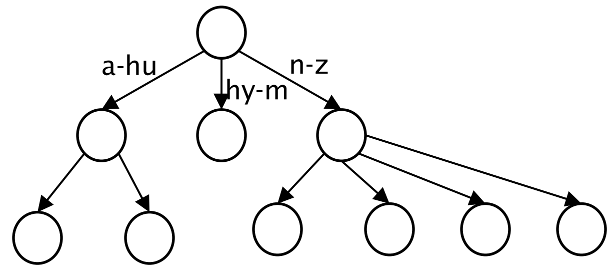
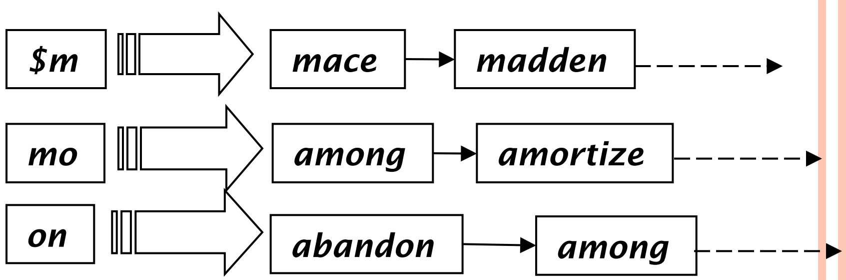
Index Construction

Plan

- Last lecture:
  - Dictionary data structures
  - Tolerant retrieval
    - Wildcards
    - Spell correction
    - Soundex

- This time:
  - Index construction





Index construction

- How do we construct an index?
- What strategies can we use with limited main memory?

Hardware basics

- Many design decisions in information retrieval are based on the characteristics of hardware
- We begin by reviewing hardware basics

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 several GB of main memory, sometimes hundreds of GB.
- Available disk space is several (2–3) orders of magnitude larger.
- Fault tolerance is very expensive: It's much cheaper to use many regular machines rather than one fault tolerant machine. (redundancy)

Hardware assumptions for this lecture

symbol	statistic	value
s	average seek time	5 ms = 5 x 10^{-3} s
b	transfer time per byte	0.02 μs = 2 x 10^{-8} s
	processor's clock rate	109 s^{-1}
p	low-level operation (e.g., compare & swap a word)	0.01 μs = 10^{-8} s
	size of main memory	several GB
	size of disk space	1 TB or more

RCV1: Our collection for this lecture

- Shakespeare's collected works definitely aren't large enough for demonstrating many of the points in this course.
- The collection we'll use isn't really large enough either, but it's publicly available and is at least a more plausible example.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
- This is one year of Reuters newswire (Aug 1996 to Aug 1997)

A Reuters RCV1 document

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.

Reuters RCV1 statistics

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	terms (= word types)	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
	# of tokens	100,000,000

Quiz: Why is the avg. # bytes per term larger than the # bytes per token?

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.
We have 100M items to sort.

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

SCALING INDEX CONSTRUCTION

- In-memory index construction does not scale
  - Can't stuff 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.
  - While building the index, we cannot easily exploit compression tricks (you can, but much more complex)
- The final postings for any term are incomplete until the end.
- At 12 bytes per non-positional postings entry *(termid, docid, freq)*, demands a lot of space for large collections.
- T = 100,000,000 records in the case of RCV1
  - So … we can do this in memory in 2023, but typical collections are much larger. E.g., the *New York Times* provides an index of >150 years of newswire
- Thus: We need to 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: Sorting T = 100,000,000 records on disk is too slow – too many disk seeks.
- We need an *external* sorting algorithm.

