Information Retrieval

Lecture 4

Recap of last time

- Postings pointer storage
- Dictionary storage
- Compression
- Wild-card queries

This lecture

- Query expansion
  - What queries can we now process?
- Index construction
  - Estimation of resources

Spell correction and expansion

Expansion

- Wild-cards were one way of “expansion”:
  - i.e., a single “term” in the query hits many postings.
- There are other forms of expansion:
  - Spell correction
  - Thesauri
  - Soundex (homonyms).

Spell correction

- Expand to terms within (say) edit distance 2
  - Edit∈{Insert/Delete/Replace}
  - Expand at query time from query
  - *e.g., Alanis Morisette*
- Spell correction is expensive and slows the query (upto a factor of 100)
  - Invoke only when index returns (near-)zero matches.
  - What if docs contain mis-spellings?

Thesauri

- Thesaurus: language-specific list of synonyms for terms likely to be queried
  - Generally hand-crafted
  - Machine learning methods can assist – more on this in later lectures.

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific
  - E.g., *chebyshev* → *tchebycheff*
- How do we use thesauri/soundex?
  - Can "expand" query to include equivalences
    - Query *car tyres* → *car tyres automobile tires*
  - Can expand index
    - Index docs containing *car* under *automobile*, as well

Query expansion

- Usually do query expansion rather than index expansion
  - No index blowup
  - Query processing slowed down
    - Docs frequently contain equivalences
  - May retrieve more junk
    - *puma* $\rightarrow$ *jaguar* retrieves documents on cars instead of on sneakers.

Language detection

- Many of the components described require language detection
  - For docs/paragraphs at indexing time
  - For query terms at query time – much harder
- For docs/paragraphs, generally have enough text to apply machine learning methods
- For queries, generally lack sufficient text
  - Augment with other cues, such as client properties/specification from application
  - Domain of query origination, etc.

What queries can we process?

- We have
  - Basic inverted index with skip pointers
  - Wild-card index
  - Spell-correction
- Queries such as

*an*er* AND (moriset /3 toronto)*

Aside – results caching

- If 25% of your users are searching for

  britney AND spears

  then you probably *do* need spelling correction, but you *don't* need to keep on intersecting those two postings lists
- Web query distribution is extremely skewed, and you can usefully cache results for common queries – more later.

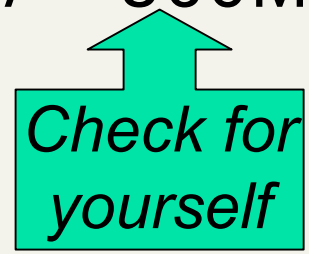
Index construction

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:
- $n + n/2 + n/3 + ... + n/m \sim n \ln m =$ 560M entries
- No positional info yet



