Introduction to

Information Retrieval

Hinrich Schütze and Christina Lioma

http://nlp.stanford.edu/IR-book/

Outline

1 – Introduction

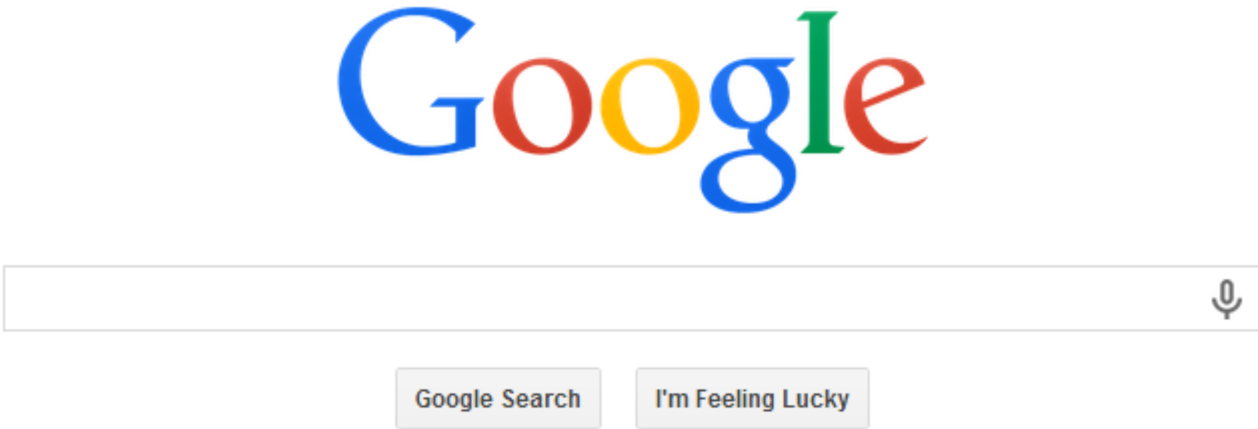
2 – Text

3 – Index

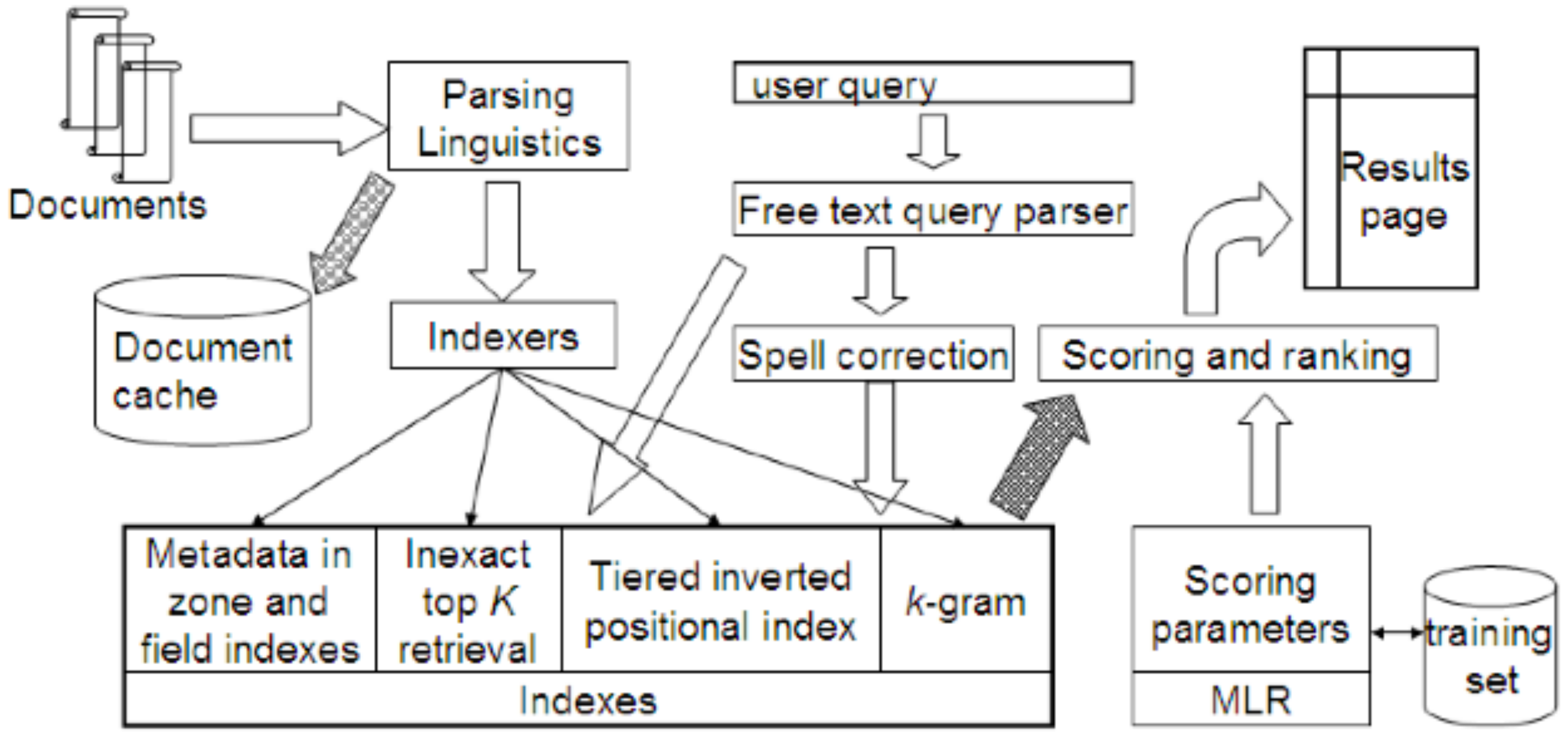
4 – Ranking

5 – System

Success of Google: “It is simple“



Complete search system



Definition of *information retrieval*

Information retrieval (IR) is finding material (usually documents) of

an unstructured nature (usually text) that satisfies an information

need from within large collections (usually stored on computers).

Boolean retrieval

- The Boolean model is arguably the simplest model to base an information retrieval system on.
- Queries are Boolean expressions, e.g., CAESAR AND BRUTUS
- The seach engine returns all documents that satisfy the
- Boolean expression.

Does Google use the Boolean model?

