CS3245

Information Retrieval

13

Revision

Last Week: Web Search

- Search Advertising
- Duplicate Detection
- Crawling

- Anchor Text
- PageRank

Today

- Exam Format

- Revision
  Revision
  and more Revision

- Where to go from here

Exam Format

Open Book

Topics
in order

1. This examination paper contains SIX (6) questions and comprises SEVEN (7) printed pages, including this page. Some questions have multiple parts.
2. It is suggested that you limit your response length to the space in the boxes provided.
3. You may use the backs of the pages as scratch paper, as they *will* be disregarded, unless specifically noted by you.
4. This is an OPEN BOOK examination. You may consult books and any other printed or handwritten materials for this test.
5. You may use pencil or other erasable medium in answering this paper.
6. The questions are presented in order by which their topic is covered in the syllabus, not by their perceived difficulty or estimated time to answer. You may want to do the questions out of order.
7. Please write your Matriculation Number below. Do not write your name.

MATRICULATION NO: ______________________________

This portion is for examiner's use only

Question	Q1	Q2	Q3	Q4	Q5	Q6	Total
Max	**20**	**18**	**17**	**12**	**15**	**18**	100
Marks							

Exam Topics

- Anything covered in lecture (through slides)
- And corresponding sections in textbook
- Tutorial and homework essays are the exam models
  - Often I wrote the exam questions when writing tutorials
  - There are no past exams for this course (new course)
  - But there is a graduate level module that I've taught before (CS 5246; I taught it in 2007)
- Not responsible for sections that we didn't cover
  - If in doubt, ask on the forum

Exam Topic Distribution

- Emphasize on second half of semester, but must skip some topics
- About ½ are calculation questions, on crucial topics on ones skipped in tutorial or homework
  - Time consuming but easy and straightforward
  - Don't forget your calculator!
- Others are thinking essay questions (cf. tutorials)

Date	
W1 (13 Jan)	Orientation and Language Models
W2 (20 Jan)	Boolean Retrieval
W3 (27 Jan)	Terms and Postings
W4 (3 Feb)	Chinese New Year Holiday -- Watch recor Dictionaries and Tolerant Retrieval
W5 (10 Feb)	Index Construction
W6 (17 Feb)	Index Compression
W7 (3 Mar)	Scoring and the Vector Space Model
W8 (10 Mar)	A Complete Search System
W9 (17 Mar)	IR Evaluation
W10 (24 Mar)	XML Retrieval
W11 (31 Mar)	Probabilistic IR
W12 (7 Apr)	Web Search
W13 (14 Apr)	Revision

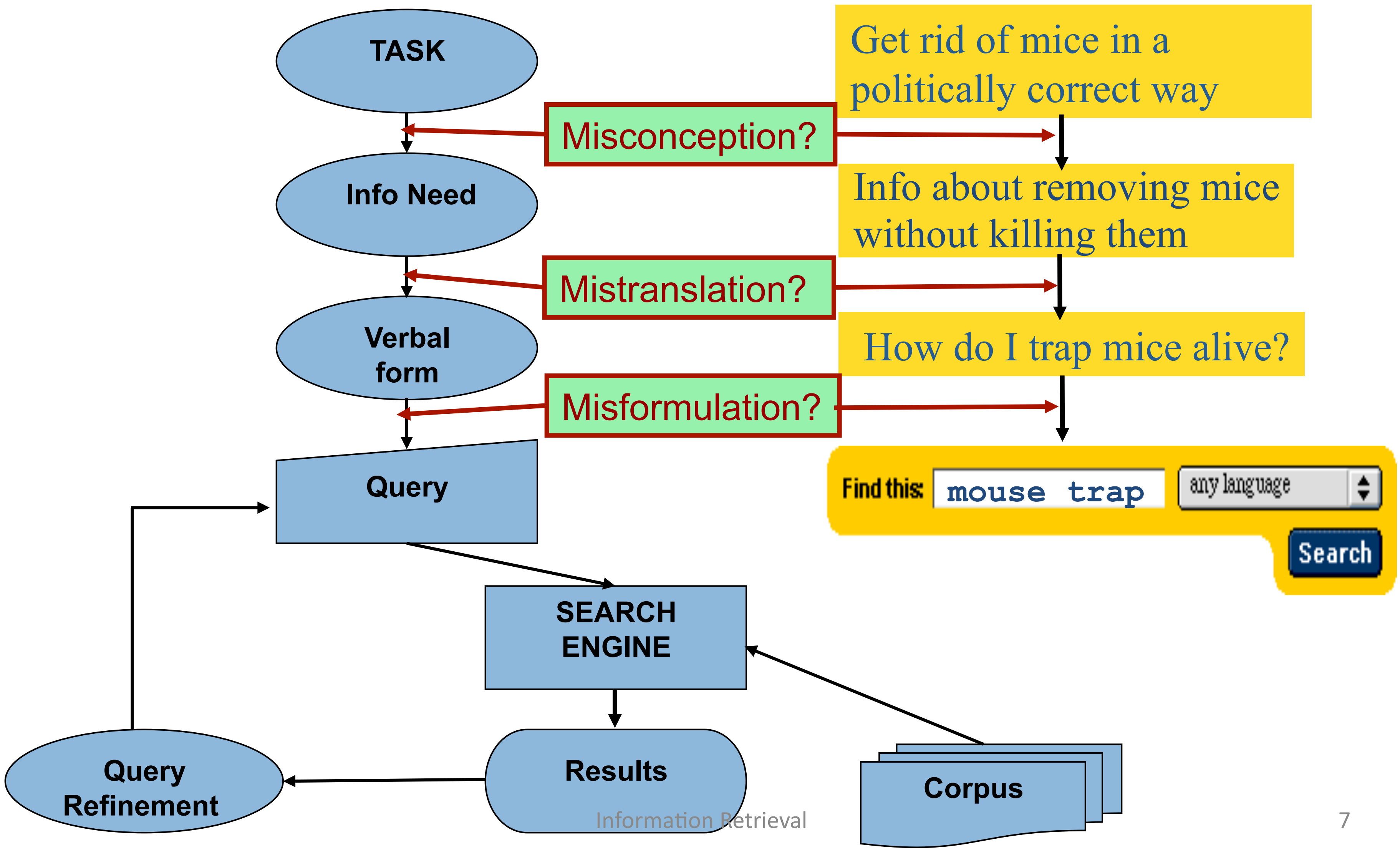
1st half: 1st 20pt question

2nd half: 5 questions (80pts total)

