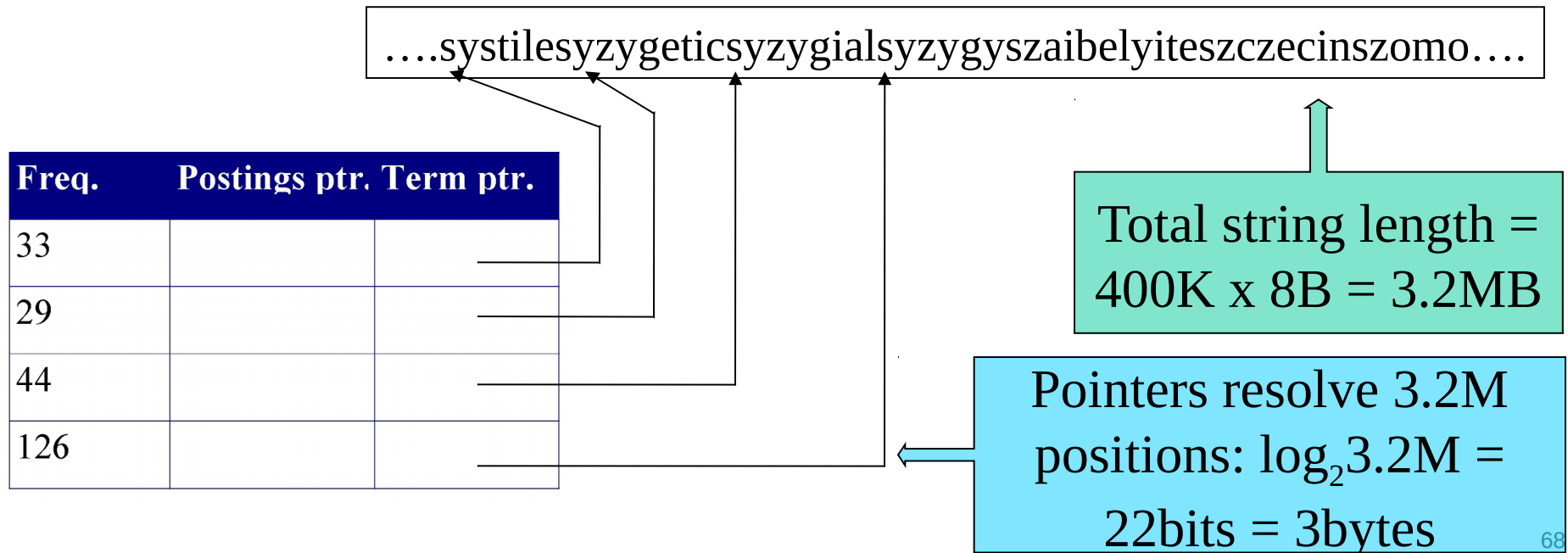


Fixed-width terms are wasteful

- 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* or *hydrochlorofluorocarbons*.
- Written English averages ~4.5 chars/word
 - Exercise: Why is/isn't this the number to use for estimating the dictionary size?
- Ave. English dictionary word: ~8 characters

Compressing the term list: Dictionary-as-a-String

- 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.