Unstructured data in 1650

- Which plays of Shakespeare contain the words BRUTUS AND CAESAR, but not CALPURNIA?
- One could grep all of Shakespeare's plays for BRUTUS and CAESAR, then strip out lines containing CALPURNIA
- Why is grep not the solution?
  - Slow (for large collections)
  - grep is line-oriented, IR is document-oriented
  - "NOT CALPURNIA" is non-trivial
  - Other operations (e.g., find the word ROMANS near COUNTRYMAN) not feasible

Term-document incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth . . .
ANTHONY	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
. . .						

Entry is 1 if term occurs. Example: CALPURNIA occurs in *Julius Caesar*.
Entry is 0 if term doesn't occur. Example: CALPURNIA doesn't occur in *The tempest*.

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer the query BRUTUS AND CAESAR AND NOT CALPURNIA:
  - Take the vectors for BRUTUS, CAESAR AND NOT CALPURNIA
  - Complement the vector of CALPURNIA
  - Do a (bitwise) and on the three vectors
  - 110100 AND 110111 AND 101111 = 100100

0/1 vector for BRUTUS

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth . . .
ANTHONY	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
. . .						
result:	1	0	0	1	0	0

Answers to query

Anthony and Cleopatra, Act III, Scene ii

Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,
When Antony found Julius Caesar dead,
He cried almost to roaring; and he wept
When at Philippi he found Brutus slain.

Hamlet, Act III, Scene ii

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

Bigger collections

- Consider $N = 10^6$ documents, each with about 1000 tokens
- $\Rightarrow$ total of 10^9 tokens
- On average 6 bytes per token, including spaces and punctuation $\Rightarrow$ size of document collection is about $6 \cdot 10^9 = 6$ GB
- Assume there are $M = 500{,}000$ distinct terms in the collection
- (Notice that we are making a term/token distinction.)

Can't build the incidence matrix

- $M = 500{,}000 \times 10^6$ = half a trillion 0s and 1s.
- But the matrix has no more than one billion 1s.
  - Matrix is extremely sparse.
- What is a better representations?
  - We only record the 1s.

Inverted Index

For each term *t*, we store a list of all documents that contain *t*.

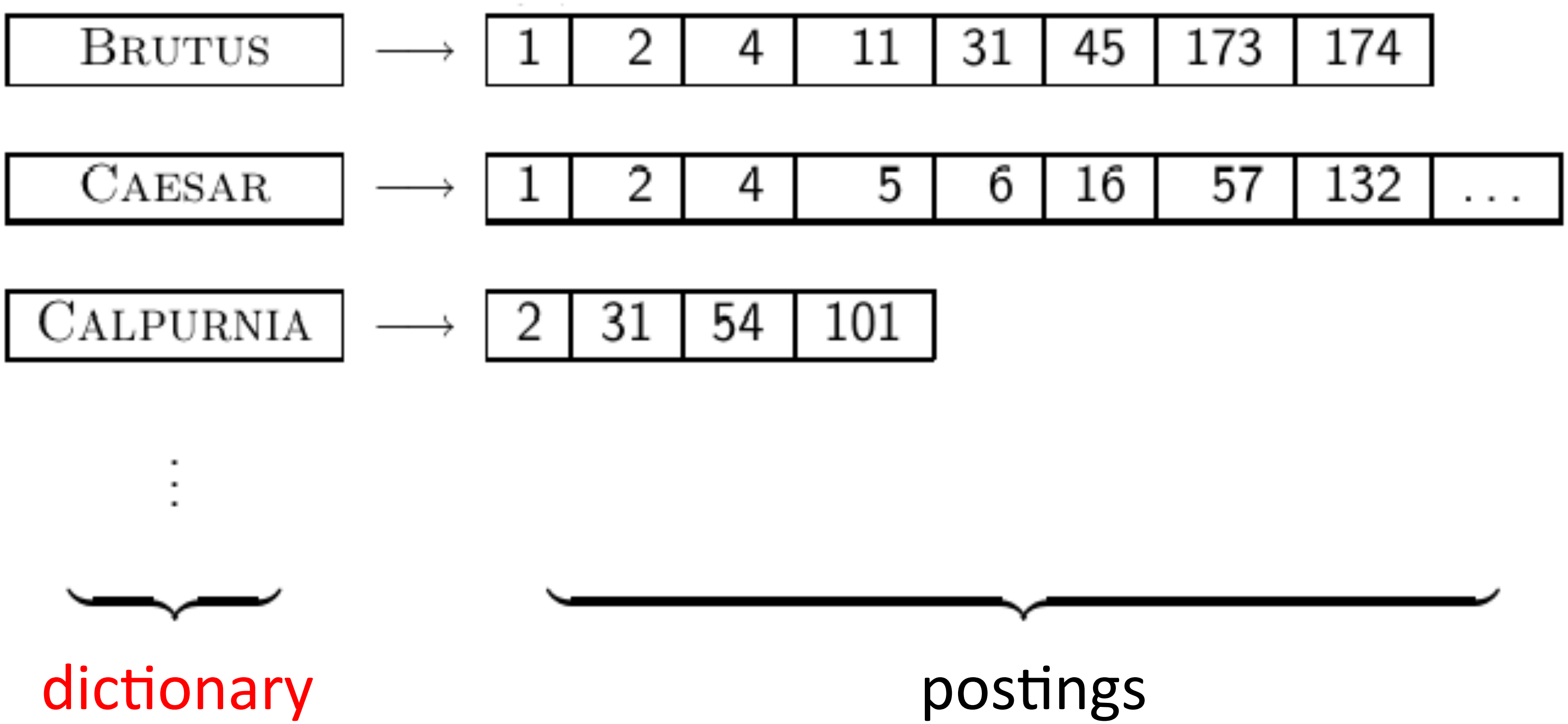
BRUTUS	→	1	2	4	11	31	45	173	174	
CAESAR	→	1	2	4	5	6	16	57	132	. . .
CALPURNIA	→	2	31	54	101					
⋮										

dictionary

postings

Inverted Index

For each term *t*, we store a list of all documents that contain *t*.



Inverted Index

For each term *t*, we store a list of all documents that contain *t*.

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
⋮		
dictionary		postings

Inverted index construction

1. Collect the documents to be indexed:

   $\boxed{\text{Friends, Romans, countrymen.}}$ $\boxed{\text{So let it be with Caesar}}$ . . .

2. Tokenize the text, turning each document into a list of tokens:

   $\boxed{\text{Friends}}$ $\boxed{\text{Romans}}$ $\boxed{\text{countrymen}}$ $\boxed{\text{So}}$ . . .