BOTTLENECK

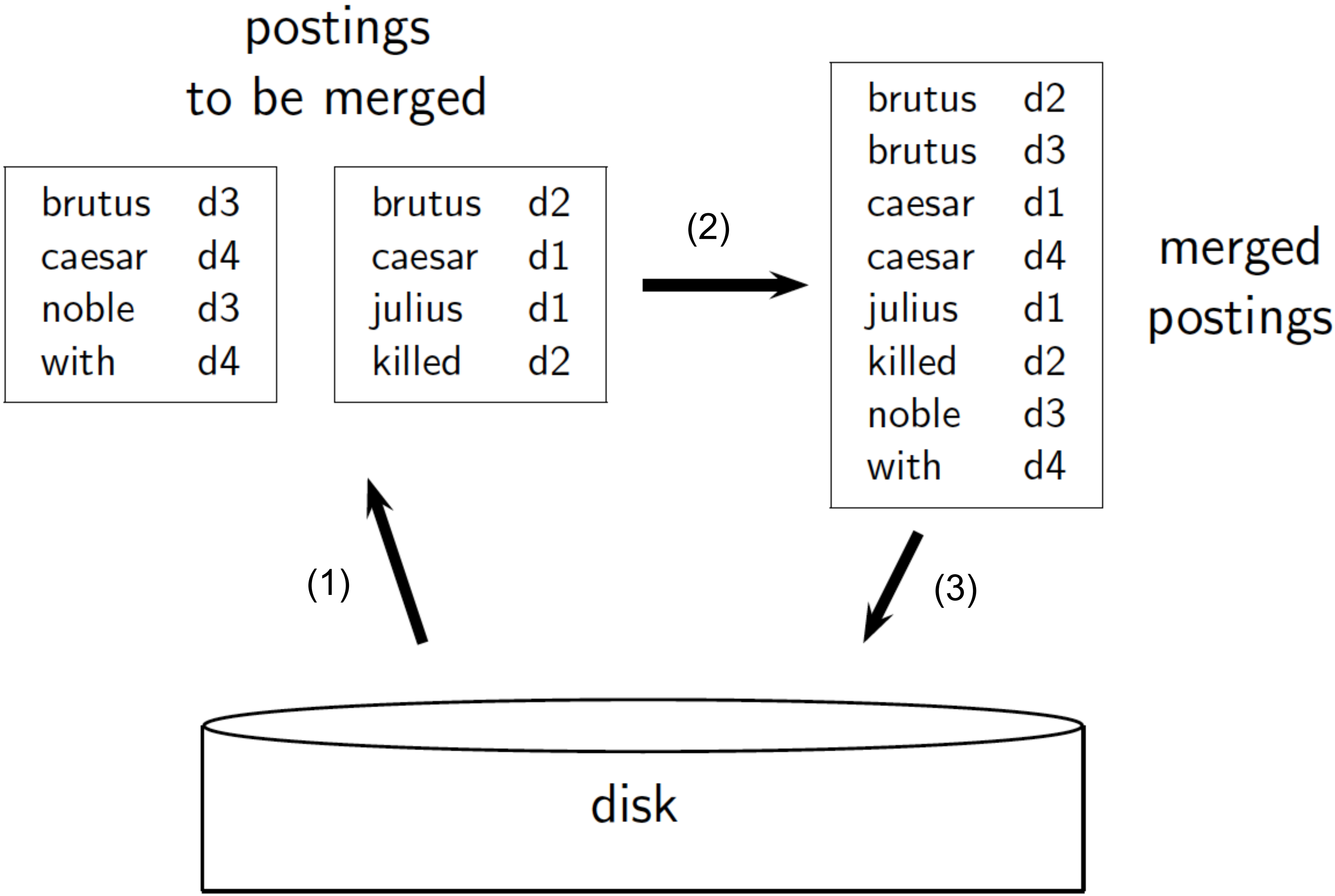
- Parse and build postings entries one doc at a time
- Now sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow – must sort T=100M records

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

s	average seek time	5 ms = 5×10^{-3} s
b	transfer time per byte	0.02 μs = 2×10^{-8} s
	processor's clock rate	10^9 s^{-1}
p	low-level operation	0.01 μs = 10^{-8} s

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

- 12-byte (4+4+4) records *(termid, docid, freq)*.
- These are generated as we parse docs.
- Must now sort 100M such 12-byte records by *term*.
- Define a <u>Block</u> ~10M such records
  - Can easily fit a couple 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
(2)
brutus d2
brutus d3
caesar d1
caesar d4
julius d1
killed d2
noble d3
with d4
merged
postings
(1)
(3)
disk

Sorting 10 blocks of 10M records

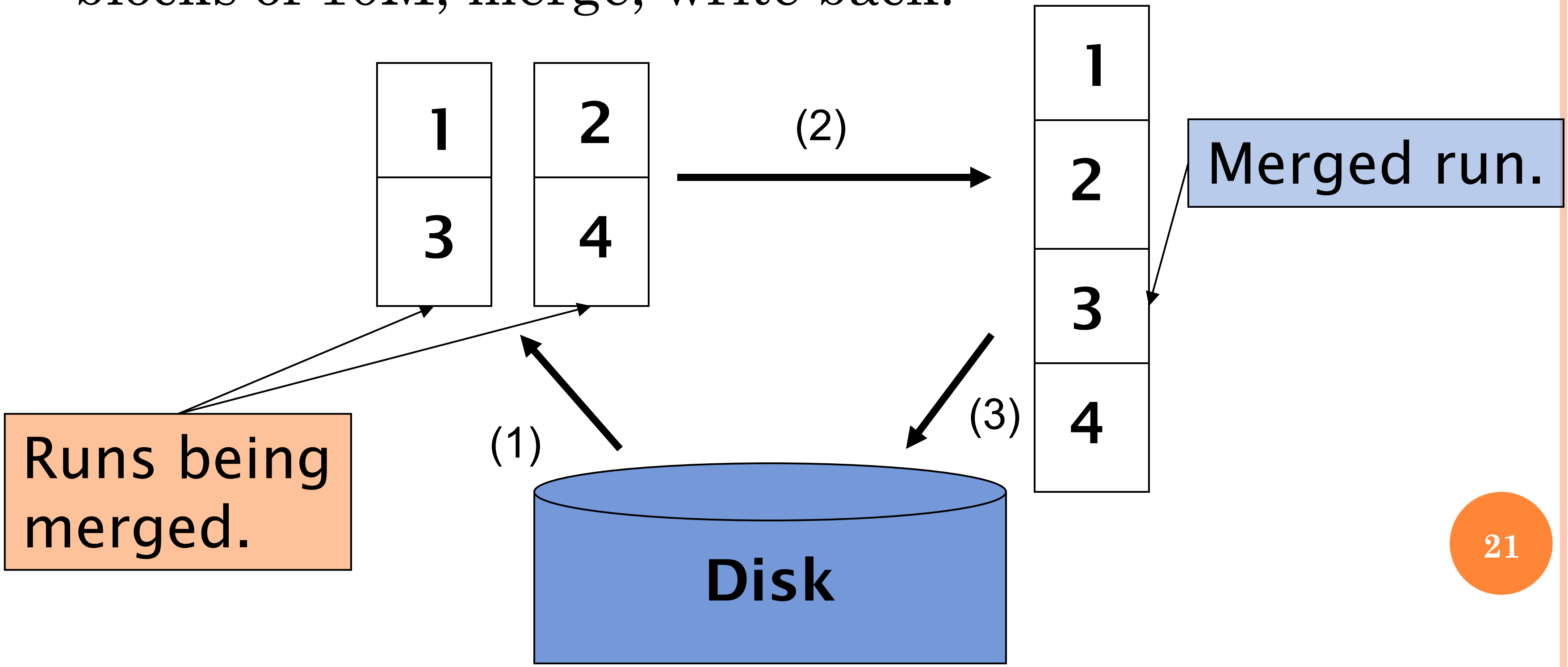
- First, read each block and sort within:
  - Quicksort takes $2N \ln N$ expected steps
  - In our case 2*10M ln 10M steps
- *Exercise: estimate total time to read each block from disk and quicksort it.*
- 10 times this estimate – gives us 10 sorted <u>*runs*</u> of 10M records each.
- Done straightforwardly,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