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

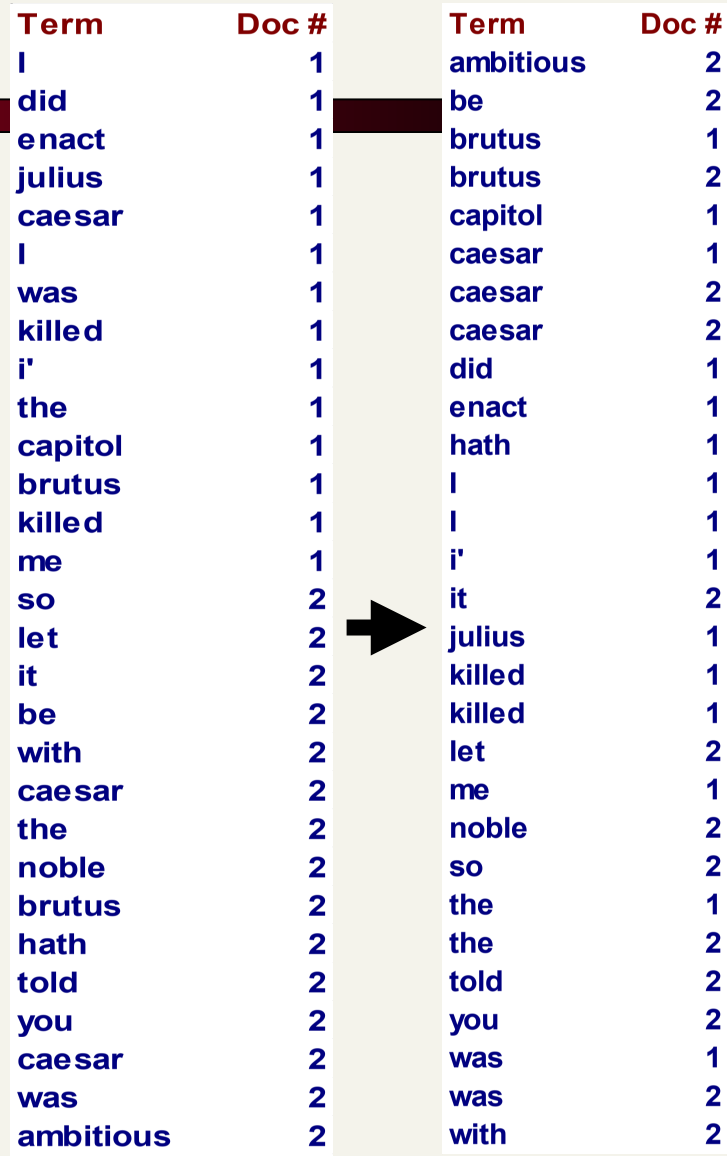
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

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

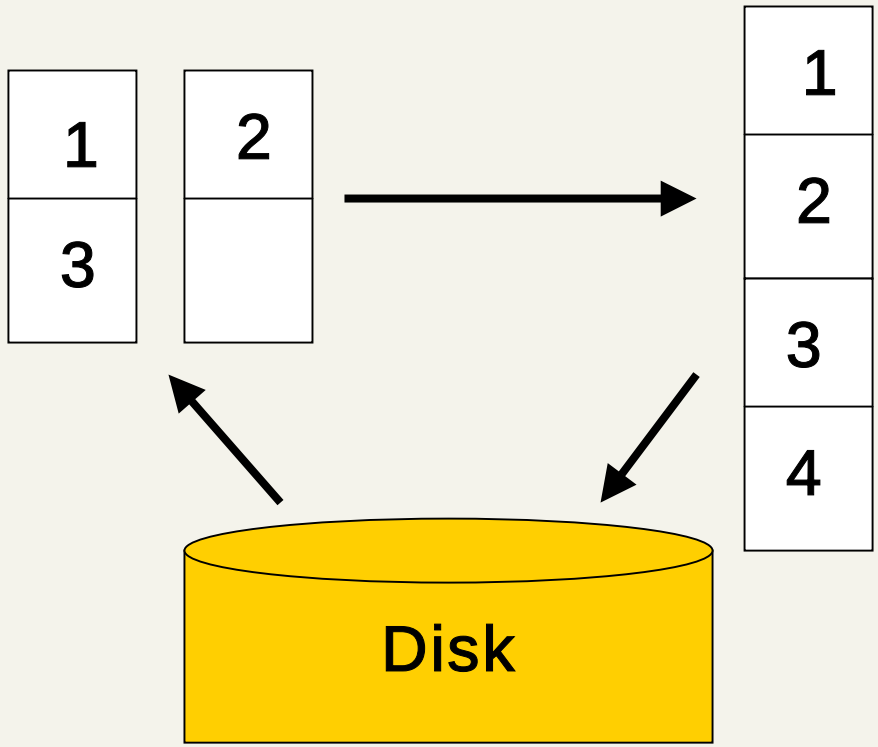
- 12-byte (4+4+4) records *(term, doc, freq).*
- These are generated as we parse docs.
- Must now sort 560M such 12-byte records by *term*.
- Define a Block = 10M such records
  - can “easily” fit a couple into memory.
- Will sort within blocks first, then merge the blocks into one long sorted order.