3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms: $\boxed{\text{friend}}$ $\boxed{\text{roman}}$ $\boxed{\text{countryman}}$ $\boxed{\text{so}}$ . . .

4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

Tokenizing and preprocessing

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:

$\Longrightarrow$

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

Generate posting

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

$\Longrightarrow$

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

Sort postings

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

$\Longrightarrow$

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

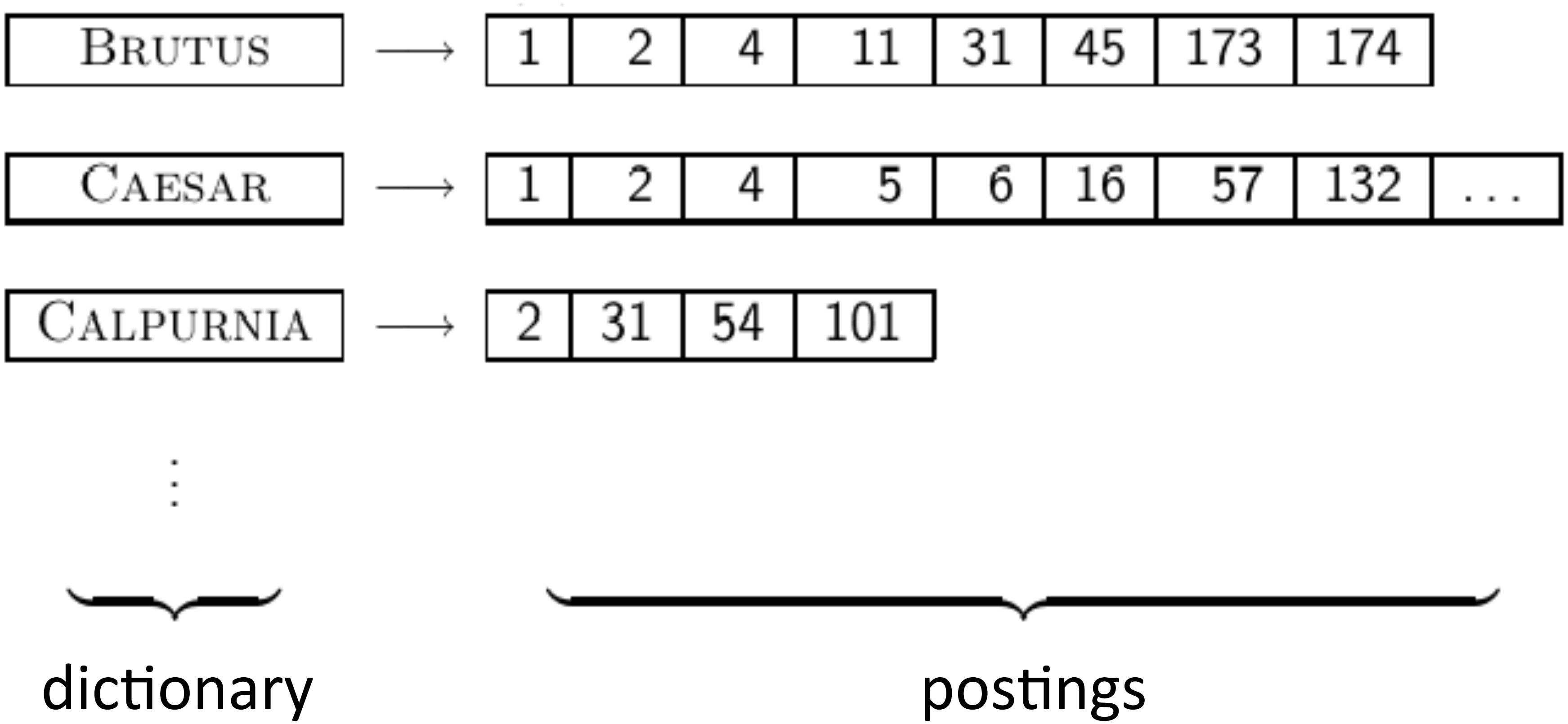
Create postings lists, determine document frequency

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

Split the result into dictionary and postings file



Simple conjunctive query (two terms)

- Consider the query: BRUTUS AND CALPURNIA
- To find all matching documents using inverted index:
  1. Locate BRUTUS in the dictionary
  2. Retrieve its postings list from the postings file
  3. Locate CALPURNIA in the dictionary
  4. Retrieve its postings list from the postings file
  5. Intersect the two postings lists
  6. Return intersection to user

Intersecting two posting lists

BRUTUS ⟶ 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174

CALPURNIA ⟶ 2 → 31 → 54 → 101

Intersection ⟹ 2 → 31

- This is linear in the length of the postings lists.
- Note: This only works if postings lists are sorted.

Boolean queries

- The Boolean retrieval model can answer any query that is a Boolean expression.
  - Boolean queries are queries that use AND, OR and NOT to join
  - query terms.
  - Views each document as a set of terms.
  - Is precise: Document matches condition or not.
- Primary commercial retrieval tool for 3 decades
- Many professional searchers (e.g., lawyers) still like Boolean queries.
  - You know exactly what you are getting.
- Many search systems you use are also Boolean: spotlight, email, intranet etc.

Does Google use the Boolean model?

- On Google, the default interpretation of a query [w_1 w_2 . . .w_n] is w_1 AND w_2 AND . . .AND w_n
- Cases where you get hits that do not contain one of the *wi :*
  - anchor text
  - page contains variant of w_i (morphology, spelling correction, synonym)
  - long queries (*n* large)
  - boolean expression generates very few hits
- Simple Boolean vs. Ranking of result set
  - Simple Boolean retrieval returns matching documents in no particular order.
  - Google (and most well designed Boolean engines) rank the result set – they rank good hits (according to some estimator of relevance) higher than bad hits.

Review – Introduction

1 – Information Retrieval Problem

2 – Inverted index

- Dictionary
- Posts

3 – Boolean retrieval

Outline

1 – Introduction

2 – Text

3 – Index

4 – Ranking

5 – System

Parsing a document

- We need to deal with format and language of each document.
- What format is it in? pdf, word, excel, html etc.
- What language is it in?
- What character set is in use?
- Each of these is a classification problem
- Alternative: use heuristics

Format/Language: Complications

- A single index usually contains terms of several languages.
  - Sometimes a document or its components contain multiple languages/formats.
  - French email with Spanish pdf attachment
- What is the document unit for indexing?
- A file?
- An email?
- An email with 5 attachments?
- A group of files (ppt or latex in HTML)?
- Upshot: Answering the question "what is a document?" is not trivial and requires some design decisions.
- Also: XML

Definitions

- Word – A delimited string of characters as it appears in the text.
- Term – A "normalized" word (case, morphology, spelling etc); an equivalence class of words.
- Token – An instance of a word or term occurring in a document.
- Type – The same as a term in most cases: an equivalence class of tokens.