Quiz: Why do we need 2 copies of data on the disk?

```
BSBIndexConstruction()
n ← 0
while (all documents have not been processed)
do n ← n + 1
    block ← ParseNextBlock()
    BSBI-Invert(block)
    WriteBlockToDisk(block, f_n)
MergeBlocks(f_1, ..., f_n; f_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.



HOW TO MERGE THE SORTED RUNS?

- But it is more efficient to do a multi-way (in stead of binary) merge, where you are reading from all blocks simultaneously
- Providing you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you're not killed by disk seeks
- Typically there's an input buffer and output buffer; write out to disk when the buffer is full.

Remaining problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a *term* to *termID* mapping.
- Actually, we could work with term,docID postings instead of termID,docID postings . . .
- . . . but then intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)

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)
output_file = NewFile()
dictionary = NewHash()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
      then postings_list = AddToDictionary(dictionary, term(token))
      else postings_list = GetPostingsList(dictionary, term(token))
    if full(postings_list)
      then postings_list = DoublePostingsList(dictionary, term(token))
    AddToPostingsList(postings_list, docID(token))
sorted_terms ← SortTerms(dictionary)
WriteBlockToDisk(sorted_terms, dictionary, output_file)
return output_file
```

Merging of blocks is analogous to BSBI.

SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
- See next lecture

Distributed indexing

- For web-scale indexing (don't try this on your PC!):
  must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?

Web search engine data centers

- Web search data centers (Google, Bing, Baidu) mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: Google ~2.5 million servers, 7.2 million processors/cores (Gartner 2016)

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: 37% (0.999^1000)

Quiz: Failed Servers

Suppose a server will fail after 4 years. For an installation of 1 million servers, what is the average time interval between machine failures?

Distributed indexing

- Maintain a *master* machine directing the indexing job – considered "failsafe".
- 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
  - Inverters
- Break the input document collection into *splits*
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)