Sorting 56 blocks of 10M records

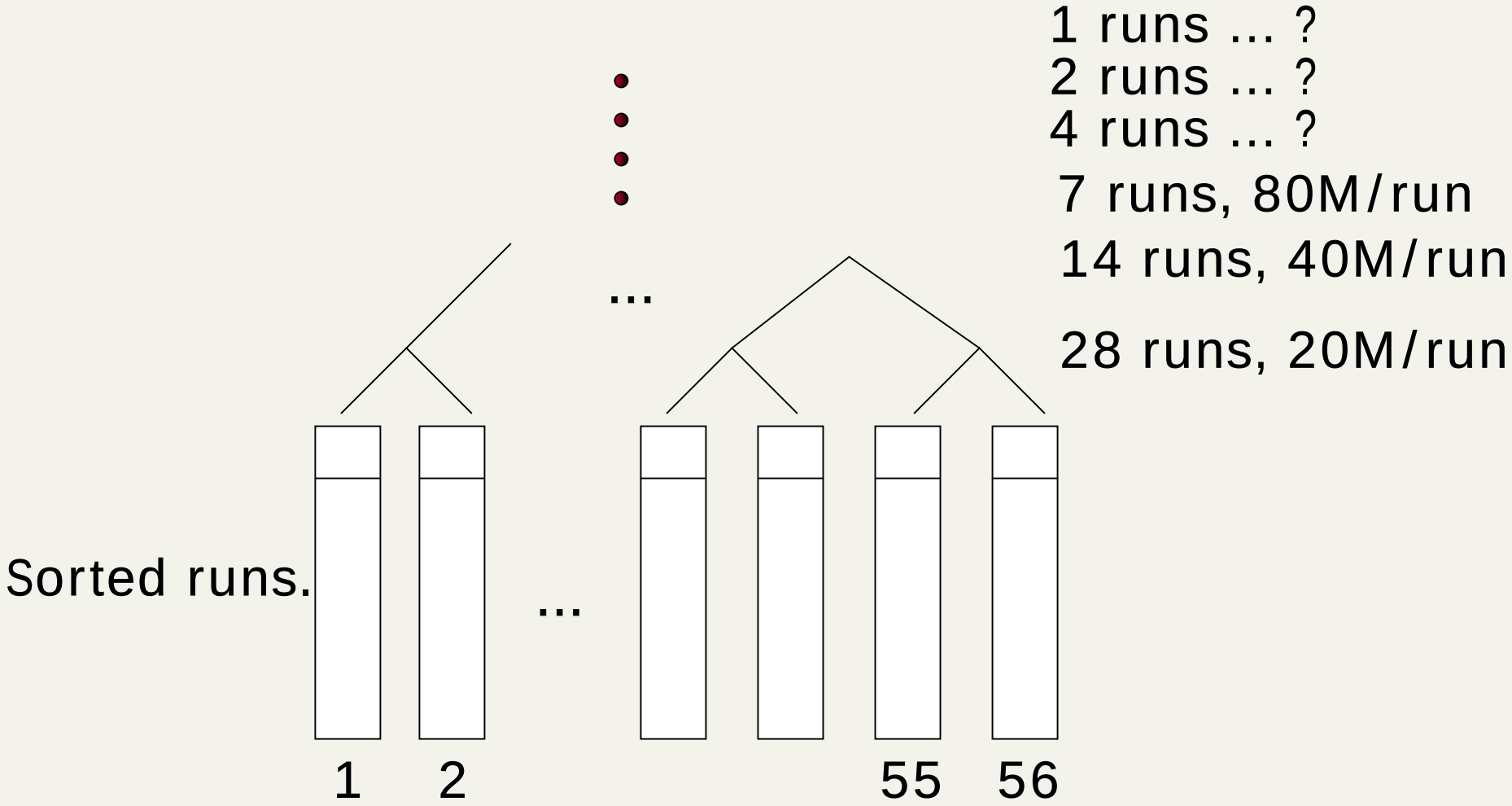
- First, read each block and sort within:
  - Quicksort takes about 2 x (10M ln 10M) steps
- *Exercise: estimate total time to read each block from disk and and quicksort it.*
- 56 times this estimate - gives us 56 sorted *runs* of 10M records each.
- Need 2 copies of data on disk, throughout.