Normalization

- Need to "normalize" terms in indexed text as well as query terms into the same form.
- Example: We want to match *U.S.A.* and *USA*
- We most commonly implicitly define equivalence classes of terms.
- Alternatively: do asymmetric expansion
  - window → window, windows
  - windows → Windows, windows
  - Windows (no expansion)
- More powerful, but less efficient
- Why don't you want to put *window, Window, windows,* and *Windows* in the same equivalence class?

Recall: Inverted index construction

- Input:

Friends, Romans, countrymen.	So let it be with Caesar	. . .

- Output:

friend	roman	countryman	so	. . .

- Each token is a candidate for a postings entry.
- What are valid tokens to emit?

Tokenization problems: One word or two? (or several)

- Hewlett-Packard
- State-of-the-art
- co-education
- the hold-him-back-and-drag-him-away maneuver
- data base
- San Francisco
- Los Angeles-based company
- cheap San Francisco-Los Angeles fares
- York University vs. New York University

Numbers

- 3/20/91
- 20/3/91
- Mar 20, 1991
- B-52
- 100.2.86.144
- (800) 234-2333
- 800.234.2333
- Older IR systems may not index numbers . . .
- . . . but generally it's a useful feature.

Chinese: No whitespace

莎拉波娃现在居住在美国东南部的佛罗里达。今年４月９日，莎拉波娃在美国第一大城市纽约度过了１８岁生日。生日派对上，莎拉波娃露出了甜美的微笑。

Ambiguous segmentation in Chinese

和尚

The two characters can be treated as one word meaning 'monk' or as a sequence of two words meaning 'and' and 'still'.

Other cases of “no whitespace”

- Compounds in Dutch, German, Swedish
- Computerlinguistik → Computer + Linguistik
- Lebensversicherungsgesellschaftsangestellter
- → leben + versicherung + gesellschaft + angestellter
- Inuit: tusaatsiarunnanngittualuujunga (I can’t hear very well.)
- Many other languages with segmentation difficulties: Finnish, Urdu, . . .

Japanese

ノーベル平和賞を受賞したワンガリ・マータイさんが名誉会長を務めるＭＯＴＴＡＩＮＡＩキャンペーンの一環として、毎日新聞社とマガジンハウスは「私の、もったいない」を募集します。皆様が日ごろ「もったいない」と感じて実践していることや、それにまつわるエピソードを８００字以内の文章にまとめ、簡単な写真、イラスト、図などを添えて１０月２０日までにお送りください。大賞受賞者には、５０万円相当の旅行券とエコ製品２点の副賞が贈られます。

4 different “alphabets”: Chinese characters, hiragana syllabary for inflectional endings and functional words, katakana syllabary for transcription of foreign words and other uses, and latin. No spaces (as in Chinese). End user can express query entirely in hiragana!

Arabic script

ك ِ ت ا ب ٌ ⇐ كِتابٌ

un b ā t i k

/kitābun/ *‘a book’*

Arabic script: Bidirectionality

استقلت الجزائر في سنة 1962 بعد 132 عاما من الاحتلال الفرنسي.

← → ← → ← START

'Algeria achieved its independence in 1962 after 132 years of French occupation.'

Bidirectionality is not a problem if text is coded in Unicode.

Accents and diacritics

- Accents: résumé vs. resume (simple omission of accent)
- Umlauts: Universität vs. Universitaet (substitution with special letter sequence “ae”)
- Most important criterion: How are users likely to write their queries for these words?
- Even in languages that standardly have accents, users often do not type them. (Polish?)

Case folding

- Reduce all letters to lower case
- Possible exceptions: capitalized words in mid-sentence
- MIT vs. mit
- Fed vs. fed
- It's often best to lowercase everything since users will use lowercase regardless of correct capitalization.

Stop words

- stop words = extremely common words which would appear to be of little value in helping select documents matching a user need
- Examples: *a, an, and, are, as, at, be, by, for, from, has, he, in, is, it, its, of, on, that, the, to, was, were, will, with*
- Stop word elimination used to be standard in older IR systems.
- But you need stop words for phrase queries, e.g. "King of Denmark"
- Most web search engines index stop words.

More equivalence classing

- Soundex: phonetic equivalence: Muller = Mueller
- Thesauri: semantic equivalence, car = automobile

Lemmatization

- Reduce inflectional/variant forms to base form
- Example: *am, are, is* → *be*
- Example: *car, cars, car's, cars'* → *car*
- Example: *the boy's cars are different colors* → *the boy car be different color*
- Lemmatization implies doing "proper" reduction to dictionary headword form (the lemma).
- Inflectional morphology (*cutting* → *cut)* vs. derivational morphology (*destruction* → *destroy)*

Stemming

- Definition of stemming: Crude heuristic process that chops off the ends of words in the hope of achieving what "principled" lemmatization attempts to do with a lot of linguistic knowledge.
- Language dependent
- Often inflectional and derivational
- Example for derivational: *automate, automatic, automation* all reduce to *automat*

Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options
- Conventions + 5 phases of reductions
- Phases are applied sequentially
- Each phase consists of a set of commands.
  - Sample command: Delete final *ement* if what remains is longer than 1 character
  - replacement → replac
  - cement → cement
- Sample convention: Of the rules in a compound command, select the one that applies to the longest suffix.

Porter stemmer: A few rules

Rule	Example
SSES → SS	caresses → caress
IES → I	ponies → poni
SS → SS	caress → caress
S →	cats → cat

Three stemmers: A comparison

Sample text: Such an analysis can reveal features that are not easil visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

Porter stemmer: such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to pictur of express that is more biolog transpar and access to interpret

Lovins stemmer: such an analys can reve featur that ar not eas vis from th vari in th individu gen and can lead to a pictur of expres that is mor biolog transpar and acces to interpres

Paice stemmer: such an analys can rev feat that are not easy vis from the vary in the individ gen and can lead to a pict of express that is mor biolog transp and access to interpret

Exercise: What does Google do?