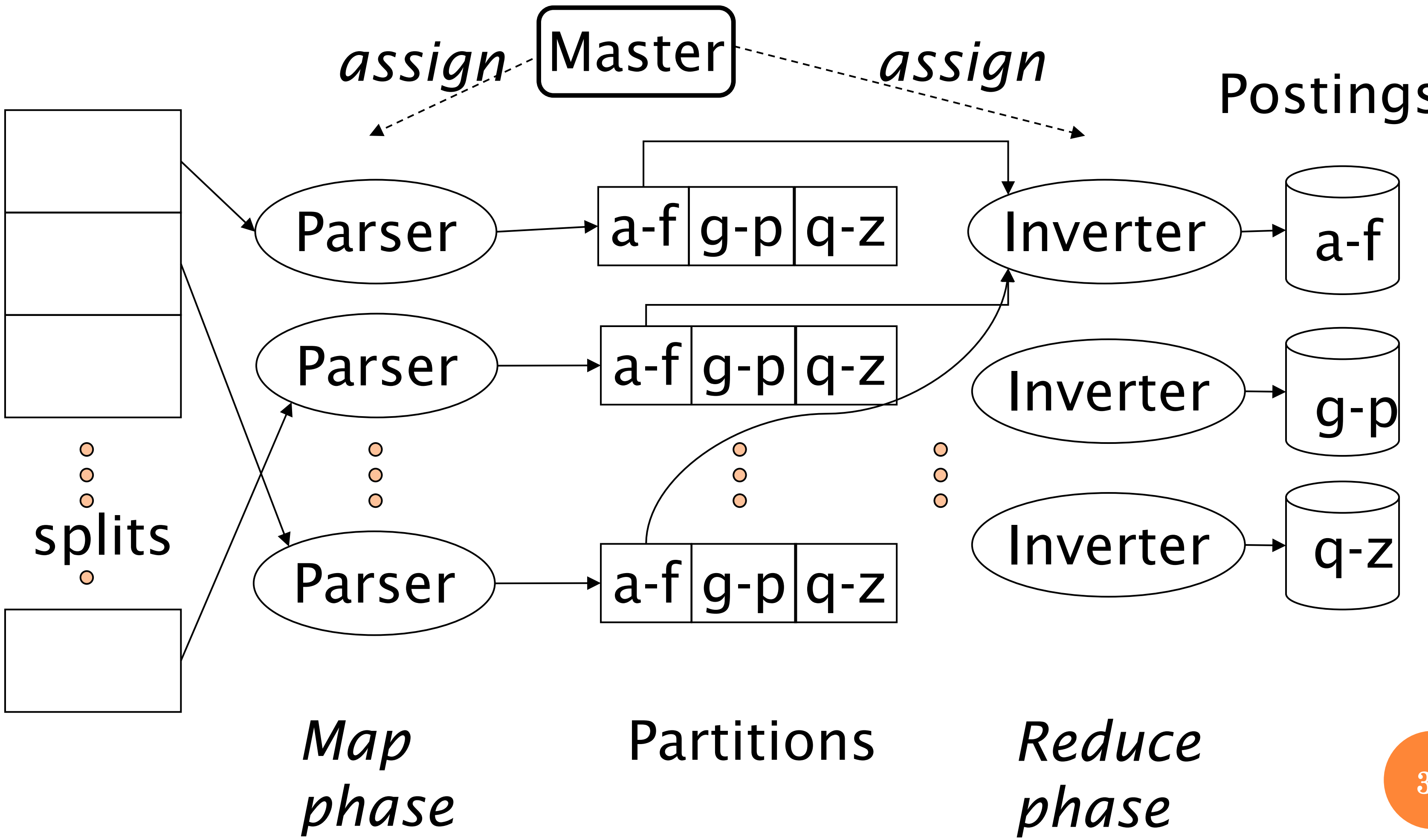
PARSERS

- 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 <u>one</u> term-partition.
- Sorts and writes to postings lists

DATA FLOW


Master
assign
assign
Postings
Parser
Parser
Parser
a-f
g-p
q-z
a-f
g-p
q-z
a-f
g-p
q-z
Inverter
Inverter
Inverter
a-f
g-p
q-z
splits
Map phase
Partitions
Reduce phase

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 the distribution part.
- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.

MapReduce

- 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
- As we'll discuss in the web part of the course, most search engines use a document-partitioned index … better load balancing, etc.

SCHEMA FOR INDEX CONSTRUCTION IN MAPREDUCE

- **Schema of map and reduce functions**

  map: input → list(k, v)

  reduce: (k,list(v)) → output

- **Instantiation of the schema for index construction**

  map: collection → list(termID, docID)

  reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …) → (postings list1, postings list2, …)

Example for index construction

- Map:

  d1 : C came, C c'ed.

  d2 : C died. →

  <C,d1>, <came,d1>, <C,d1>, <c'ed, d1>, <C, d2>, <died,d2>

- Reduce:

  (<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <c'ed,(d1)>) →
  (<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c'ed,(d1:1)>)

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are:
  - Documents come in over time and need to be inserted.
  - Documents are deleted and modified.
- This means that the dictionary and postings lists have to be modified:
  - Postings updates for terms already in dictionary
  - New terms added to dictionary

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

Issues with main and auxiliary indexes

- Problem of frequent merges – you touch stuff a lot
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient for OS.
- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists, collect all postings lists of length 1 in one file etc.)

Logarithmic merge

- Logarithmic merging amortizes the cost of merging indexes over time.
  - → Users see smaller effect on response times.
- Maintain a series of indexes, each twice as large as the previous one
  - At any time, some of these powers of 2 are instantiated
- Keep smallest (Z_0) in memory
- Larger ones (I_0, I_1, …) on disk
- If Z_0 gets too big ($> n$), write to disk as I_0
- . . . or merge with I_0 (if I_0 already exists) and write merger to I_1 etc.

LMergeAddToken(*indexes*, Z_0, *token*)

$Z_0 \leftarrow$ Merge(Z_0, $\{token\}$)
if $|Z_0| = n$
  then for $i \leftarrow 0$ **to** ∞
    do if $I_i \in$ *indexes*
      then $Z_{i+1} \leftarrow$ Merge(I_i, Z_i)
        (Z_{i+1} is a temporary index on disk.)
        indexes $\leftarrow$ *indexes* $- \{I_i\}$
      else $I_i \leftarrow Z_i$ *(Z_i becomes the permanent index I_i.)*
        indexes $\leftarrow$ *indexes* $\cup \{I_i\}$
        Break
    $Z_0 \leftarrow \emptyset$

LogarithmicMerge()

$Z_0 \leftarrow \emptyset$ *(Z_0 is the in-memory index.)*
indexes $\leftarrow \emptyset$
while true
do LMergeAddToken(*indexes*, Z_0, getNextToken())