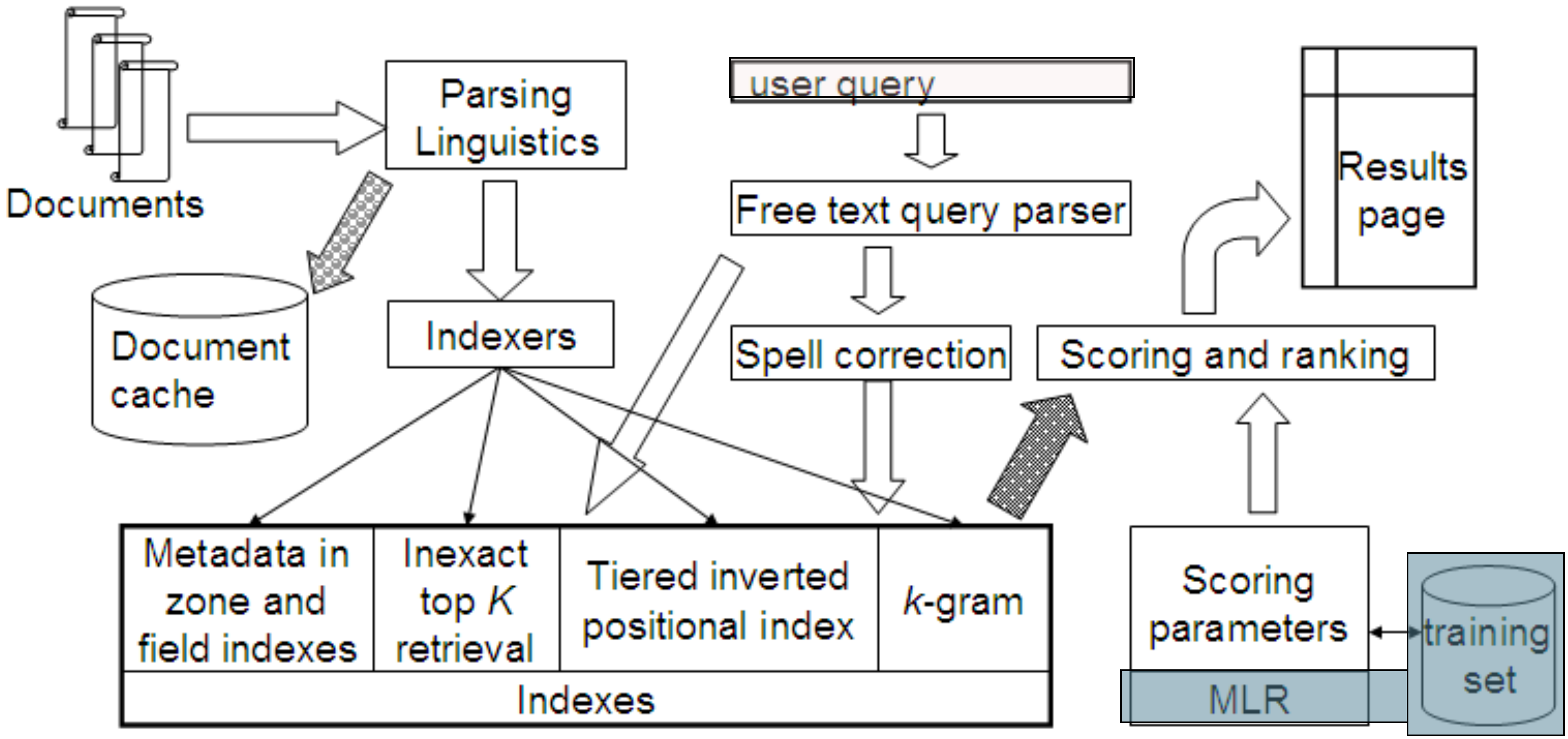
No homework topic

No tutorial topic

Understanding the user: The classic search model

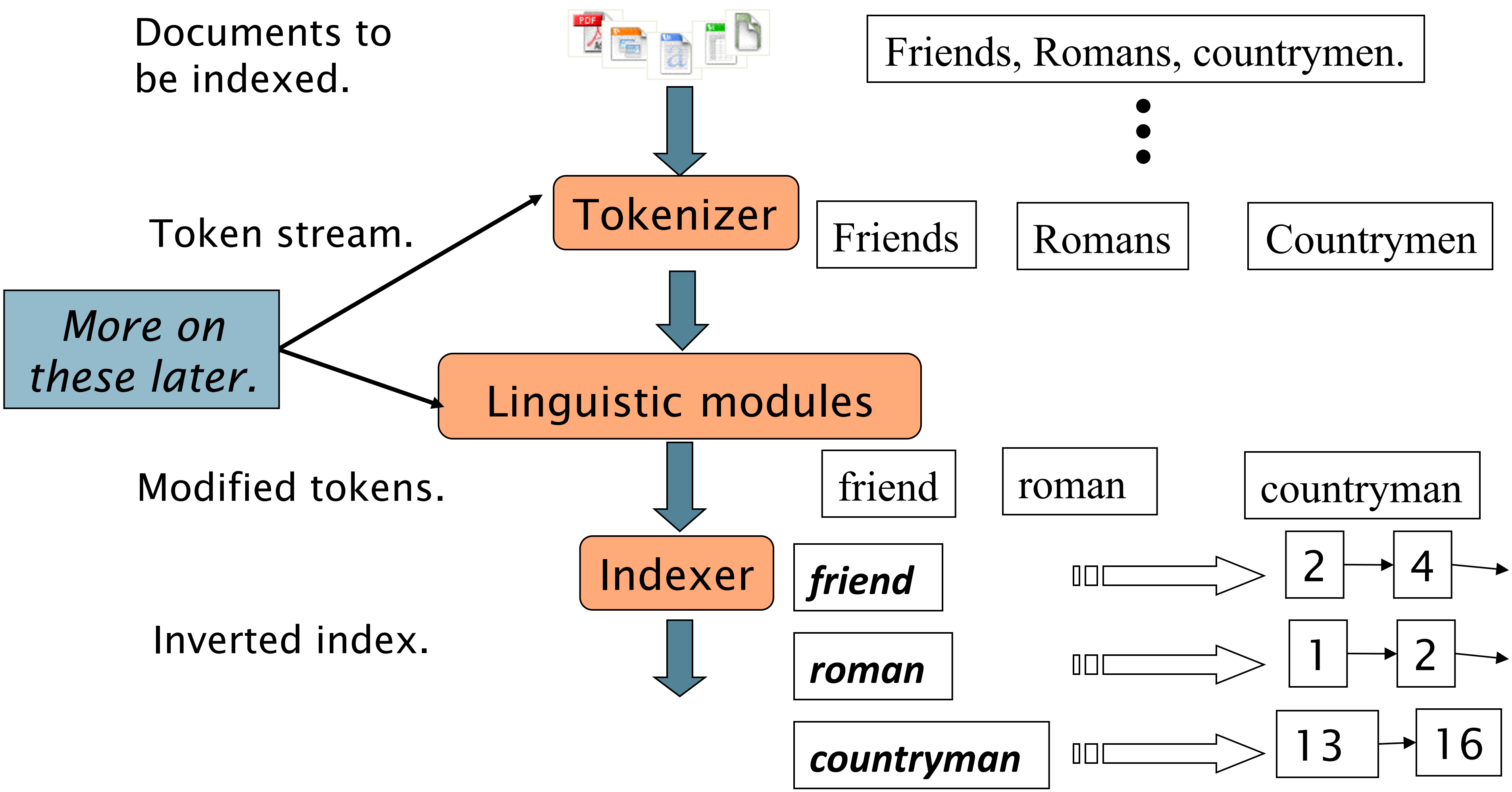


The IR System

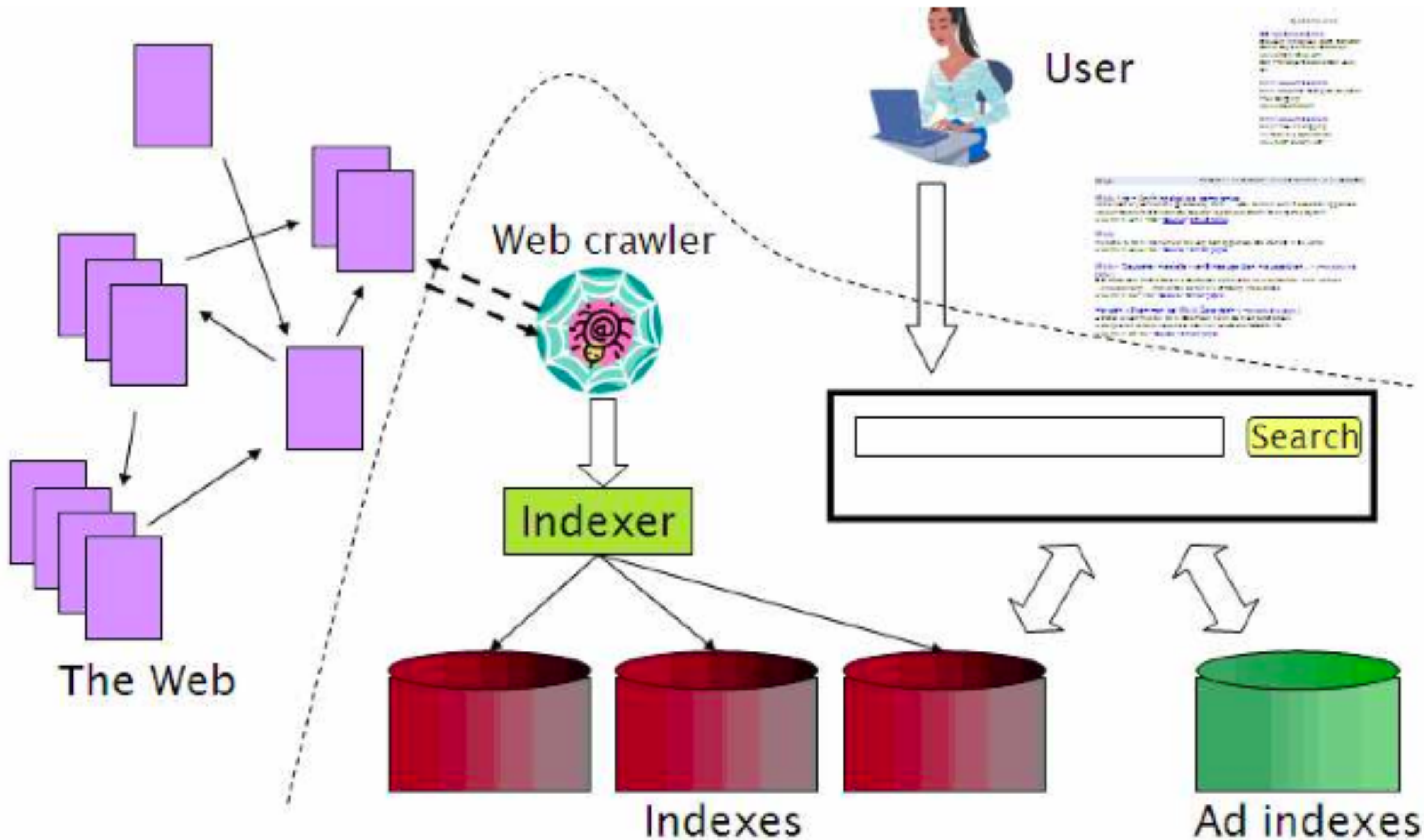


Won't be covering these blue modules in this course

Zoom in: Index Construction



Zoom Out: Web search



REVISION

Week 1: Ngram Models

- Unigram LM: Bag of words
- Ngram LM: use n-1 tokens of context to predict n^{th} token
- Larger n-gram models more accurate but each increase in order requires exponentially more space

Your turn: what do you think? Can we use a LM to do information retrieval?