- Stop words
- Normalization
- Tokenization
- Lowercasing
- Stemming
- Non-latin alphabets
- Umlauts
- Compounds
- Numbers

Phrase queries

- We want to answer a query such as [stanford university] – as a phrase.
- Thus *The inventor Stanford Ovshinsky never went to university* should not be a match.
- The concept of phrase query has proven easily understood by users.
- About 10% of web queries are phrase queries.
- Consequence for inverted index: it no longer suffices to store docIDs in postings lists.
- Two ways of extending the inverted index:
  - biword index
  - positional index

Biword indexes

- Index every consecutive pair of terms in the text as a phrase.
- For example, *Friends, Romans, Countrymen* would generate two biwords: *"friends romans"* and *"romans countrymen"*
- Each of these biwords is now a vocabulary term.
- Two-word phrases can now easily be answered.

Issues with biword indexes

- Why are biword indexes rarely used?
- False positives, as noted above
- Index blowup due to very large term vocabulary

Positional indexes

- Positional indexes are a more efficient alternative to biword indexes.
- Postings lists in a nonpositional index: each posting is just a docID
- Postings lists in a positional index: each posting is a docID and a list of positions

Positional indexes: Example

Query: *"to$_1$ be$_2$ or$_3$ not$_4$ to$_5$ be$_6$"* TO, 993427:

‹ 1: ‹7, 18, 33, 72, 86, 231›;

2: ‹1, 17, 74, 222, 255›;

4: ‹8, 16, 190, 429, 433›;

5: ‹363, 367›;

7: ‹13, 23, 191›; . . . ›

BE, 178239:

‹ 1: ‹17, 25›;

4: ‹17, 191, 291, 430, 434›;

5: ‹14, 19, 101›; . . . › Document 4 is a match!

Proximity search

- We just saw how to use a positional index for phrase searches.
- We can also use it for proximity search.
- For example: employment /4 place
- Find all documents that contain EMPLOYMENT and PLACE within 4 words of each other.
- *Employment agencies that place healthcare workers are seeing growth* is a hit.
- *Employment agencies that have learned to adapt now place healthcare workers* is not a hit.

Proximity search

- Use the positional index
- Simplest algorithm: look at cross-product of positions of (i) EMPLOYMENT in document and (ii) PLACE in document
- Very inefficient for frequent words, especially stop words
- Note that we want to return the actual matching positions, not just a list of documents.
- This is important for dynamic summaries etc.

Combination scheme

- Biword indexes and positional indexes can be profitably combined.
- Many biwords are extremely frequent: Michael Jackson, Britney Spears etc
- For these biwords, increased speed compared to positional postings intersection is substantial.
- Combination scheme: Include frequent biwords as vocabulary terms in the index. Do all other phrases by positional intersection.
- Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme. Faster than a positional index, at a cost of 26% more space for index.

"Positional" queries on Google

- For web search engines, positional queries are much more expensive than regular Boolean queries.
- Let's look at the example of phrase queries.
- Why are they more expensive than regular Boolean queries?
- Can you demonstrate on Google that phrase queries are more expensive than Boolean queries?

Review – Text

1 – Definitions

- Document
- Term

2 – Techniques

- Tokenization
  - Case folding
  - Stop words
- Lemmatization
- Stemming

3 – Phrase queries

- Biword
- Positional

Outline

1 – Introduction

2 – Text

3 – Index

4 – Ranking

5 – System

Outline – Index

1 – BSBI algorithm

2 – SPIMI algorithm

3 – Distributed indexing

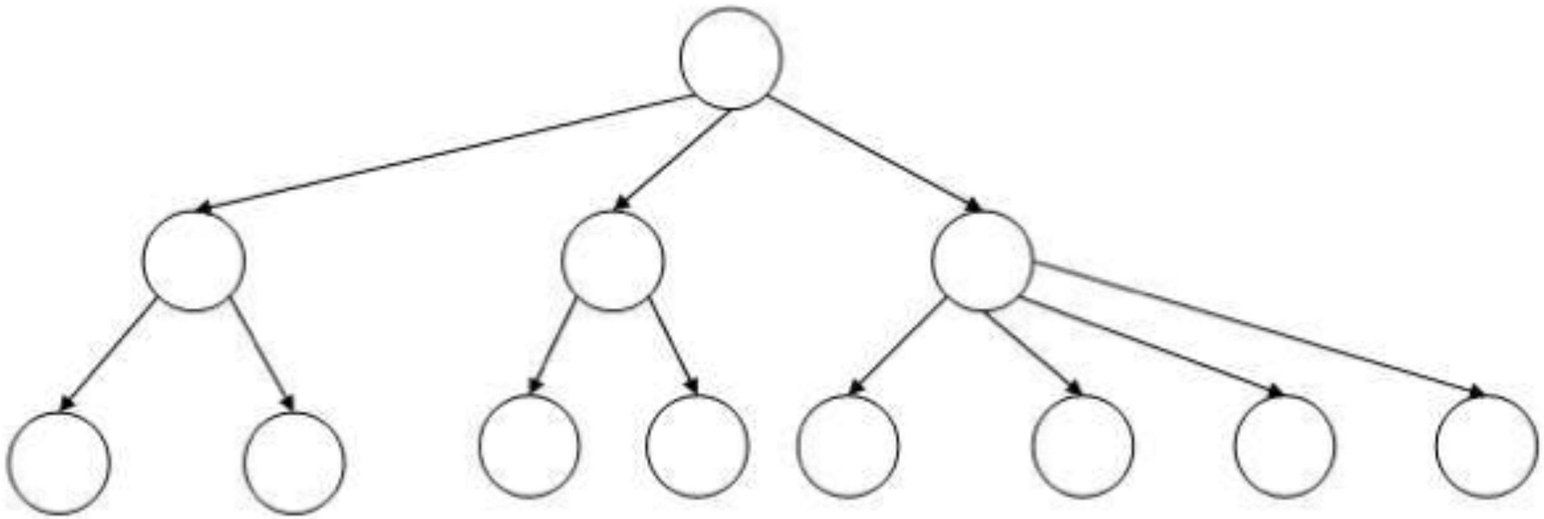
4 – Dynamic indexing

Dictionary as array of fixed-width entries

term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$

space needed: 20 bytes 4 bytes 4 bytes

B-tree for looking up entries in array



Hardware basics

- Many design decisions in information retrieval are based on hardware constraints.
- We begin by reviewing hardware basics that we'll need in this course.