LOGARITHMIC MERGE

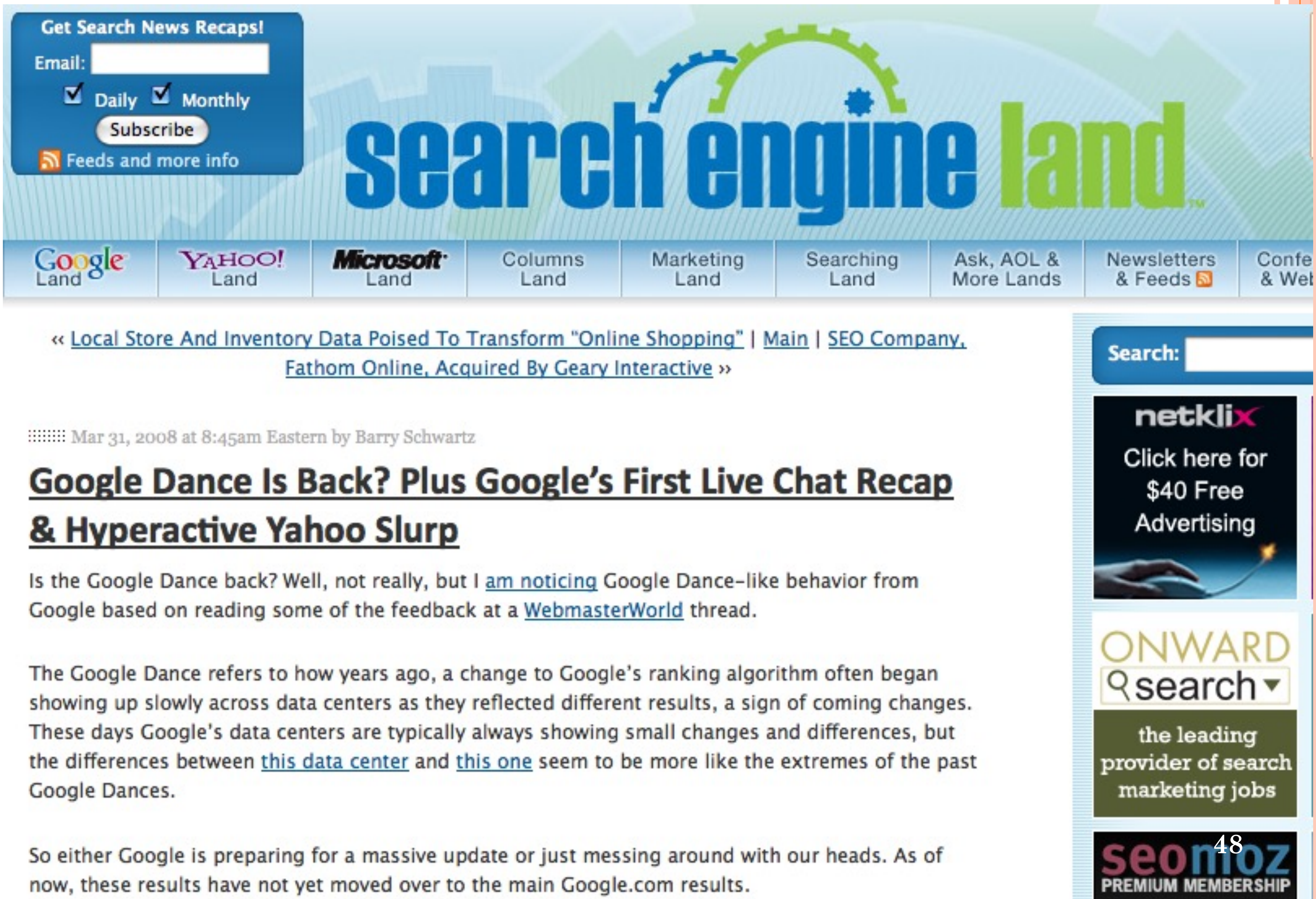
- Auxiliary and main index: index construction time is $O(T^2)$ as each posting is touched in each merge.
- Logarithmic merge: Each posting is merged O(log T) times, so complexity is O(T log T)
- So logarithmic merge is much more efficient for index construction
- But query processing now requires the merging of O(log T) indexes
  - Whereas it is O(1) if you just have a main and auxiliary index

FURTHER ISSUES WITH MULTIPLE INDEXES

- Collection-wide statistics are hard to maintain
- E.g., when we spoke of spell-correction: which of several corrected alternatives do we present to the user?
  - We said, pick the one with the most hits
- How do we maintain the top ones with multiple indexes and invalidation bit vectors?
  - One possibility: ignore everything but the main index for such ordering
- Will see more such statistics used in results ranking

Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
    - Volodymyr Zelenskyy, …
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is deleted



Get Search News Recaps!
Email:
Daily Monthly
Subscribe
Feeds and more info

search engine land

Google Land | Yahoo! Land | Microsoft Land | Columns Land | Marketing Land | Searching Land | Ask, AOL & More Lands | Newsletters & Feeds | Confe & Wel

« Local Store And Inventory Data Poised To Transform "Online Shopping" | Main | SEO Company, Fathom Online, Acquired By Geary Interactive »

Mar 31, 2008 at 8:45am Eastern by Barry Schwartz

Google Dance Is Back? Plus Google's First Live Chat Recap & Hyperactive Yahoo Slurp

Is the Google Dance back? Well, not really, but I am noticing Google Dance-like behavior from Google based on reading some of the feedback at a WebmasterWorld thread.

The Google Dance refers to how years ago, a change to Google's ranking algorithm often began showing up slowly across data centers as they reflected different results, a sign of coming changes. These days Google's data centers are typically always showing small changes and differences, but the differences between this data center and this one seem to be more like the extremes of the past Google Dances.

So either Google is preparing for a massive update or just messing around with our heads. As of now, these results have not yet moved over to the main Google.com results.

Search:

OTHER SORTS OF INDEXES

- Positional indexes
  - Same sort of sorting problem … just larger

Why?