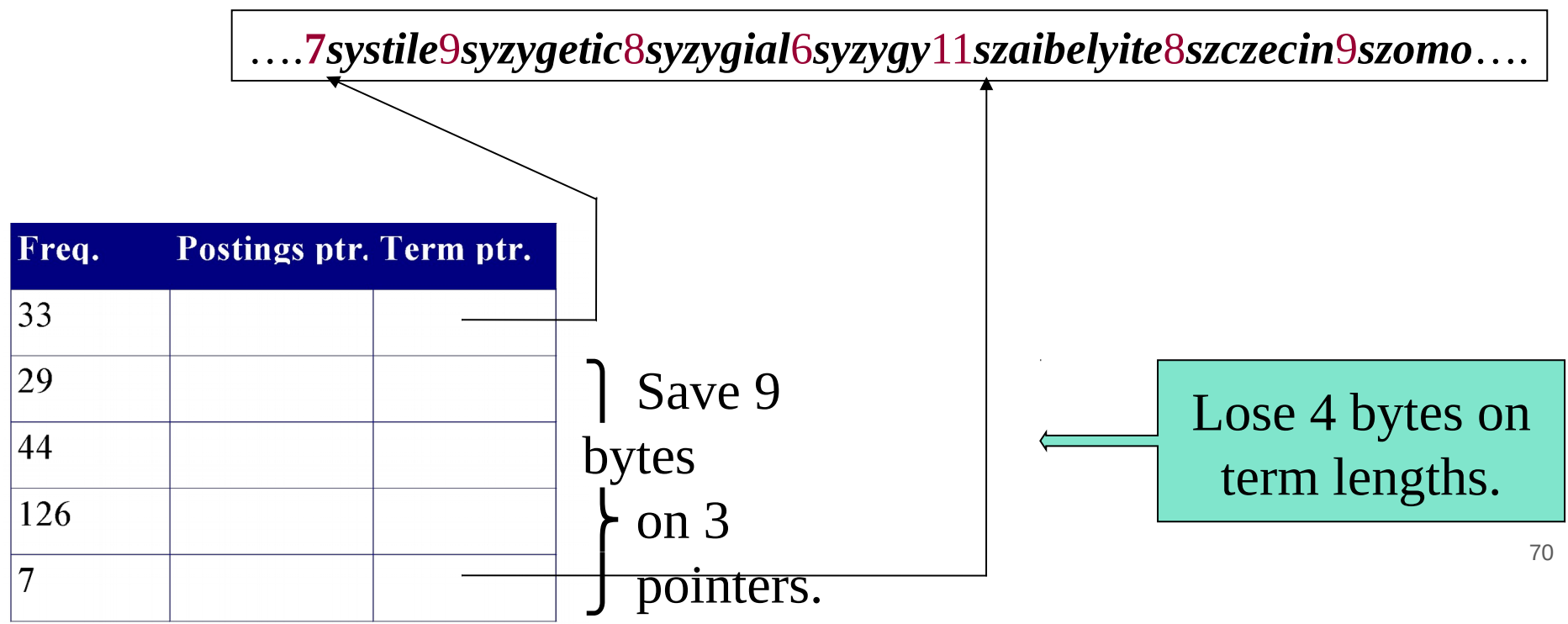


Space for dictionary as a string

- 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
 - 400K terms x 19 \Rightarrow 7.6 MB (against 11.2MB for fixed width)
- } Now avg.
 11 bytes/term,
 } not 20.

Blocking

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



Net

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
 - $3 \times 4 = 12$ bytes,

now we use $3 + 4 = 7$ bytes.