Merging 56 sorted runs

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

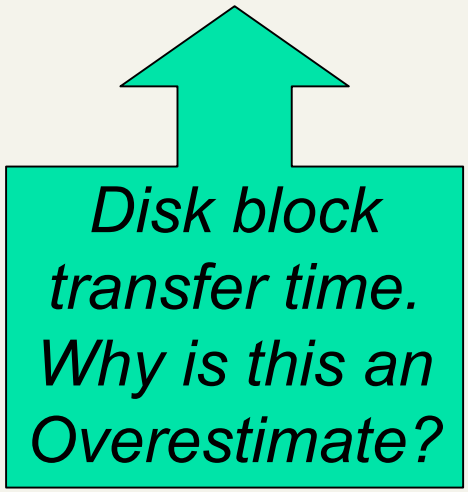


Merge tree


1 runs ... ?
2 runs ... ?
4 runs ... ?
7 runs, 80M/run
14 runs, 40M/run
28 runs, 20M/run
Sorted runs.
1
2
55
56

Merging 56 runs

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



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

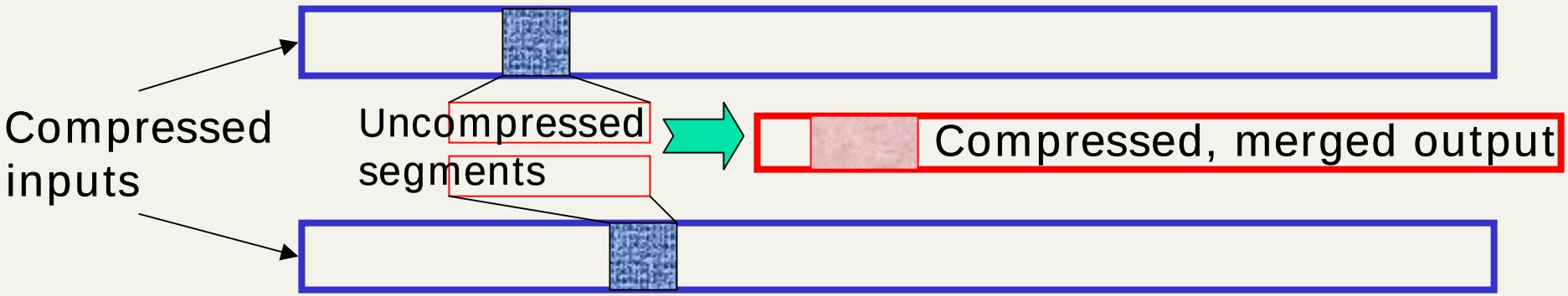
- Compressed temporary files
  - compress terms in temporary dictionary runs
- How do we merge compressed runs to generate a compressed run?
  - Given two γ-encoded runs, merge them into a new γ-encoded run
  - To do this, first γ-decode a run into a sequence of gaps, then actual records:
  - 33,14,107,5...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