- Building character n-gram indexes:
  - As text is parsed, enumerate *n*-grams.
  - For each *n*-gram, need pointers to all dictionary terms containing it – the "postings".
  - Note that the same "postings entry" will arise repeatedly in parsing the docs – need efficient hashing to keep track of this.
    - E.g., that the trigram <u>*uou*</u> occurs in the term ***deciduous*** will be discovered on each text occurrence of ***deciduous***
    - Optimization: only process each term once

Resources for today's lecture

- Chapter 4 of IIR
- MG Chapter 5
- Original publication on MapReduce: Dean and Ghemawat (2004)
- Original publication on SPIMI: Heinz and Zobel (2003)

Index Compression

TODAY

BRUTUS	→	1	2	4	11	31	45	173	174	
CAESAR	→	1	2	4	5	6	16	57	132	. . .
CALPURNIA	→	2	31	54	101					

- Collection statistics in more detail (with RCV1)
  - How big will the dictionary and postings be?
- Dictionary compression
- Postings compression

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

Recall Reuters RCV1

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	terms (= word types)	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
	non-positional postings	100,000,000

Lossless vs. lossy compression

- Lossless compression: All information is preserved.
  - 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.
- Later: Prune postings entries that are unlikely to turn up in the top k list for any query.
  - Almost no-loss for top k list.

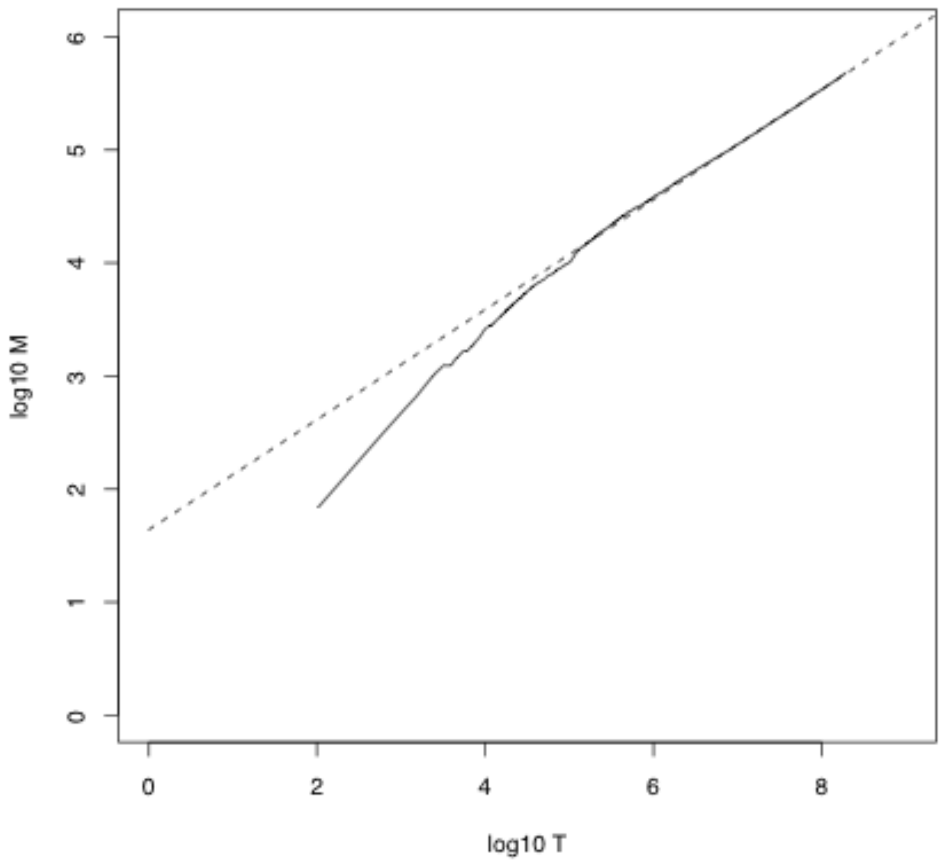
VOCABULARY VS. COLLECTION SIZE

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Not really: At least $70^{20} = 2^{123}$ different words of length 20
  - 70 different characters incl. digits, punctuations and accents
- In practice, the vocabulary will keep growing with the collection size
  - Especially with Unicode ☺

VOCABULARY VS. COLLECTION SIZE

- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection
- Typical values: $30 \le k \le 100$ and $b \approx 0.5$
- In a log-log plot of vocabulary size M vs. T, Heaps' law predicts a line with slope about ½
  - It is the simplest possible relationship between the two in log-log space
  - An empirical finding ("empirical law")

Heaps' law for Reuters



Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1.

For these data, the dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least squares fit.

Thus, $M = 10^{1.64} T^{0.49}$ and $k = 10^{1.64} \approx 44$ and $b = 0.49$.

QUIZ: VOCABULARY SIZE

- Looking at a collection of web pages, you find that there are 4000 different terms in the first 10,000 tokens and 40,000 different terms in the first 1,000,000 tokens.
- Assume a search engine indexes a total of 3,000,000 (3×10^6) pages, 300 tokens per page on average
- What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