Hardware basics

- Access to data is much faster in memory than on disk. (roughly a factor of 10)
- Disk seeks are "idle" time: No data is transferred from disk while the disk head is being positioned.
- To optimize transfer time from disk to memory: one large chunk is faster than 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
- Servers used in IR systems typically have several GB of main memory, sometimes tens of GB, and TBs or 100s of GB of disk space.
- Fault tolerance is expensive: It's cheaper to use many regular machines than one fault tolerant machine.

Some stats (ca. 2008)

symbol	statistic	value
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	lowlevel 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

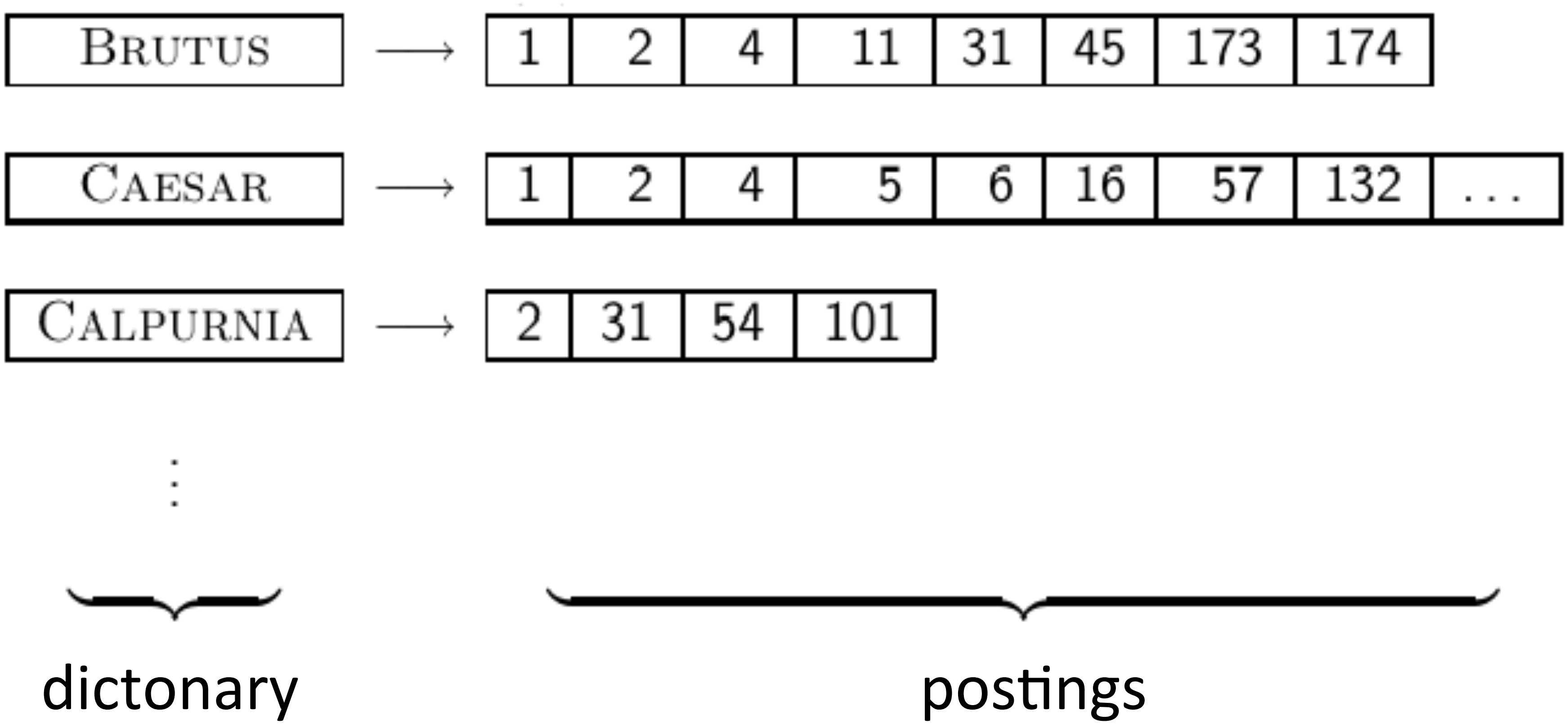
Reuters RCV1 statistics

N	documents	800,000
L	tokens per document	200
M	terms (= word types)	400,000
	bytes per token (incl. spaces/punct.)	6
	bytes per token (without spaces/punct.)	4.5
	bytes per term (= word type)	7.5
T	non-positional postings	100,000,000

Exercise: Average frequency of a term (how many tokens)? 4.5

bytes per word token vs. 7.5 bytes per word type: why the difference? How many positional postings?

Goal: construct the inverted Index



Index construction in IIR 1: Sort postings in memory

term	docID		term	docID
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	$\Longrightarrow$	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

Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- Can we keep all postings in memory and then do the sort in-memory at the end?
- No, not for large collections
- At 10–12 bytes per postings entry, we need a lot of space for large collections.
- *T* = 100,000,000 in the case of RCV1: we can do this in memory on a typical machine in 2010.
- But in-memory index construction does not scale for large collections.
- Thus: We need to store intermediate results on disk.