Shaved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

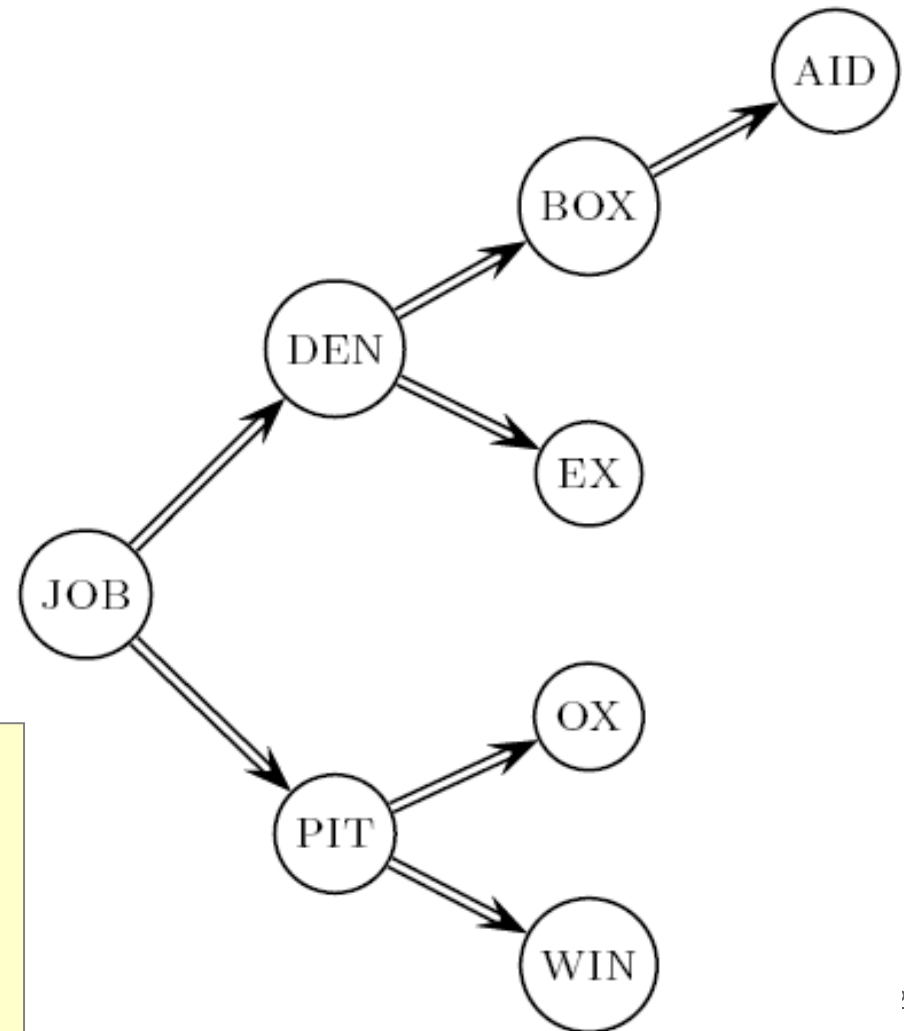
We can save more with larger k .

Why not go with larger k ?

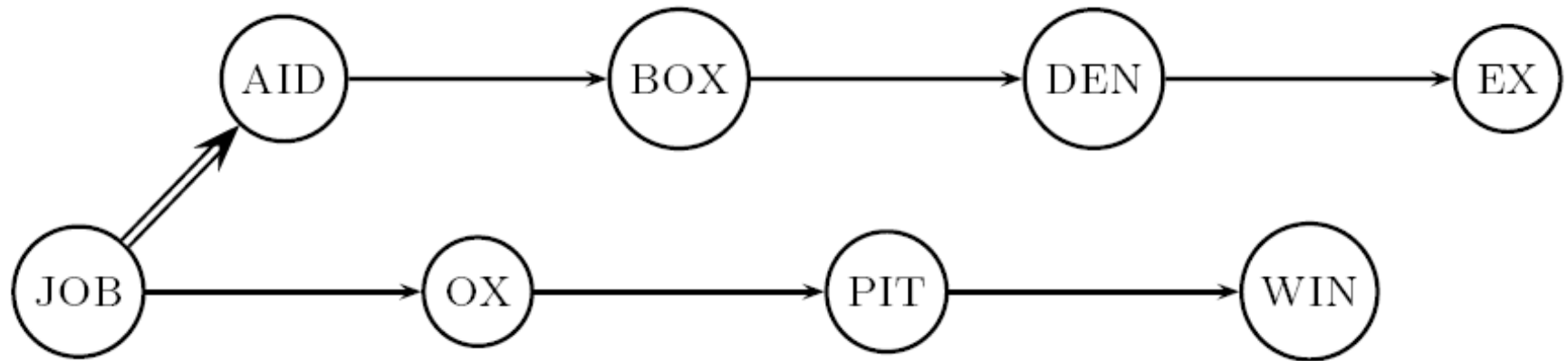
Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons = $(1+2 \cdot 2+4 \cdot 3+4)/8 \sim 2.6$

If the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?