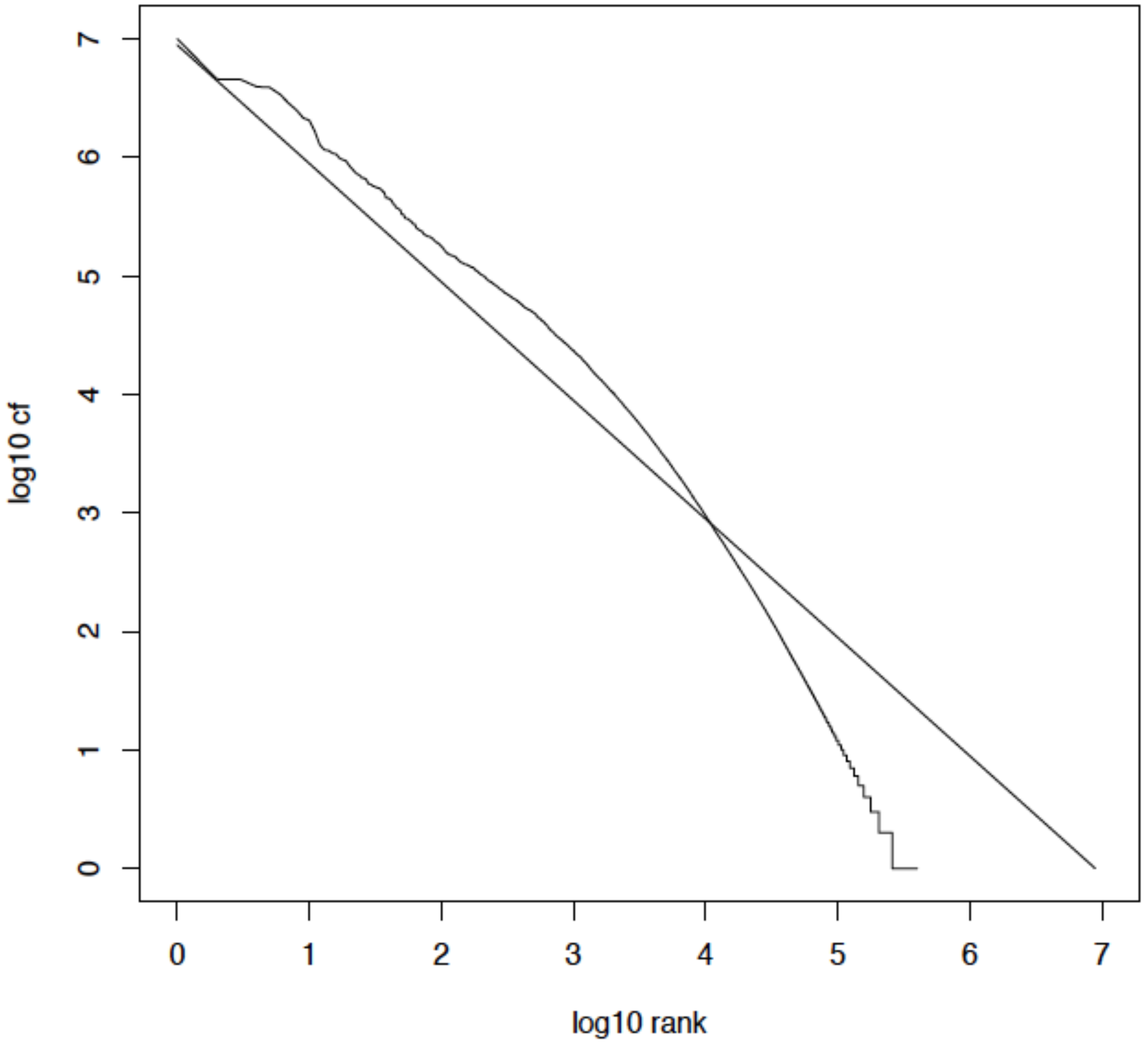
Zipf's law

- Heaps' law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In natural language, there are a few very frequent terms and many, many very rare terms.
- Zipf's law: The *i*th most frequent term has frequency proportional to $1/i$.
- $\mathrm{cf}_i \propto 1/i = K/i$ where K is a normalizing constant
- cf_i is <u>collection frequency</u>: the number of occurrences of the term t_i in the collection.

Zipf consequences

- If the most frequent term (*the*) occurs cf_1 times
  - then the second most frequent term (*of*) occurs $cf_1/2$ times
  - the third most frequent term (*and*) occurs $cf_1/3$ times …
- Equivalent: $cf_i = K/i$ where K is a normalizing factor, so
  - $\log cf_i = \log K - \log i$
  - Linear relationship between $\log cf_i$ and $\log i$
- Another power law relationship

Zipf's law for Reuters RCV1



COMPRESSION

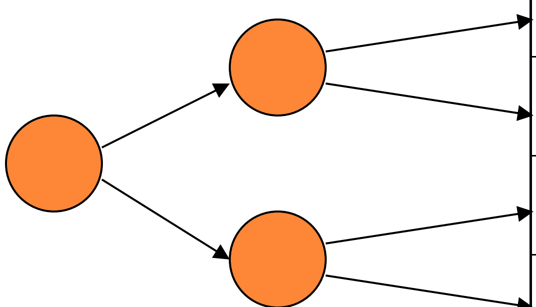
- Now, we will consider compressing the space for the dictionary and postings
  - Basic Boolean index only
  - No study of positional indexes, etc.
  - We will consider compression schemes

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 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.



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

Dictionary search structure

20 bytes

4 bytes each

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 characters/word.
  - Exercise: Why is/isn't this the number to use for estimating the dictionary size?
- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not term average.

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.

….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

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

Now avg. 11 bytes/term, not 20.

- Avg. 8 bytes per term in term string
- 400K terms x 19 $\Rightarrow$ 7.6 MB (against 11.2MB for fixed width)

Blocking

- Store pointers to every *k*th 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

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
  - 3 x 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*.