You bet. We'll return to this in probabilistic information retrieval.

The Unigram Model

- View language as a unordered collection of tokens
  - Each of the n tokens contributes one count (or 1/n) to the model
  - Also known as a “bag of words”
- Outputs a count (or probability) of an input based on its individual token
  - $\text{Count(input)} = \Sigma_n \text{ Count (n)}$
  - $\text{P(input)} = \pi_n \text{ P(n)}$

Add 1 smoothing

- Not used in practice, but most basic to understand
- Idea: add 1 count to all entries in the LM, including those that are not seen

e.g., assuming |V| = 11

Total # of word types in both lines

I	2	eyes	2
don't	2	your	1
want	2	love	1
to	2	and	1
close	2	revenge	1
my	2	**Total Count**	18

I	3	eyes	1
don't	1	your	3
want	3	love	2
to	1	and	2
close	1	revenge	2
my	1	**Total Count**	20

- Q2 (By **Probability**) : "I don't want"

P(Aerosmith): .11 * .11 * .11 = 1.3E-3

P(LadyGaga): .15 * .05 * .15 = 1.1E-3

Winner: Aerosmith

Week 2: Basic (Boolean) IR

- Basic inverted indexes:
  - In memory dictionary and on disk postings

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

  - Key characteristic: Sorted order for postings
- Boolean query processing
  - Intersection by linear time “merging”
  - Simple optimizations by expected size

Term-document incidence

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Brutus* AND *Caesar* BUT NOT *Calpurnia

1 if play contains word, 0 otherwise

Indexer steps: Dictionary & Postings

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

Why frequency?

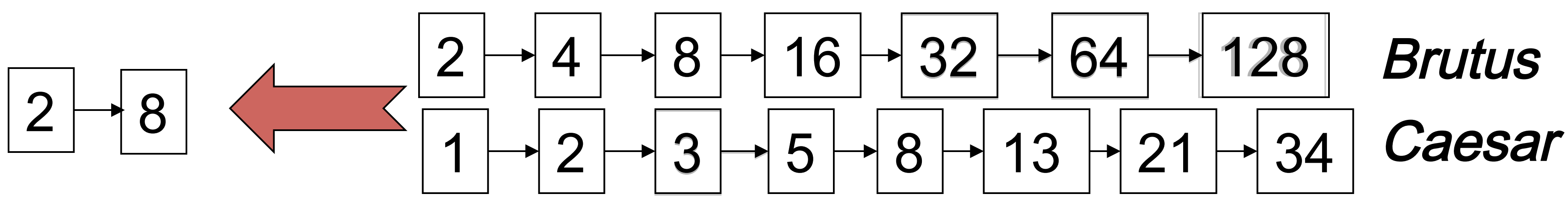
Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

→

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

The merge

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



If the list lengths are *x* and *y*, the merge takes O(*x*+*y*) operations.
Crucial: postings must be sorted by docID.

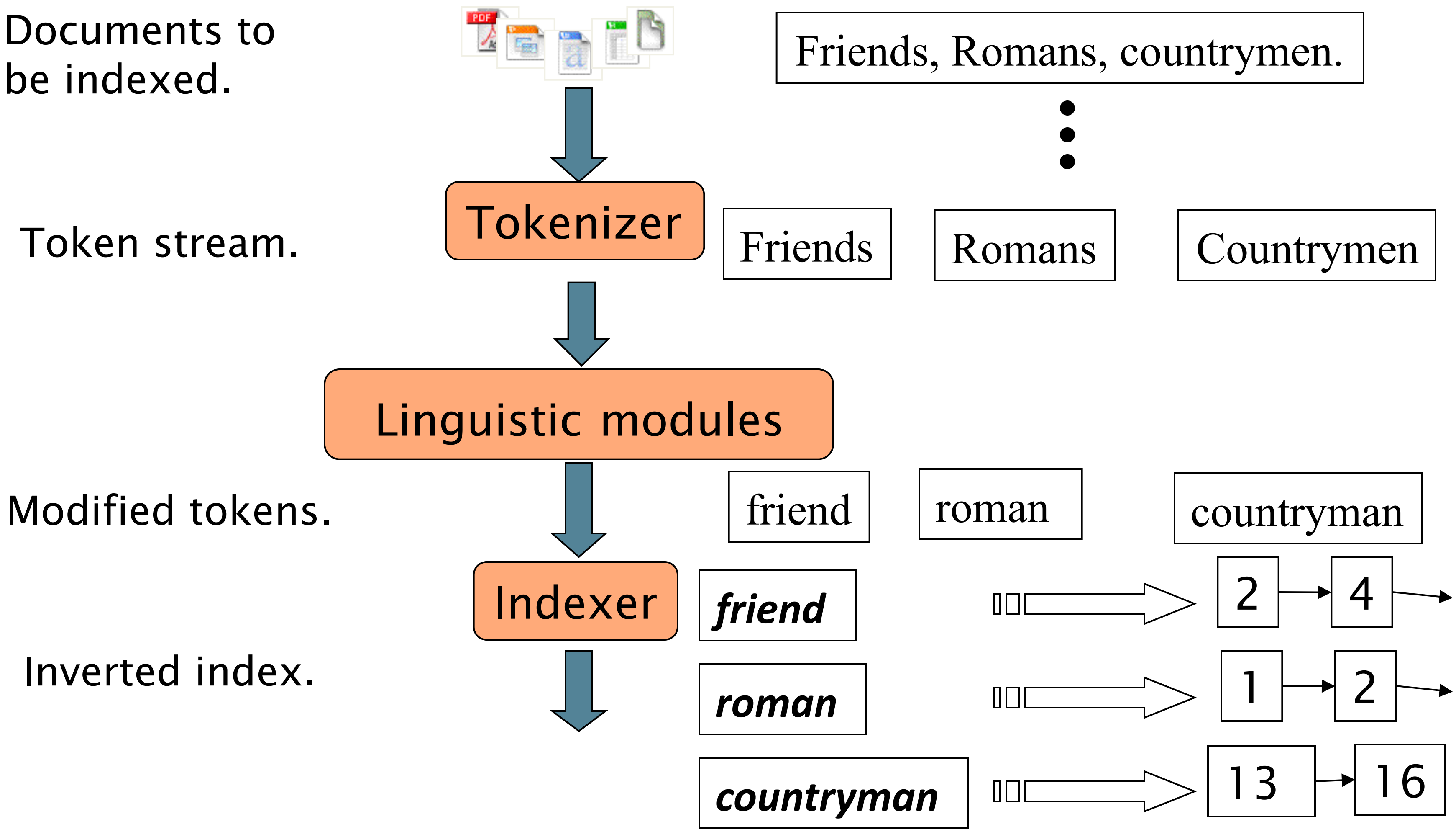
Week 3: Terms and Postings Details

- The type/token distinction
  - Terms are normalized types put in the dictionary
- Tokenization problems
  - Hyphens, apostrophes, spaces, compounds
  - Language specific problems
- Term equivalence classing (or not)
  - Numbers, case folding, stemming, lemmatization

- Skip pointers
  - Encoding a tree-like structure in a postings list

- Biword indexes for phrases
- Positional indexes for phrases/proximity queries

Inverted index construction

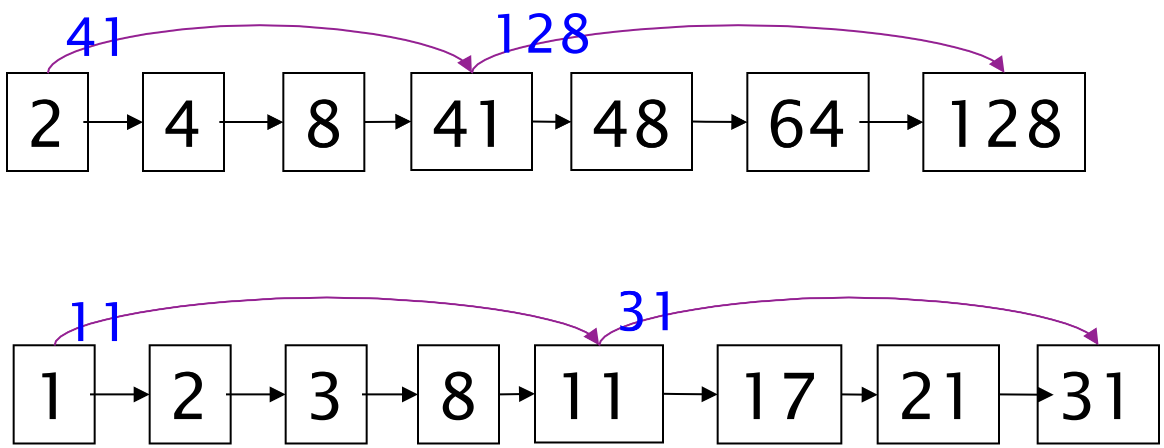


Tokenization and Normalization

- Definitely language specific

- In English, we worry about
  - Tokenization – Spaces and Punctuation
  - Case folding
  - Stopwording
  - Normalization – Stemming or Lemmatization

Adding skip pointers to postings



- Done at indexing time.
- Why?
- How to do it? And where do we place skip pointers?