Dictionary search with blocking



- Binary search down to 4-term block;
 - Then linear search through terms in block.
- Blocks of 4 (binary tree), avg. = $(1+2 \cdot 2+2 \cdot 3+2 \cdot 4+5)/8 = 3$ compares

Front coding

■ 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**automatio**
 $n \rightarrow$ 8**automat***a1◇e2◇ic3◇io
n

Encodes **automat**

Extra length
beyond **automat.**

Begins to resemble general string compression.

RCV1 dictionary compression summary

Technique	Size in MB
Fixed width	11.2
Dictionary-as-String with pointers to every term	7.6
Also, blocking $k = 4$	7.1
Also, Blocking + front coding	5.9

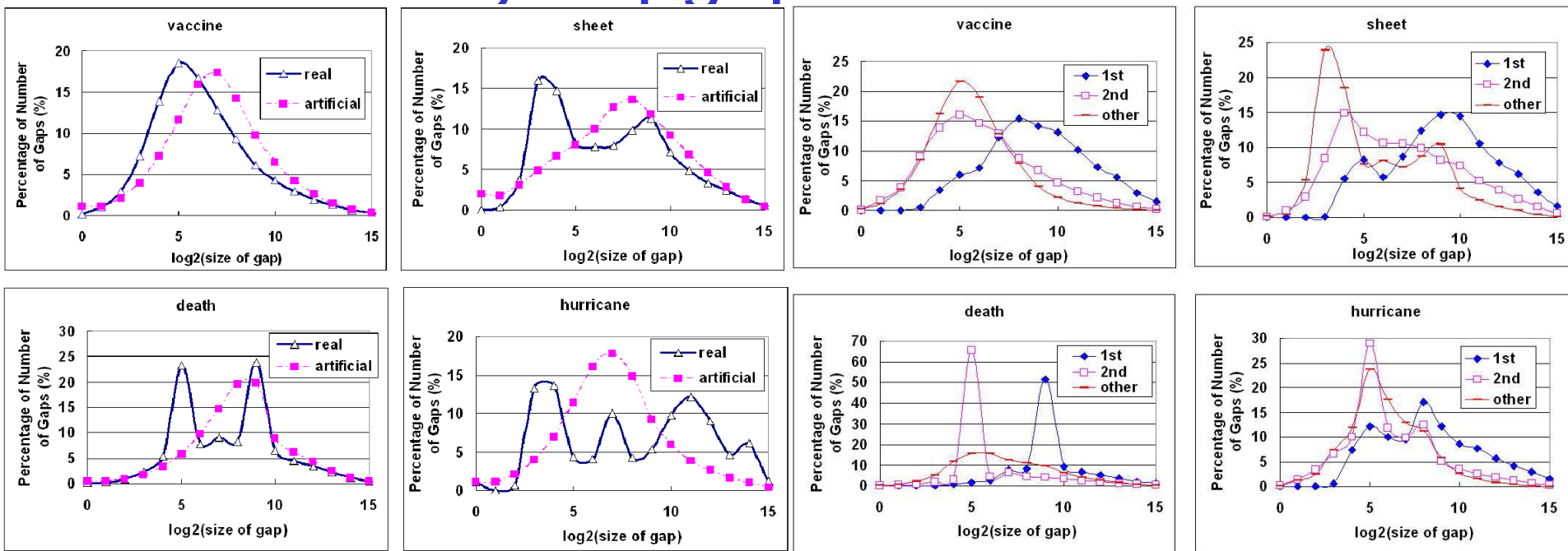
Index compression summary

- 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.