QUIZ: LARGE OR SMALL?

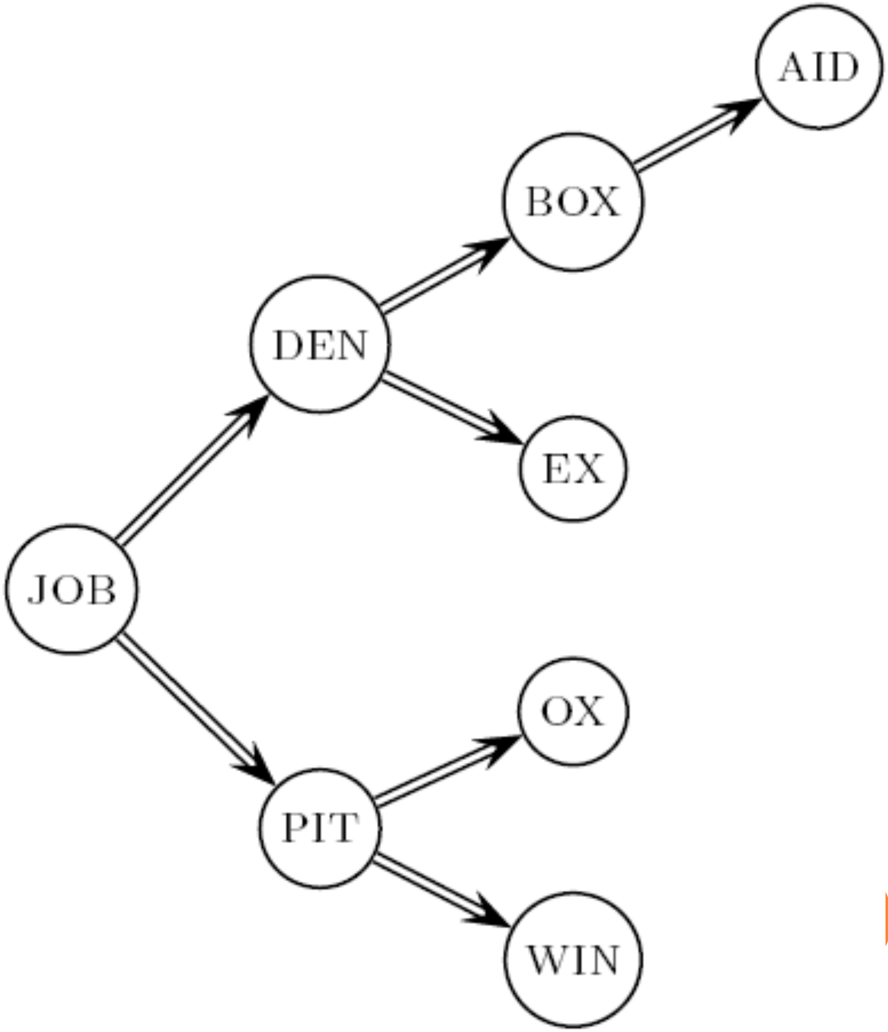
- What's the pros and cons for a Large K and a Small K?

--- Write down at least 1 pro and 1 con.

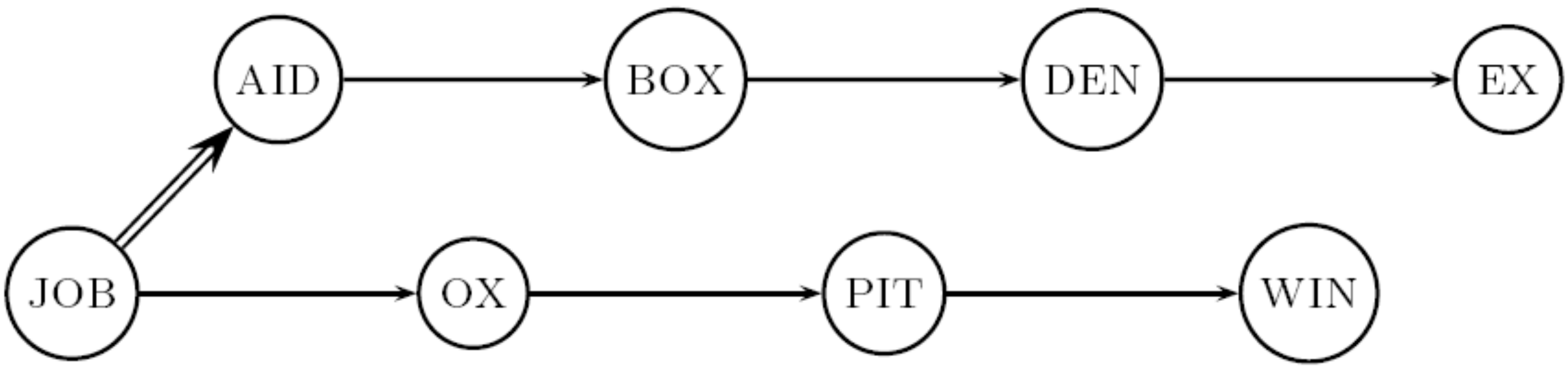
DICTIONARY SEARCH WITHOUT BLOCKING

- Assuming each dictionary term equally likely to appear in query (not really so in practice!), average number of comparisons = (1+2·2+4·3+4)/8 ~2.6

Exercise: What 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