A first attempt at phrasal queries: Biword indexes

- Index every consecutive pair of terms in the text as a phrase: bigram model using words
- For example the text "Friends, Romans, Countrymen" would generate the biwords
  - ***friends romans***
  - ***romans countrymen***
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.

Positional index example

<***be***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

Quick check:
Which of docs 1,2,4,5
could contain “***to be or not to be***”?

- For phrase queries, we use a merge algorithm recursively at the document level
- Now need to deal with more than just equality

Week 4: The dictionary and tolerant retrieval

- Data Structures for the Dictionary
  - Hash
  - Trees

- Learning to be tolerant

1. Wildcards
   - General Trees
   - Permuterm
   - Ngrams, redux
2. Spelling Correction
   - Edit Distance
   - Ngrams, re-redux
3. Phonetic – Soundex

Hash Table

Each vocabulary term is hashed to an integer

- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Not very tolerant!

B-trees handle *'s at the end of a query term

- How can we handle *'s in the middle of query term?
  - ***co*tion***
- We could look up ***co**** AND ****tion*** in a B-tree and intersect the two term sets
  - Expensive
- The solution: transform wild-card queries so that the *'s always occur at the end
- This gives rise to the **Permuterm** Index.

Permuterm index

- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell***

  where **$** is a special symbol.
- Queries:
  - **X** lookup on **X$**
  - **X*** lookup on $**X***
  - ***X** lookup on **X$***
  - ***X*** lookup on **X***
  - **X*Y** lookup on **Y$X***

Query = hel*o
X=hel, Y=o
Lookup o$hel*

Not so quick Q:
What about X*Y*Z?

Isolated word spelling correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- How do we define "closest"?
- We'll study several alternatives
  1. Edit distance (Levenshtein distance)
  2. Weighted edit distance
  3. *n*gram overlap

Week 5: Index construction

- Sort-based indexing
  - Blocked Sort-Based Indexing
    - Merge sort is effective for disk-based sorting (avoid seeks!)
  - Single-Pass In-Memory Indexing
    - No global dictionary - Generate separate dictionary for each block
    - Don’t sort postings - Accumulate postings as they occur
- Distributed indexing using MapReduce
- Dynamic indexing: Multiple indices, logarithmic merge

Hardware basics

Many design decisions in information retrieval are based on the characteristics of hardware

Especially with respect to the bottleneck:

Hard Drive Storage

- Seek Time – time to move to a random location
- Transfer Time – time to transfer a data block

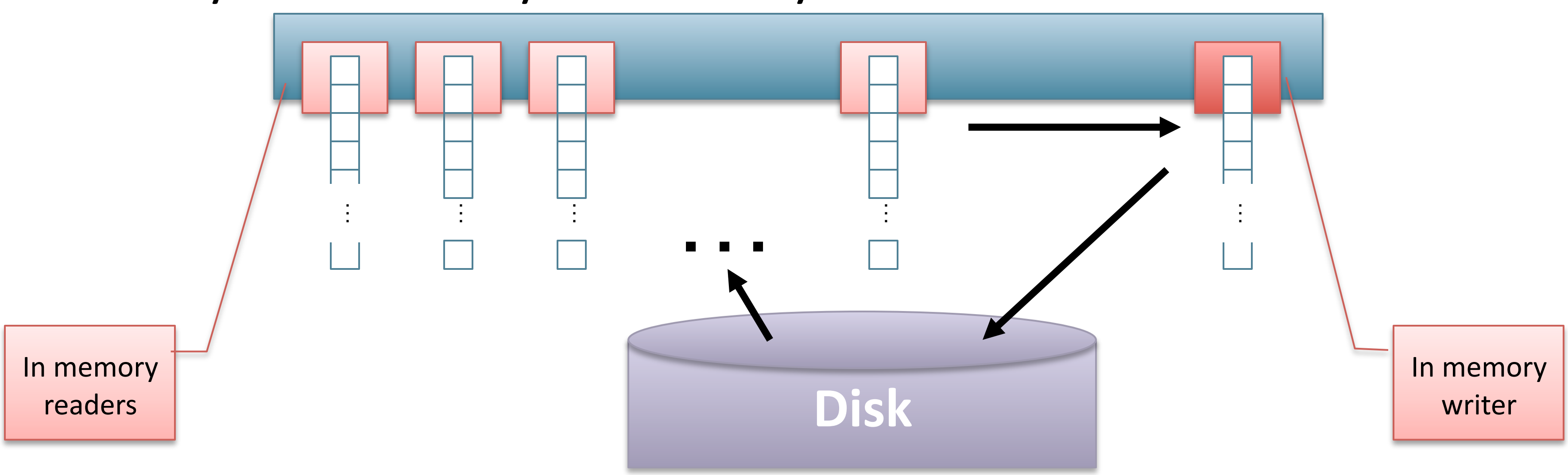
BSBI: Blocked sort-based Indexing (Sorting with fewer disk seeks)

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

How to merge the sorted runs?

Second method (better):

- It is more efficient to do a *n*-way 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 your efficiency isn't lost by disk seeks

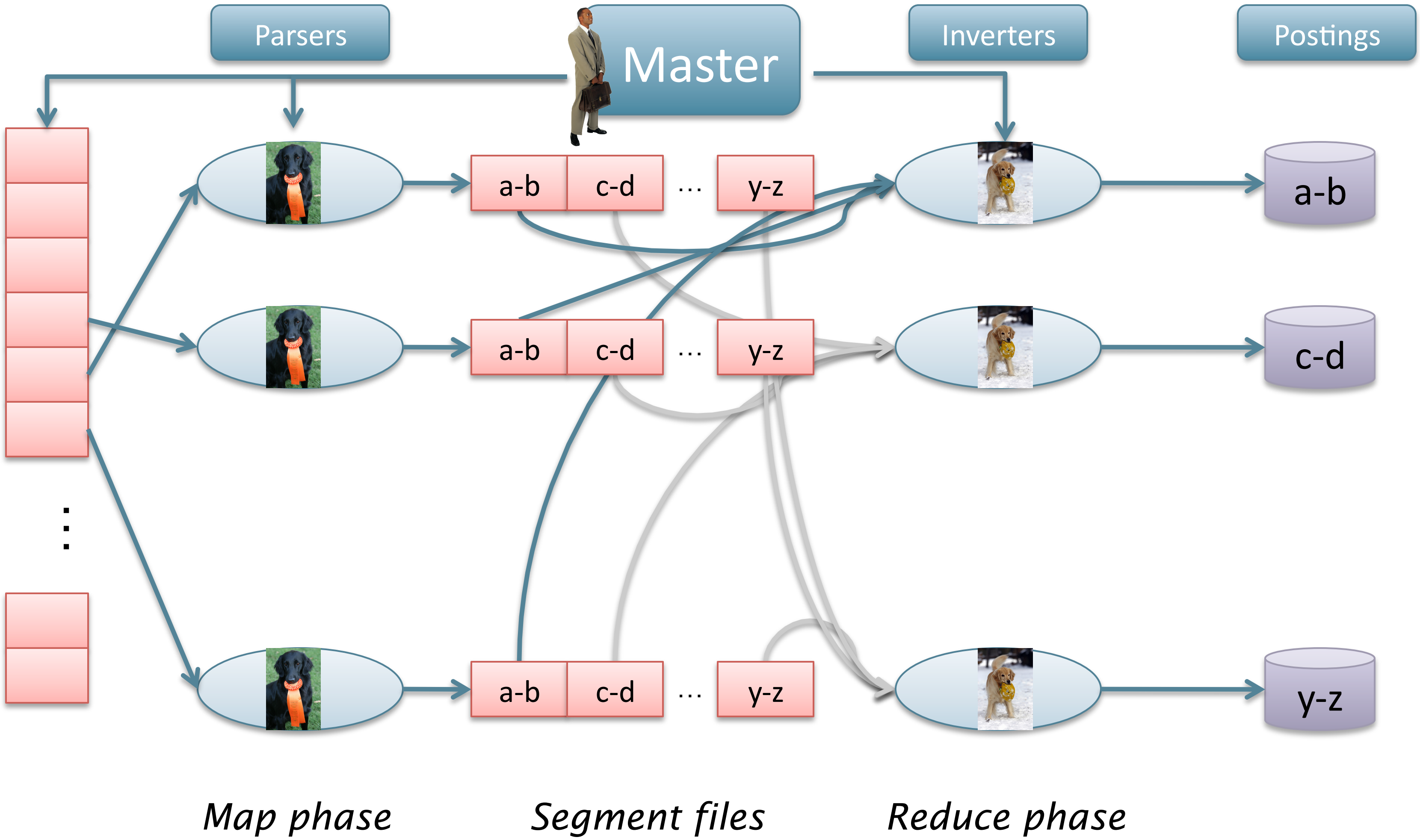


SPIMI: Single-pass in-memory indexing

- **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 indices can then be merged into one big index.