Other compression codes

- Gamma code is only one of many other possibilities, e.g. Golomb/Rice code
- Best choice depends on distribution of gaps
- Dgaps and pgaps behave differently
- Ditto for first and subsequent pgaps
- Can do doc-specific pgap compression

Study of pgap distributions



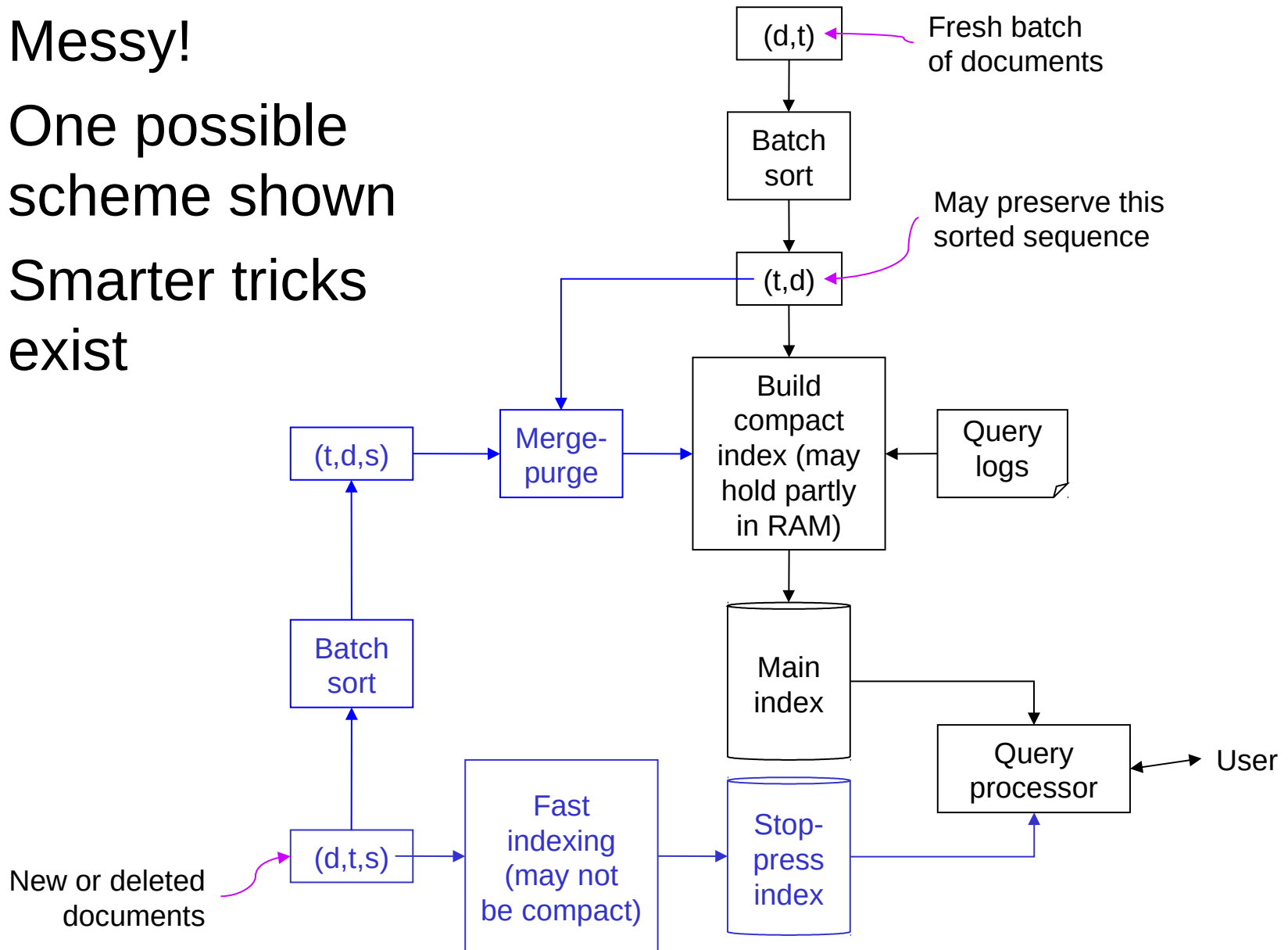
Pgap distribution very different from what would result by random placement of tokens in document of given length