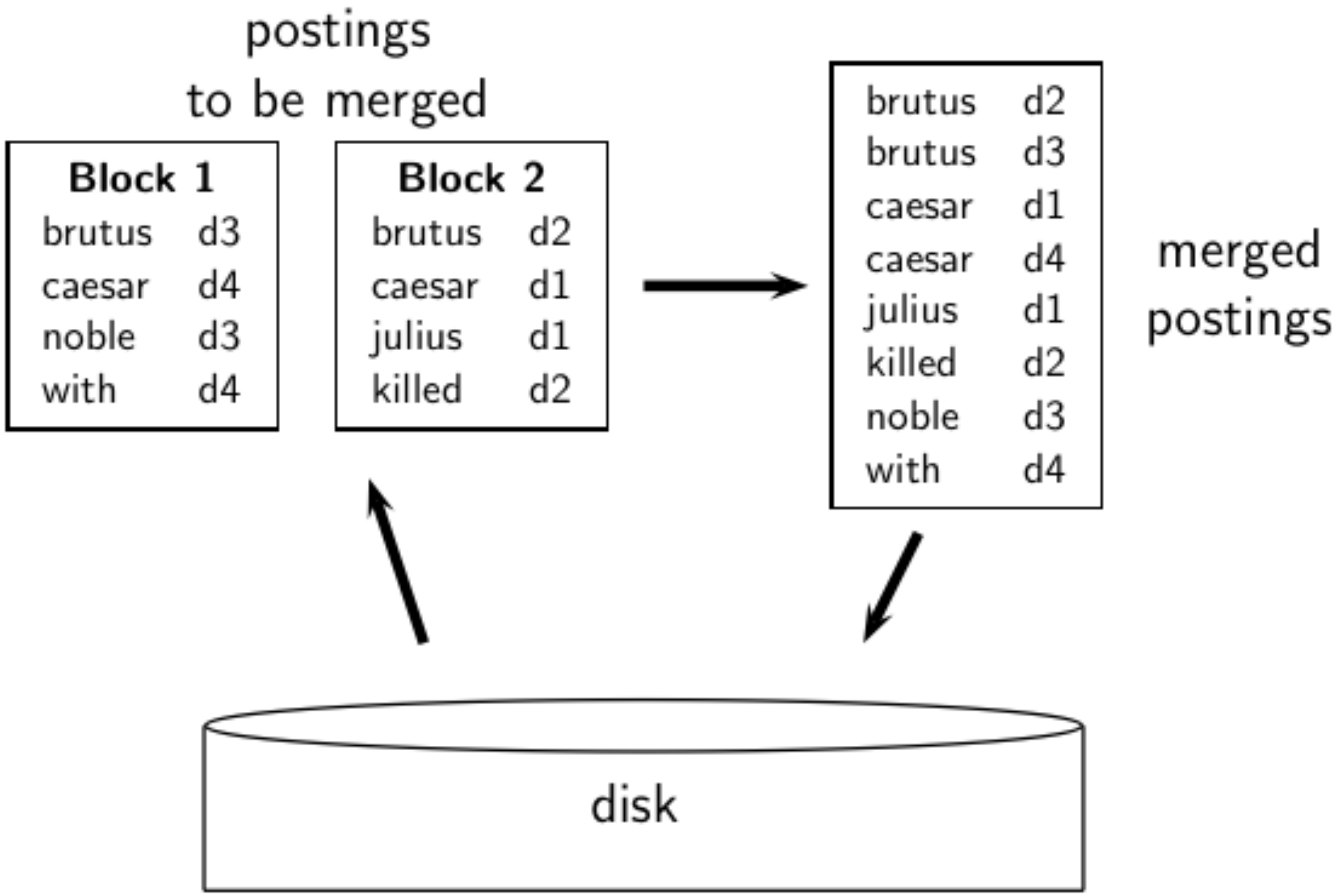
Same algorithm for disk?

- 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.

"External" sorting algorithm (using few disk seeks)

- We must sort T = 100,000,000 non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, document frequency).
- Define a block to consist of 10,000,000 such postings
  - We can easily fit that many postings into memory.
  - We will have 10 such blocks for RCV1.
- Basic idea of algorithm:
  - For each block: (i) accumulate postings, (ii) sort in memory, (iii) write to disk
  - Then merge the blocks into one long sorted order.

Merging two blocks



Blocked Sort-Based Indexing

```
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)
```

- Key decision: What is the size of one block?

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.)

Outline – Index

1 – BSBI algorithm

2 – SPIMI algorithm

3 – Distributed indexing

4 – Dynamic indexing

Single-pass in-memory indexing

- Abbreviation: SPIMI
- 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.

Outline – Index

Distributed indexing

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

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 define two sets of parallel tasks and deploy two types of machines to solve them:
  - Parsers
  - Inverters
- Break the input document collection into splits (corresponding to blocks in BSBI/SPIMI)
- Each split is a subset of documents.