Distributed Indexing: MapReduce Data flow



Dynamic Indexing: 2nd simplest approach

- Maintain "big" main index
- New docs go into "small" (in memory) 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
  - Assuming T total # of postings and n as size of auxiliary index, we touch each posting up to *floor(T/n)* times.

Logarithmic merge

- Idea: maintain a series of indexes, each twice as large as the previous one.
- 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) as Z_1
- Either write merge Z_1 to disk as I_1 (if no I_1) Or merge with I_1 to form Z_2

… etc.

Loop for log levels

Week 6: Index Compression

- Collection and vocabulary statistics: Heaps' and Zipf's laws

Compression to make index smaller, faster

- Dictionary compression for Boolean indexes
  - Dictionary string, blocks, front coding
- Postings compression: Gap encoding

Empirical Laws

Heaps' law: $M = kT^b$

- M is the size of the vocabulary, T is the number of tokens in the collection
- 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

Zipf's law: $cf_i \propto 1/i = K/i$

- Zipf's law: The ith most frequent term has frequency proportional to $1/i$.
- where K is a normalizing constant
- cf_i is collection frequency (not document frequency): the number of occurrences of the term t_i in the collection.

Index Compression: Dictionary-as-a-String and 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*** …

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

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

Postings Compression: 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.

Variable Byte Encoding: Example

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

512+256+32+16+8 = 824

Postings stored as the byte concatenation
00000110 10111000 10000101 00001101 00001100 10110001

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

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

Week 7: Vector space ranking

1. Represent the query as a weighted tf-idf vector
2. Represent each document as a weighted tf-idf vector
3. Compute the cosine similarity score for the query vector and each document vector
4. Rank documents with respect to the query by score
5. Return the top *K* (e.g., *K* = 10) to the user

Term-document count matrices

- Store the number of occurrences of a term in a document:
  - Each document is a **count vector** in $\mathbb{N}^{v}$: a column below

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$\mathrm{w}_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \times \log_{10}(N / \mathrm{df}_t)$$

- **Best known weighting scheme IR**
  - Note: the "-" in tf-idf is a hyphen, not a minus sign!
  - Alternative names: tf.idf, tf x idf
- **Increases** with the number of occurrences within a document
- **Increases** with the rarity of the term in the collection

Queries as vectors

- Key idea 1: Do the same for queries: represent them as vectors in the space; they are "mini-documents"
- Key idea 2: Rank documents according to their proximity to the query in this space

- proximity = similarity of vectors
- proximity ≈ inverse of distance
- Motivation: Want to get away from the you're-either-in-or-out Boolean model.
- Instead: rank more relevant documents higher than less relevant documents

Length normalization

- A vector can be (length-) normalized by dividing each of its components by its length – for this we use the L_2 norm:

$$\left\|\vec{x}\right\|_2 = \sqrt{\sum_i x_i^2}$$

- Dividing a vector by its L_2 norm makes it a unit (length) vector (on surface of unit hypersphere)
- Effect on the two documents d and d' (d appended to itself) from earlier slide: they have identical vectors after length normalization.
  - Long and short documents now have comparable weights

Cosine for length-normalized vectors

- For length-normalized vectors, cosine similarity is simply the dot product (or scalar product):

$$\cos(\vec{q},\vec{d}) = \vec{q} \bullet \vec{d} = \sum_{i=1}^{|V|} q_i d_i$$

for q, d length-normalized.

Week 8: Hacking IR in practice

Making the Vector Space Model more efficient to compute

- Approximating the actual correct results
- Skipping unnecessary documents

In actual data: dealing with zones and fields, query term proximity

Resources for today

- IIR 7, 6.1

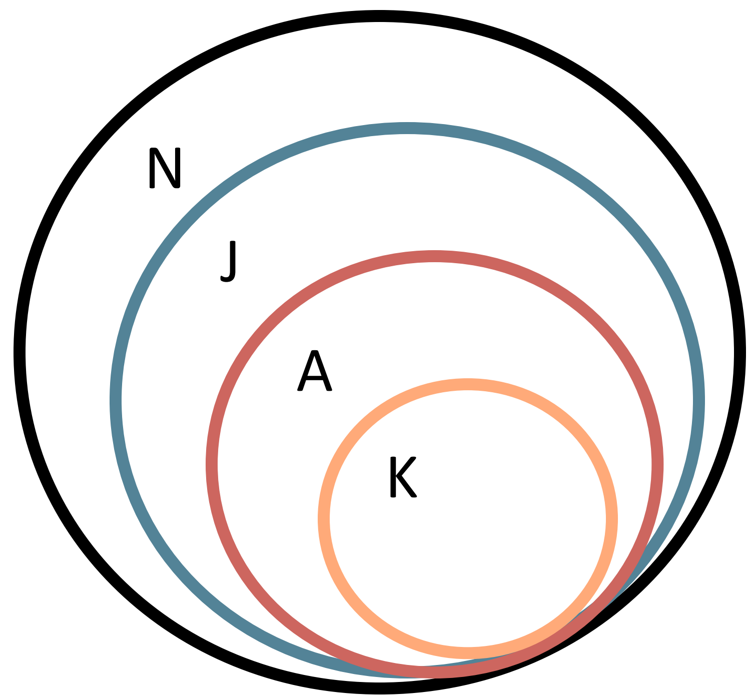
Recap: Computing cosine scores

COSINESCORE(q)

float Scores$[N] = 0$
float Length$[N]$
for each query term t
do calculate $w_{t,q}$ and fetch postings list for t
  for each pair(d, $\text{tf}_{t,d}$) in postings list
  $\boxed{\textbf{do } Scores[d] += w_{t,d} \times w_{t,q}}$
Read the array *Length*
for each d
do $Scores[d] = Scores[d]/Length[d]$
return Top K components of $Scores[]$

Generic approach

- Find a set *A* of *contenders*, with $K < |A| \ll N$
  - *A* does not necessarily contain the top *K*, but has many docs from among the top *K*
  - Return the top *K* docs in *A*
- Think of *A* as <u>pruning</u> non-contenders
- The same approach can also used for other (non-cosine) scoring functions



Net score

- Consider a simple total score combining cosine relevance and authority

$$\text{net-score}(q,d) = g(d) + \text{cosine}(q,d)$$

  - Can use some other linear combination than an equal weighting
  - Indeed, any function of the two "signals" of user happiness