Gap from beginning to first term occurrence very different from second and subsequent gaps

X-axis= $\log(\text{pgap})$, y-axis=frequency

Updating the index

- Messy!
- One possible scheme shown
- Smarter tricks exist



Web search engine data centers

- Web search data centers (Google, Bing, Baidu) mainly contain commodity machines
- Data centers are distributed around the world
- Ancient estimate: Google ~1 million servers, 3 million processors/cores (Gartner 2007)

Massive data centers

- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system?
- Answer: 63%
- Exercise: Calculate the number of servers failing per minute for an installation of 1 million servers.

Heavier iron

GFS / HDFS: Distributed replicated fault-tolerant file system

Map-reduce / Hadoop: Bulk-synchronous parallel computation paradigm on top of GFS/HDFS

SSTable: “[Sorted strings table](#)”—Persistent sorted immutable key-value map, typically stored on GFS/HDFS

[BigTable](#) / Hbase: Distributed, persistent, fault-tolerant map where key = (row_string, column_string, timestamp) and cell value = arbitrary binary payload

[Percolator](#): Workflow management implemented on top of BigTable; enables asynchronous crawling and indexing

Distributed indexing

- Maintain a *master* machine directing the indexing job – considered “safe”.
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.

Parallel tasks

- We will use two sets of parallel tasks
 - Parsers (scan and tokenize)
 - Inverters or ‘transposers’
- Break the input document collection into *splits*
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)