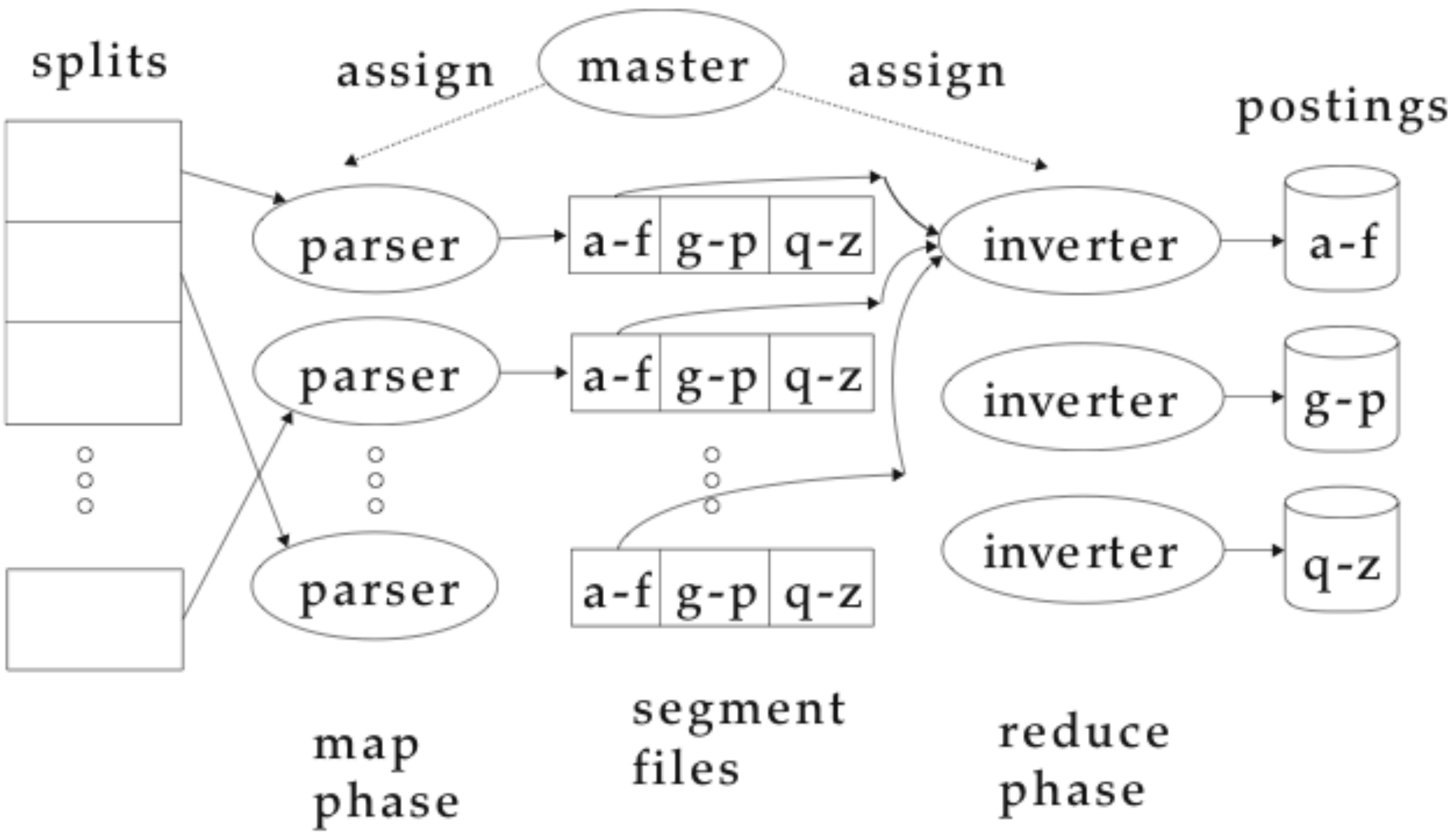
Parsers

- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and emits (term,docID)-pairs.
- Parser writes pairs into j term-partitions.
- Each for a range of terms' first letters
  - E.g., a-f, g-p, q-z (here: $j = 3$)

Inverters

- An inverter collects all (term,docID) pairs (= postings) for one term-partition (e.g., for a-f).
- Sorts and writes to postings lists

Data flow


splits
assign
master
assign
postings
parser
a-f
g-p
q-z
inverter
a-f
parser
a-f
g-p
q-z
inverter
g-p
parser
a-f
g-p
q-z
inverter
q-z
map phase
segment files
reduce phase

MapReduce

- The index construction algorithm we just described is an instance of MapReduce.
- MapReduce is a robust and conceptually simple framework for distributed computing . . .
- . . .without having to write code for the distribution part.
- The Google indexing system (ca. 2002) consisted of a number of phases, each implemented in MapReduce.
- Index construction was just one phase.
- Another phase: transform term-partitioned into document-partitioned index.

Index construction in MapReduce

Schema of map and reduce functions

map:	input	$\rightarrow$ list(k, v)
reduce:	(k,list(v))	$\rightarrow$ output

Instantiation of the schema for index construction

map:	web collection	$\rightarrow$ list(termID, docID)
reduce:	($\langle$termID$_1$, list(docID)$\rangle$, $\langle$termID$_2$, list(docID)$\rangle$, ...)	$\rightarrow$ (postings_list$_1$, postings_list$_2$, ...)

Example for index construction

map:	d_2 : C DIED. d_1 : C CAME, C C'ED.	$\rightarrow$ ($\langle$C, $d_2\rangle$, $\langle$DIED,$d_2\rangle$, $\langle$C,$d_1\rangle$, $\langle$CAME,$d_1\rangle$, $\langle$C,$d_1\rangle$, $\langle$C'ED,$d_1\rangle$)
reduce:	($\langle$C,(d_2,d_1,d_1)$\rangle$,$\langle$DIED,(d_2)$\rangle$,$\langle$CAME,(d_1)$\rangle$,$\langle$C'ED,(d_1)$\rangle$)	$\rightarrow$ ($\langle$C,(d_1:2,d_2:1)$\rangle$,$\langle$DIED,(d_2:1)$\rangle$,$\langle$CAME,(d_1:1)$\rangle$,$\langle$C'ED,(d_1:1)$\rangle$)

Exercise

- What information does the task description contain that the master gives to a parser?
- What information does the parser report back to the master upon completion of the task?
- What information does the task description contain that the master gives to an inverter?
- What information does the inverter report back to the master upon completion of the task?

Outline – Index

1 – BSBI algorithm

2 – SPIMI algorithm

3 – Distributed indexing

4 – Dynamic indexing

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are: Documents are inserted, deleted and modified.
- This means that the dictionary and postings lists have to be dynamically modified.

Dynamic indexing: Simplest approach

- Maintain big main index on disk
- New docs go into small auxiliary index in memory.
- Search across both, merge results
- Periodically, merge auxiliary index into big index
- Deletions:
  - Invalidation bit-vector for deleted docs
  - Filter docs returned by index using this bit-vector

Issue with auxiliary and main index

- Frequent merges
- Poor search performance during index merge
- Actually:
  - Merging of the auxiliary index into the main index is not that costly 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.
- 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 into several files, collect small postings lists 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.
- 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 ← Merge(Z_0, {token})
  if |Z_0| = n
     then for i ← 0 to ∞
              do if I_i ∈ indexes
                    then Z_{i+1} ← Merge(I_i, Z_i)
                          (Z_{i+1} is a temporary index on disk.)
                         indexes ← indexes − {I_i}
                    else I_i ← Z_i     (Z_i becomes the permanent index I_i.)
                         indexes ← indexes ∪ {I_i}
                         Break
              Z_0 ← ∅

LogarithmicMerge()
  Z_0 ← ∅     (Z_0 is the in-memory index.)
  indexes ← ∅
  while true
  do LMergeAddToken(indexes, Z_0, getNextToken())
```

Binary numbers: $I_3I_2I_1I_0 = 2^3 2^2 2^1 2^0$

- 0001
- 0010
- 0011
- 0100
- 0101
- 0110
- 0111
- 1000
- 1001
- 1010
- 1011
- 1100

Logarithmic merge

- Number of indexes bounded by $O(\log T)$ (T is total number of postings read so far)
- So query processing requires the merging of $O(\log T)$ indexes
- Time complexity of index construction is $O(T \log T)$.
- . . . because each of T postings is merged $O(\log T)$ times.
- Auxiliary index: index construction time is $O(T^2)$ as each posting is touched in each merge.
  - Suppose auxiliary index has size a
  - $a + 2a + 3a + 4a + \ldots + na = a\frac{n(n+1)}{2} = O(n^2)$
- So logarithmic merging is an order of magnitude more efficient.

Dynamic indexing at large search engines

- Often a combination
  - Frequent incremental changes
  - Rotation of large parts of the index that can then be swapped in
  - Occasional complete rebuild (becomes harder with increasing size – not clear if Google can do a complete rebuild)

Review – Index

- Two index construction algorithms: BSBI (simple) and SPIMI (more realistic)
- Distributed index construction: MapReduce
- Dynamic index construction: how to keep the index up-to-date as the collection changes

Resources

- Apache: Lucene and Solr
- Resources at http://ifnlp.org/ir
  - Original publication on MapReduce by Dean and Ghemawat (2004)
  - Original publication on SPIMI by Heinz and Zobel (2003)
  - YouTube video: Google data centers