- Now we seek the top *K* docs by <u>net score</u>

Parametric Indices

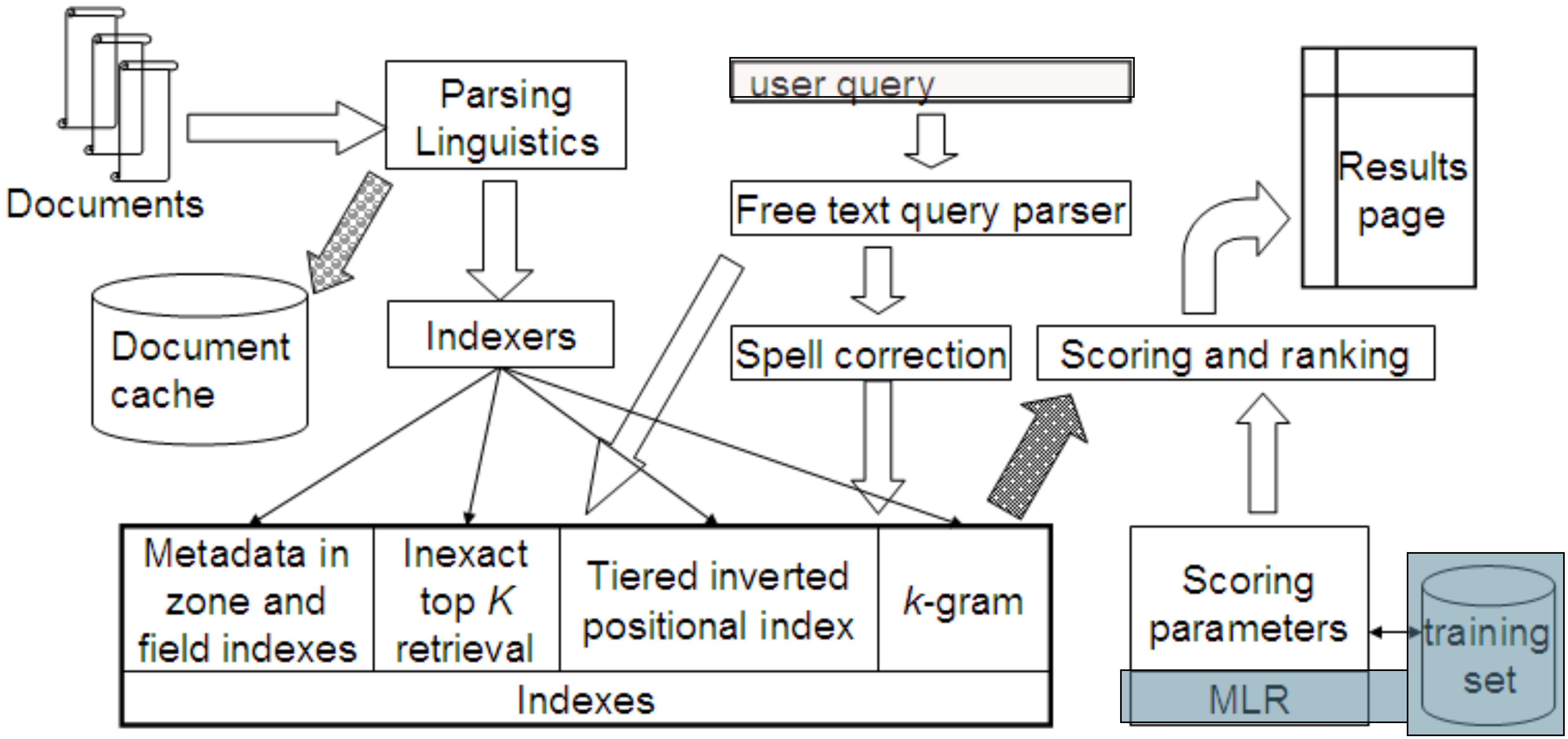
Fields

- Year = 1601 is an example of a field
- Field or parametric index: postings for each field value
  - Sometimes build range (B-tree) trees (e.g., for dates)
- Field query typically treated as conjunction
  - (doc *must* be authored by shakespeare)

Zone

- A zone is a region of the doc that can contain an arbitrary amount of text e.g.,
  - Title
  - Abstract
  - References …
- Build inverted indexes on zones as well to permit querying

Putting it all together



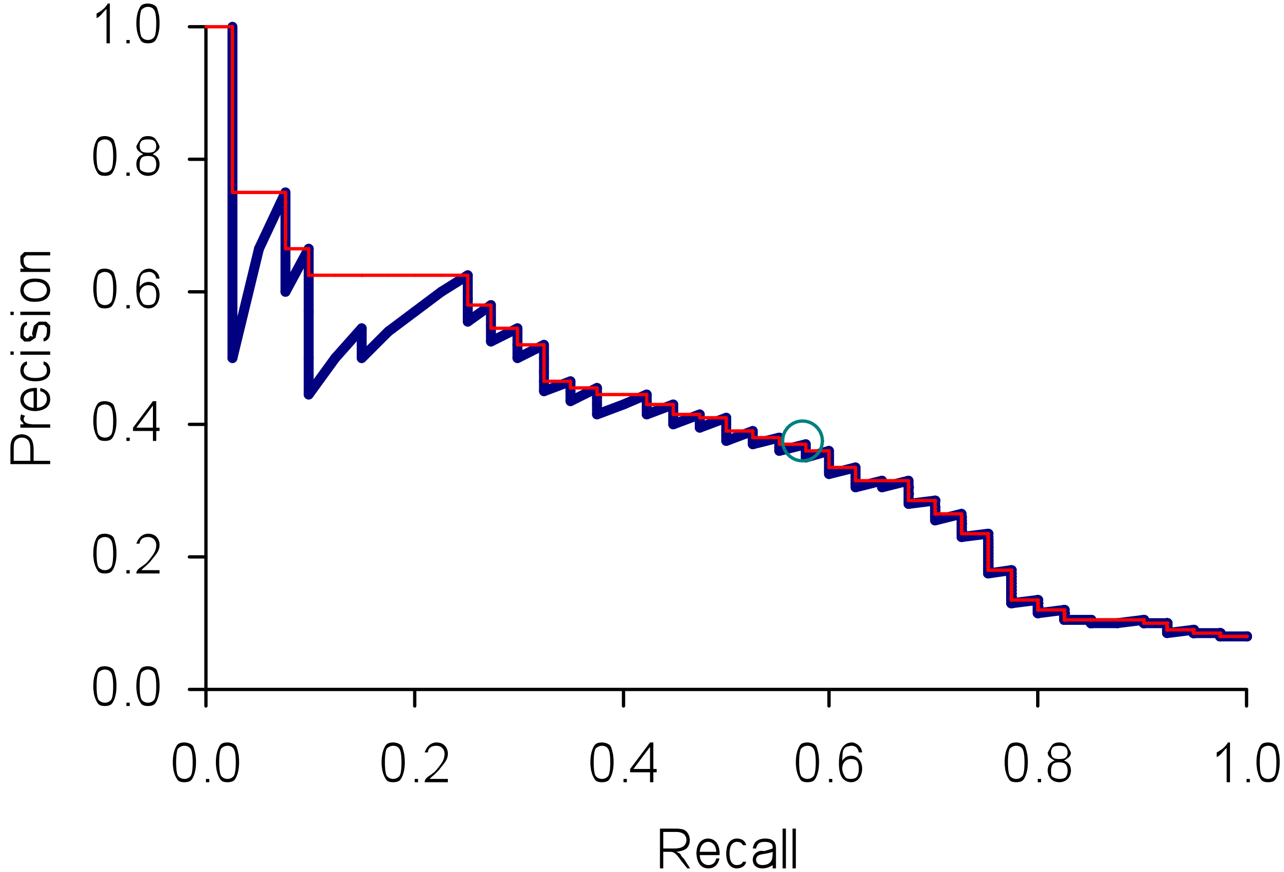
Won't be covering these blue modules in this course

Week 9: IR Evaluation

- How do we know if our results are any good?
  - Evaluating a search engine
    - Benchmarks
    - Precision and Recall; Composite measures

- Results summaries
  - Making our results usable

A precision-recall curve



Kappa Example

$P(A) = 370/400 = 0.925$

P(nonrelevant) = (10+20+70+70)/800 = 0.2125

P(relevant) = (10+20+300+300)/800 = 0.7878

$P(E) = 0.2125^2 + 0.7878^2 = 0.665$

Kappa = (0.925 – 0.665)/(1-0.665) = 0.776

- Kappa > 0.8 → good agreement
- 0.67 < Kappa < 0.8 → "tentative conclusions"

- Depends on purpose of study
- For >2 judges: average pairwise kappas

Evaluation at large search engines

- Search engines have test collections of queries and hand-ranked results
- Recall is difficult to measure on the web
- Search engines often use precision at top *k* (e.g., *k* = 10)
- . . . or measures that reward you more for getting rank 1 right than for getting rank 10 right.
  - NDCG (Normalized Cumulative Discounted Gain)
  - MRR (Mean Reciprocal Rank)
- Search engines also use non-relevance-based measures.
  - Clickthrough on first result
    - Not very reliable if you look at a single clickthrough … but pretty reliable in the aggregate.
  - Studies of user behavior in the lab
  - A/B testing

A/B testing

Purpose: Test a single innovation

Prerequisite: You have a large search engine up and running.