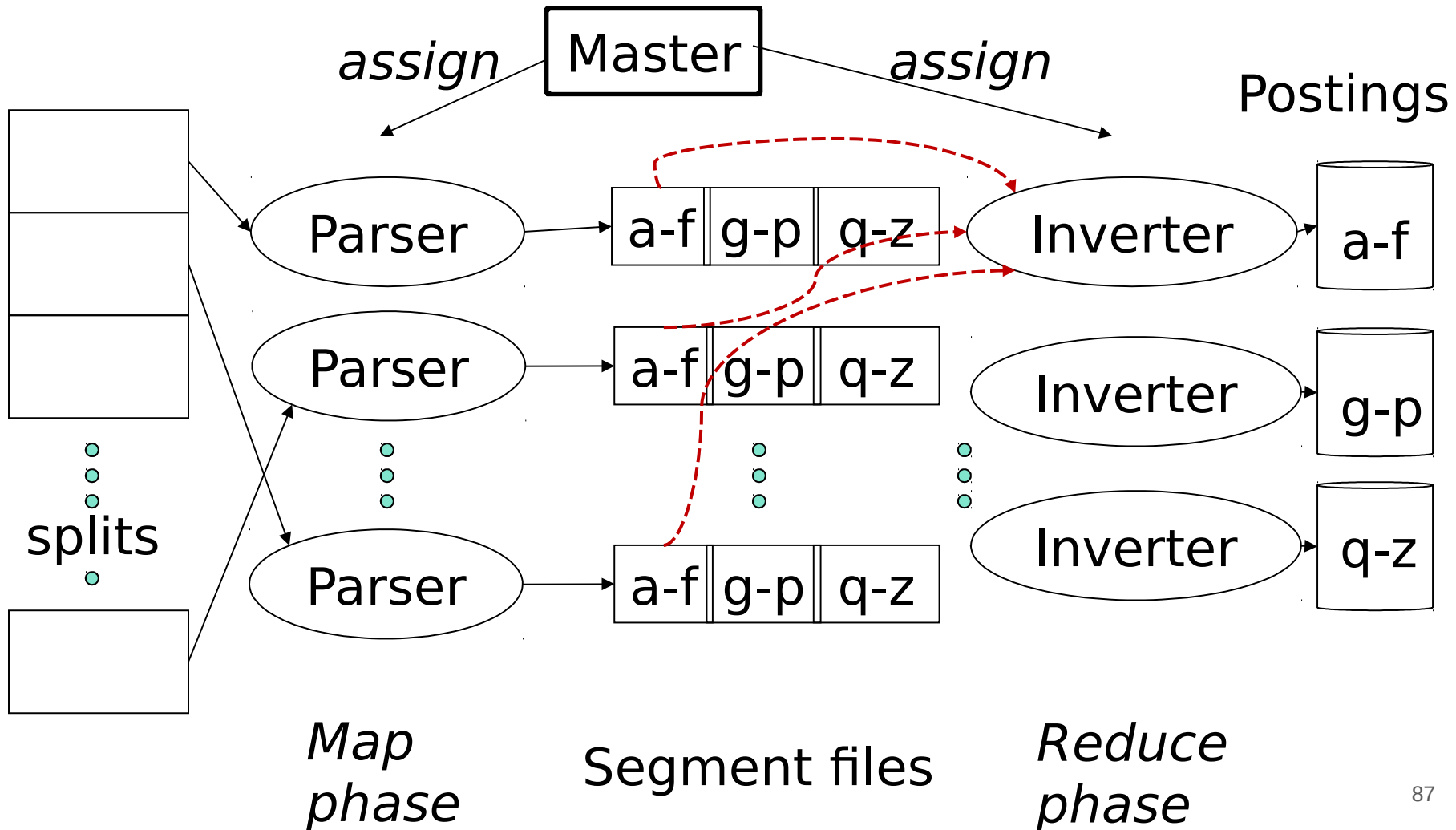
Parsers

- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, doc) pairs
- 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 $j = 3$.
- Now to complete the index inversion

Inverters

- An inverter collects all (term,doc) pairs (= postings) for one term-partition.
- Sorts and writes to postings lists

Data flow



MapReduce

- The index construction algorithm we just described is an instance of *MapReduce*.
- MapReduce (Dean and Ghemawat 2004) is a robust and conceptually simple framework for distributed computing ...
- ... without having to write code for distribution and coordination
- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.

Schema for index construction in MapReduce

- **Schema of map and reduce functions**
- $\text{map: input} \rightarrow \text{list}(k, v)$ $\text{reduce: } (k, \text{list}(v)) \rightarrow \text{output}$
- **Instantiation of the schema for index construction**
- $\text{map: collection} \rightarrow \text{list}(\text{termID}, \text{docID})$
- $\text{reduce: } (<\text{termID1}, \text{list}(\text{docID})>, <\text{termID2}, \text{list}(\text{docID})>, \dots) \rightarrow (\text{postings list1}, \text{postings list2}, \dots)$

Example for index construction

- Map input = { d1 : C came, C sat;
d2 : C spoke }
- Map output = { <C,d1>, <came,d1>,
<C,d1>, <sat, d1>, <C, d2>, <spoke,d2> }
- Reduce input = (<C,(d1,d1,d2)>, <spoke,
(d2)>, <came,(d1)>, <sat,(d1)>)
- Reduce output = (<C,(d1:2,d2:1)>, <spoke,
(d2:1)>, <came,(d1:1)>, <sat,(d1:1)>)

After creating term-partitioned index

- Index construction was just one phase.
- Another phase: transforming a term-partitioned index into a document-partitioned index.
 - *Term-partitioned*: one machine handles a subrange of terms
 - *Document-partitioned*: one machine handles a subrange of documents
- Most search engines use a document-partitioned index ... better load balancing, etc.

Other real-world Web indexing issues

- Source format and language detection
 - Multi-lingual and code-switched documents
- Sentence and word delimiter, punctuation
- Case normalization
 - MIT in English vs. mit in German
- Morphological normalization (“stemming”)
- Compound word (“multi word”) detection
- Multilingual dictionary
 - Preferably unsupervised or weakly supervised