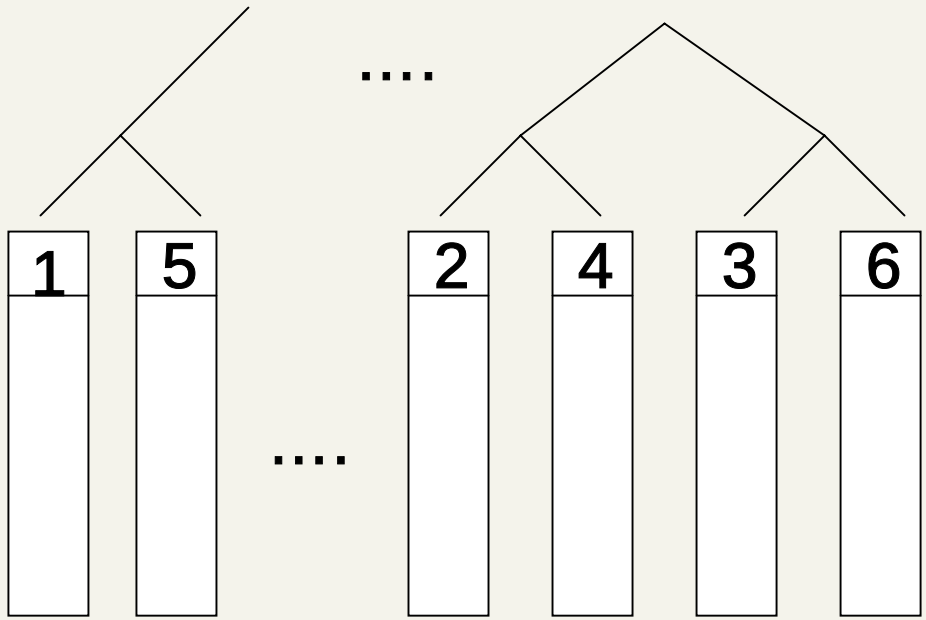
“Zipper” uncompress/decompress

- 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

More complex approach

- Fully dynamic updates
- Only one index at all times
  - No big and small indices
- Active management of a pool of space

Fully dynamic updates

- Inserting a (variable-length) record
  - e.g., a typical postings entry
- Maintain a pool of (say) 64KB *chunks*
- Chunk header maintains metadata on records in chunk, and its free space

Global tracking

- In memory, maintain a global record address table that says, for each record, the chunk it's in.
- Define one chunk to be current.
- Insertion
  - if current chunk has enough free space
    - extend record and update metadata.
  - else look in other chunks for enough space.
  - else open new chunk.

Changes to dictionary

- New terms appear over time
  - cannot use a static perfect hash for dictionary
- OK to use term character string w/pointers from postings as in lecture 2.

Index on disk vs. memory

- Most retrieval systems keep the dictionary in memory and the postings on disk
- Web search engines frequently keep both in memory
  - massive memory requirement
  - feasible for large web service installations, less so for standard usage where
    - query loads are lighter
    - users willing to wait 2 seconds for a response
- More on this when discussing deployment models

Distributed indexing

- Suppose we had several machines available to do the indexing
  - how do we exploit the parallelism
- Two basic approaches
  - stripe by dictionary as index is built up
  - stripe by documents

Indexing in the real world

- Typically, don’t have all documents sitting on a local filesystem
  - Documents need to be *spidered*
  - Could be dispersed over a WAN with varying connectivity
  - Must schedule distributed spiders/indexers
  - Could be (secure content) in
    - Databases
    - Content management applications
    - Email applications
- http often not the most efficient way of fetching these documents - native API fetching

Indexing in the real world

- Documents in a variety of formats
  - word processing formats (e.g., MS Word)
  - spreadsheets
  - presentations
  - publishing formats (e.g., pdf)
- Generally handled using format-specific “filters”
  - convert format into text + meta-data

Indexing in the real world

- Documents in a variety of languages
  - automatically detect language(s) in a document
  - tokenization, stemming, are language-dependent

“Rich” documents

- (How) Do we index images?
- Researchers have devised Query Based on Image Content (QBIC) systems
  - “show me a picture similar to this orange circle”
  - watch for next week’s lecture on vector space retrieval
- In practice, image search based on meta-data such as file name e.g., monalisa.jpg

Passage/sentence retrieval

- Suppose we want to retrieve not an entire document matching a query, but only a passage/sentence - say, in a very long document
- Can index passages/sentences as mini-documents
- But then you lose the overall relevance assessment from the complete document - more on this as we study relevance ranking
- More on this when discussing XML search

Resources, and beyond

- MG 5.
- Thus far, documents either match a query or do not.
- It’s time to become more discriminating - how well does a document match a query?
- Gives rise to ranking and scoring