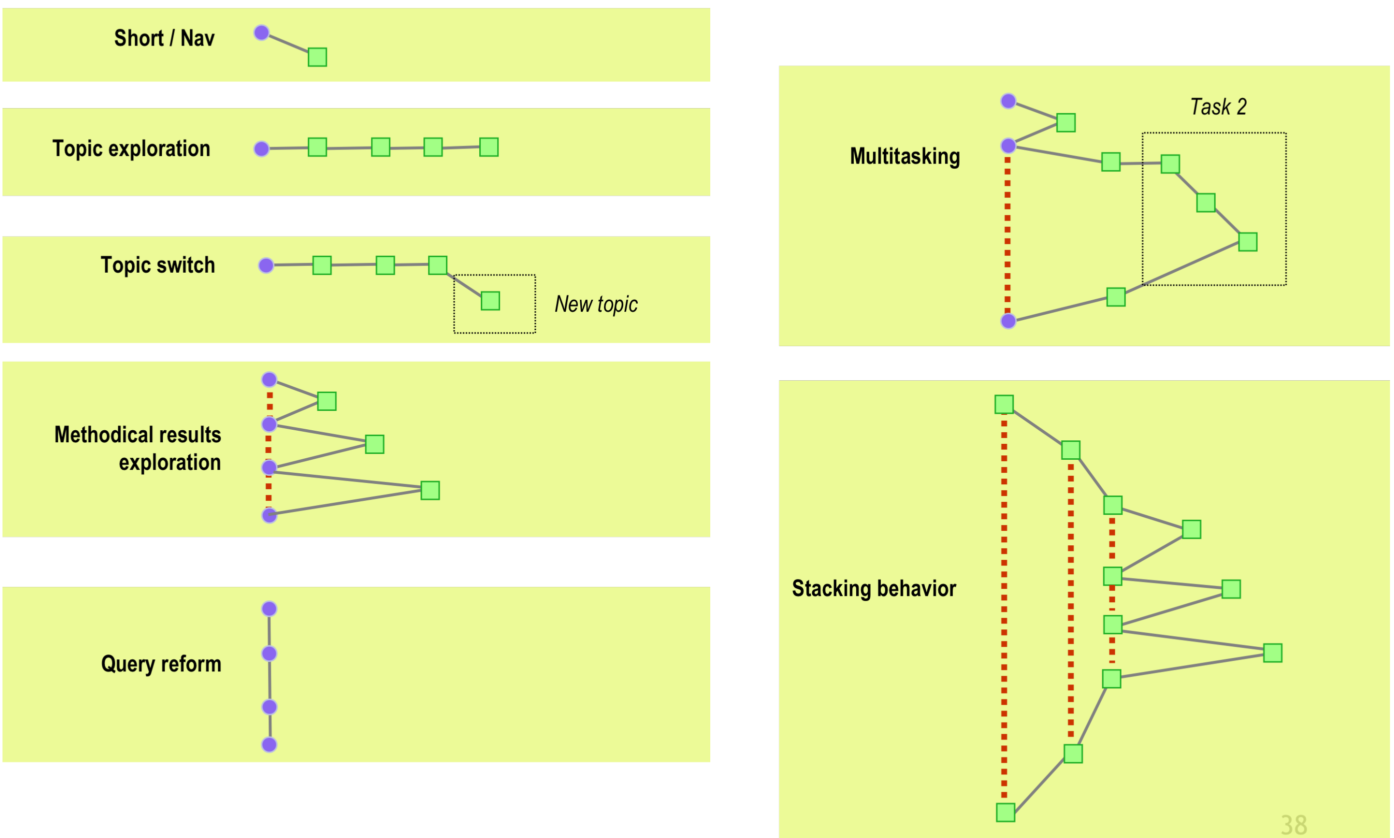
- Have most users use old system
- Divert a small proportion of traffic (e.g., 1%) to the new system that includes the innovation
- Evaluate with an "automatic" overall evaluation criterion (OEC) like clickthrough on first result
- Now we can directly see if the innovation does improve user happiness.
- Probably the evaluation methodology that large search engines trust most
- In principle less powerful than doing a multivariate regression analysis, but easier to understand

Dynamic summaries

- Present one or more "windows" within the document that contain several of the query terms
  - One of the killer features of Google (ca. 1996)
  - "KWIC" snippets: Keyword in Context presentation

Slide courtesy Google Inc.

Kinds of behaviors we see in the data



Google

Week 10: XML Retrieval

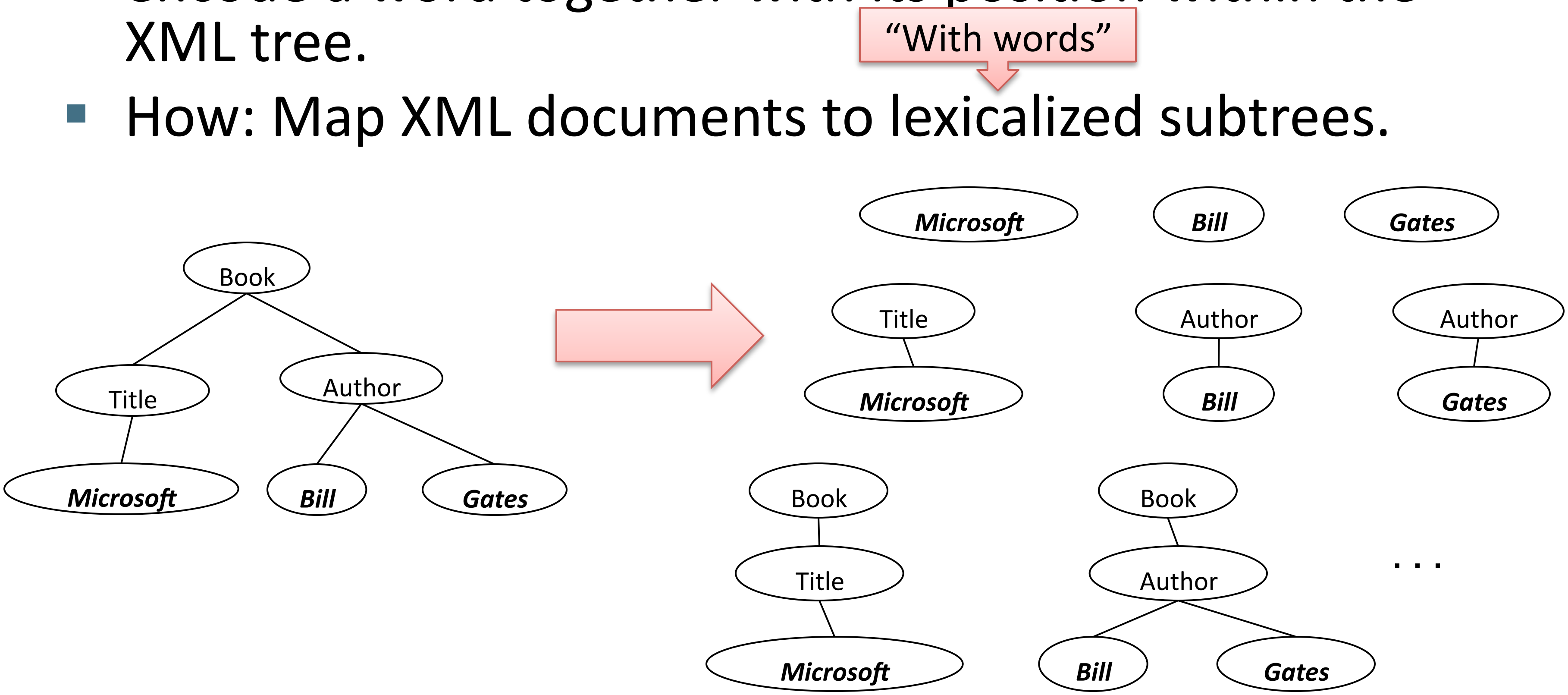
- Introduction
- Basic XML concepts
- Challenges in XML IR
- Vector space model for XML IR
- Evaluation of XML IR

XML Basics and Definitions

- **XML Document Object Model (XML DOM)**: standard for accessing and processing XML documents
  - The DOM represents elements, attributes and text within elements as nodes in a tree.
  - With a DOM API, we can process an XML documents by starting at the root element and then descending down the tree from parents to children.
- **XPath**: standard for enumerating path in an XML document collection.
  - We will also refer to paths as **XML contexts** or simply **contexts**
- **Schema**: puts constraints on the structure of allowable XML documents. E.g. a schema for Shakespeare's plays: scenes can occur as children of acts.
  - Two standards for schemas for XML documents are: XML DTD (document type definition) and XML Schema.

Main idea: lexicalized subtrees

- Aim: to have each dimension of the vector space encode a word together with its position within the XML tree.
- How: Map XML documents to lexicalized subtrees.



Context resemblance

- A simple measure of the similarity of a path c_q in a query and a path c_q in a document is the following ***context resemblance*** function CR:

$$\mathrm{CR}(c_q, c_d) = \begin{cases} \frac{1+|c_q|}{1+|c_d|} & \text{if } c_q \text{ matches } c_d \\ 0 & \text{if } c_q \text{ does not match } c_d \end{cases}$$

  $|c_q|$ and $|c_d|$ are the number of nodes in the query path and document path, respectively

- c_q matches c_d **iff** we can transform c_q into c_d by inserting additional nodes.

INEX relevance assessments

- The relevance-coverage combinations are quantized as follows:

$$\mathbf{Q}(rel, cov) = \begin{cases} 1.00 & \text{if} \quad (rel, cov) = 3\text{E} \\ 0.75 & \text{if} \quad (rel, cov) \in \{2\text{E}, 3\text{L}\} \\ 0.50 & \text{if} \quad (rel, cov) \in \{1\text{E}, 2\text{L}, 2\text{S}\} \\ 0.25 & \text{if} \quad (rel, cov) \in \{1\text{S}, 1\text{L}\} \\ 0.00 & \text{if} \quad (rel, cov) = 0\text{N} \end{cases}$$

- This evaluation scheme takes account of the fact that binary relevance judgments are not appropriate for XML retrieval. The quantization function **Q** instead allows us to grade each component as partially relevant. The number of relevant components in a retrieved set *A* of components can then be computed as:

$$\#(\text{relevant items retrieved}) = \sum_{c \in A} \mathbf{Q}(rel(c), cov(c))$$

Week 11: Probabilistic IR

Chapter 11

1. Probabilistic Approach to Retrieval / Basic Probability Theory
2. Probability Ranking Principle
3. OKAPI BM25

Chapter 12

1. Language Models for IR

Binary Independence Model (BIM)

- Traditionally used with the PRP

Assumptions:

- **Binary** (equivalent to Boolean): documents and queries represented as binary term incidence vectors
  - E.g., document *d* represented by vector $\vec{x} = (x_1, \ldots, x_M)$, where
  - $x_t = 1$ if term *t* occurs in *d* and $x_t = 0$ otherwise
  - Different documents may have the same vector representation
- **Independence**: no association between terms (not true, but works in practice – **naïve** assumption)

Okapi BM25: A Nonbinary Model

- If the query is long, we might also use similar weighting for **query terms**

$$RSV_d = \sum_{t \in q} \left[\log \frac{N}{\mathrm{df}_t}\right] \cdot \frac{(k_1+1)\mathrm{tf}_{td}}{k_1((1-b)+b\times(L_d/L_{ave}))+\mathrm{tf}_{td}} \cdot \boxed{\frac{(k_3+1)\mathrm{tf}_{tq}}{k_3+\mathrm{tf}_{tq}}}$$

- tf_{tq}: term frequency in the query q
- k_3: tuning parameter controlling term frequency scaling of the query
- No length normalization of queries
(because retrieval is being done with respect to a single fixed query)
- The above tuning parameters should be set by optimization on a development test collection. Experiments have shown reasonable values for k_1 and k_3 as values between 1.2 and 2 and b = 0.75

An Appraisal of Probabilistic Models

- The difference between 'vector space' and 'probabilistic' IR is not that great:
  - In either case you build an information retrieval scheme in the exact same way.
  - Difference: for probabilistic IR, at the end, you score queries not by cosine similarity and tf-idf in a vector space, but by a slightly different formula motivated by probability theory

Using language models in IR

- Each document is treated as (the basis for) a language model.
- Given a query *q, r*ank documents based on $P(d|q)$

$$P(d|q) = \frac{P(q|d)P(d)}{P(q)}$$

- $P(q)$ is the same for all documents, so ignore
- $P(d)$ is the prior – often treated as the same for all d
  - But we can give a prior to "high-quality" documents, e.g., those with high static quality score g(d) (cf. Section 7.14).
- $P(q|d)$ is the probability of q given d.
- So to rank documents according to relevance to q, ranking according to $P(q|d)$ and $P(d|q)$ is equivalent.

Mixture model: Summary

$$P(q|d) \propto \prod_{1 \leq k \leq |q|} (\lambda P(t_k|M_d) + (1 - \lambda) P(t_k|M_c))$$

- What we model: The user has a document in mind and generates the query from this document.
- The equation represents the probability that the document that the user had in mind was in fact this one.

Week 12: Web Search

Chapter 19

- Web search big picture
- Search Advertising
- Duplicate Detection

Chapter 20

- Crawling

Chapter 21

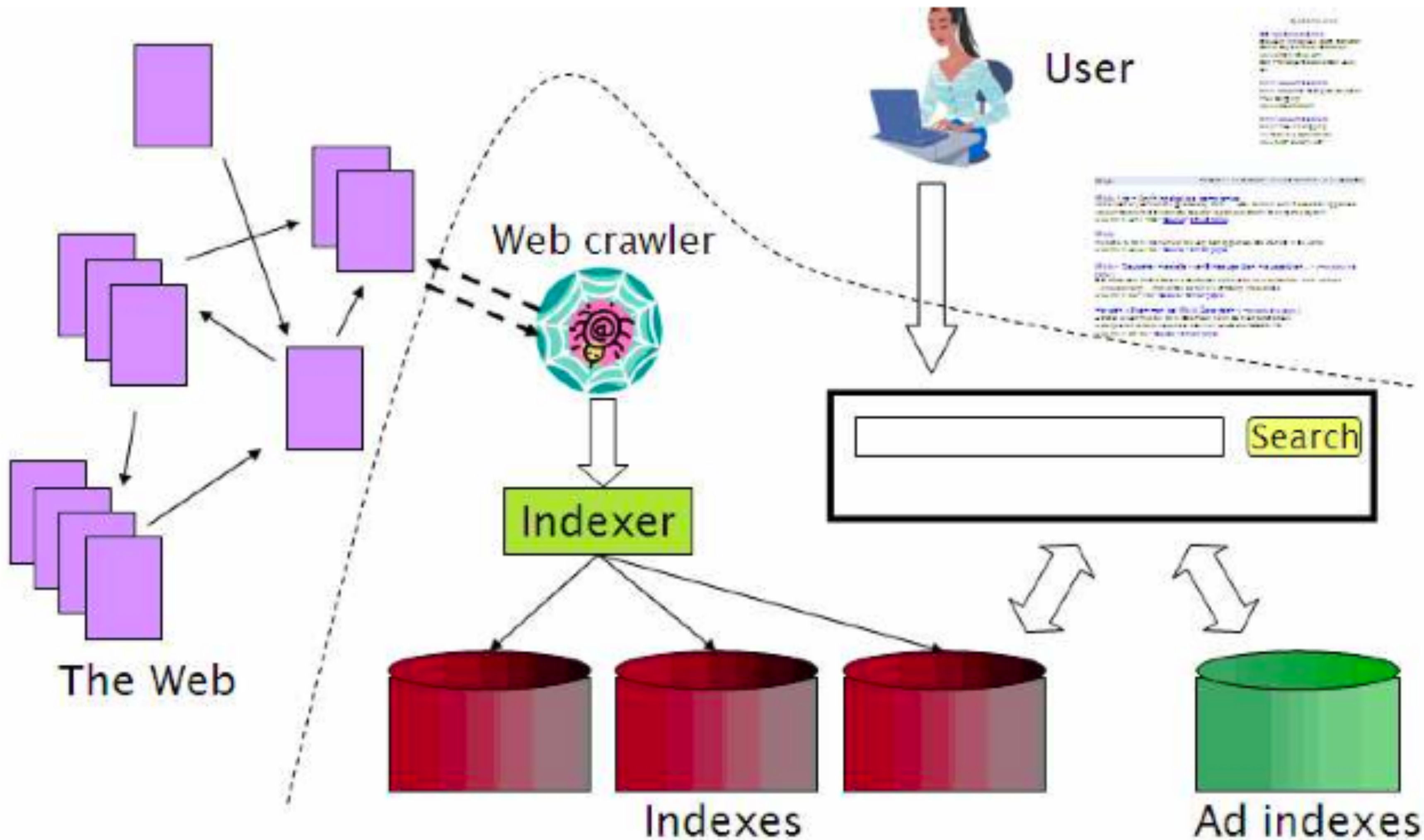
- Anchor Text
- PageRank

IR on the web vs. IR in general

- On the web, search is not just a nice feature.
  - Search is a key enabler of the web: financing, content creation, interest aggregation, etc.

  → look at search ads
- The web is a chaotic und uncoordinated collection.
  → lots of duplicates – need to detect duplicates
- No control / restrictions on who can author content.
  → lots of spam – need to detect spam
- The web is very large. → need to know how big it is.

Web search overview



Pagerank summary

- Pre-processing:
  - Given graph of links, build matrix **A**
  - From it compute **a**
  - The pagerank a_i is a scaled number between 0 and 1
- Query processing:
  - Retrieve pages meeting query
  - Rank them by their pagerank
  - Order is query-*independent*

PageRank issues

- Real surfers are not random surfers.
  - Examples of nonrandom surfing: back button, short vs. long paths, bookmarks, directories – and search!
  - → Markov model is not a good model of surfing.
  - But it’s good enough as a model for our purposes.
- Simple PageRank ranking (as described on previous slide) produces bad results for many pages.
  - Consider the query [video service].
  - The Yahoo home page (i) has a very high PageRank and (ii) contains both *video* and *service*.
  - If we rank all Boolean hits according to PageRank, then the Yahoo home page would be top-ranked.
  - Clearly not desireble.

WHERE TO GO FROM HERE

Learning Objectives

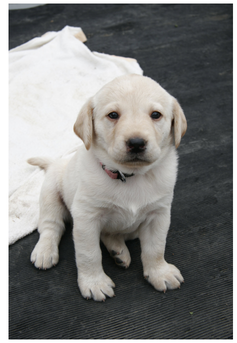
In addition to learning about IR, you have picked up skills that you will help in your future computing

- **Python** – one of the easiest and more straightforward programming languages to use.
- **NLTK** – A good set of routines and data that are useful in dealing with NLP and IR.


IR Theory
Natural Language Processing
Distributed IR
Computational Advertising
Geographic IR
Digital Libraries
Question Answering
Adversarial IR
Query Analysis
Social Network IR
Prob IR
Recommendation Systems
VSM
Boolean IR
Photo credits:
http://www.flickr.com/photos/aperezdc/

Opportunities in IR









Thanks for joining us for the journey!