

Multimedia Information Retrieval and Technology

Lecture 4. Index Construction

By : Fangyu Wu

Room: SD555



Xi'an Jiaotong-Liverpool University

西交利物浦大學

Recap

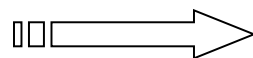
Inverted index

Brutus

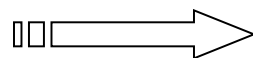
Caesar

Calpurnia

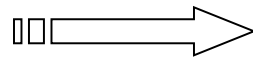
Dictionary



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



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



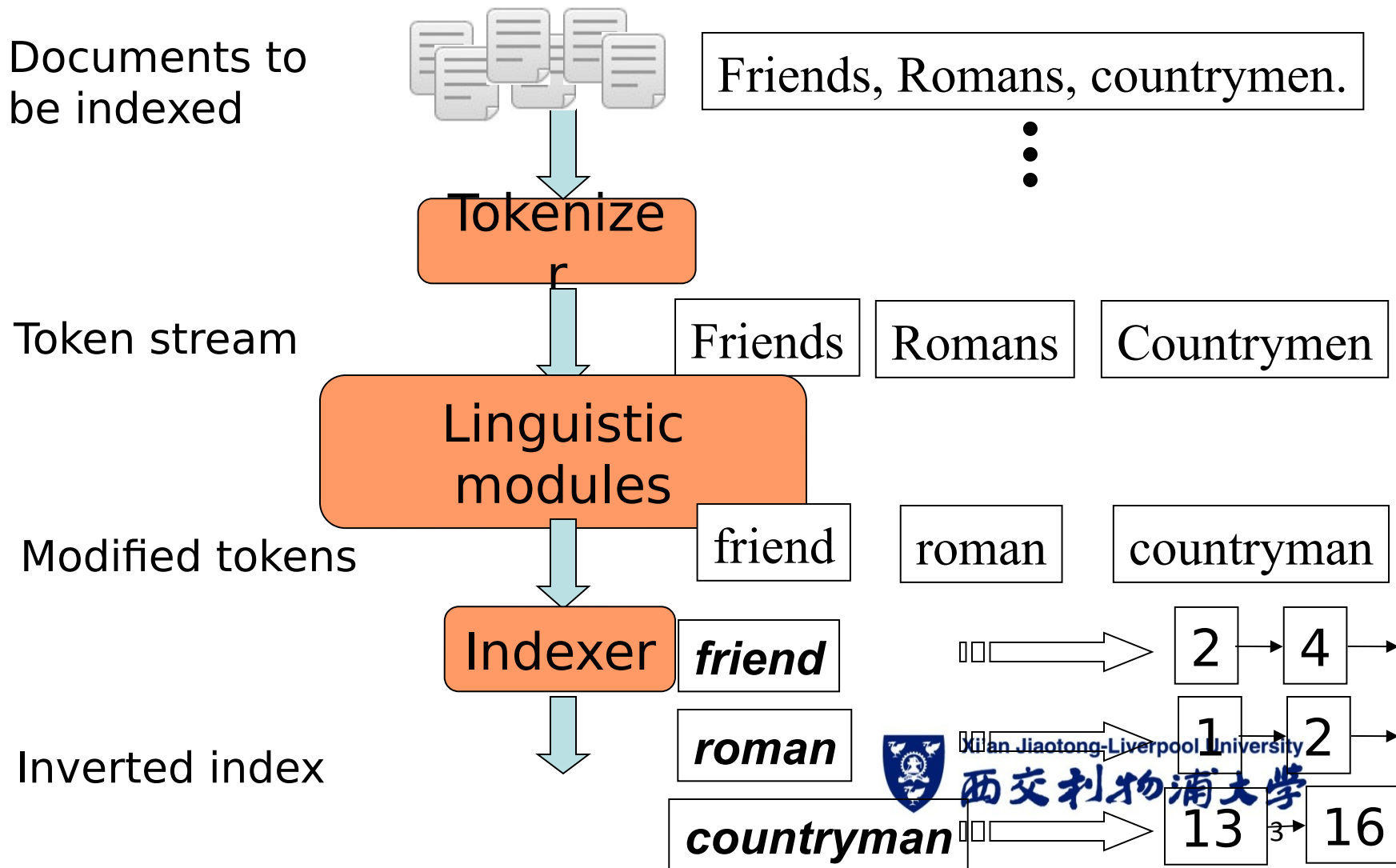
2	31	54	101				
---	----	----	-----	--	--	--	--

Posting list



Postings

Recall the basic indexing pipeline

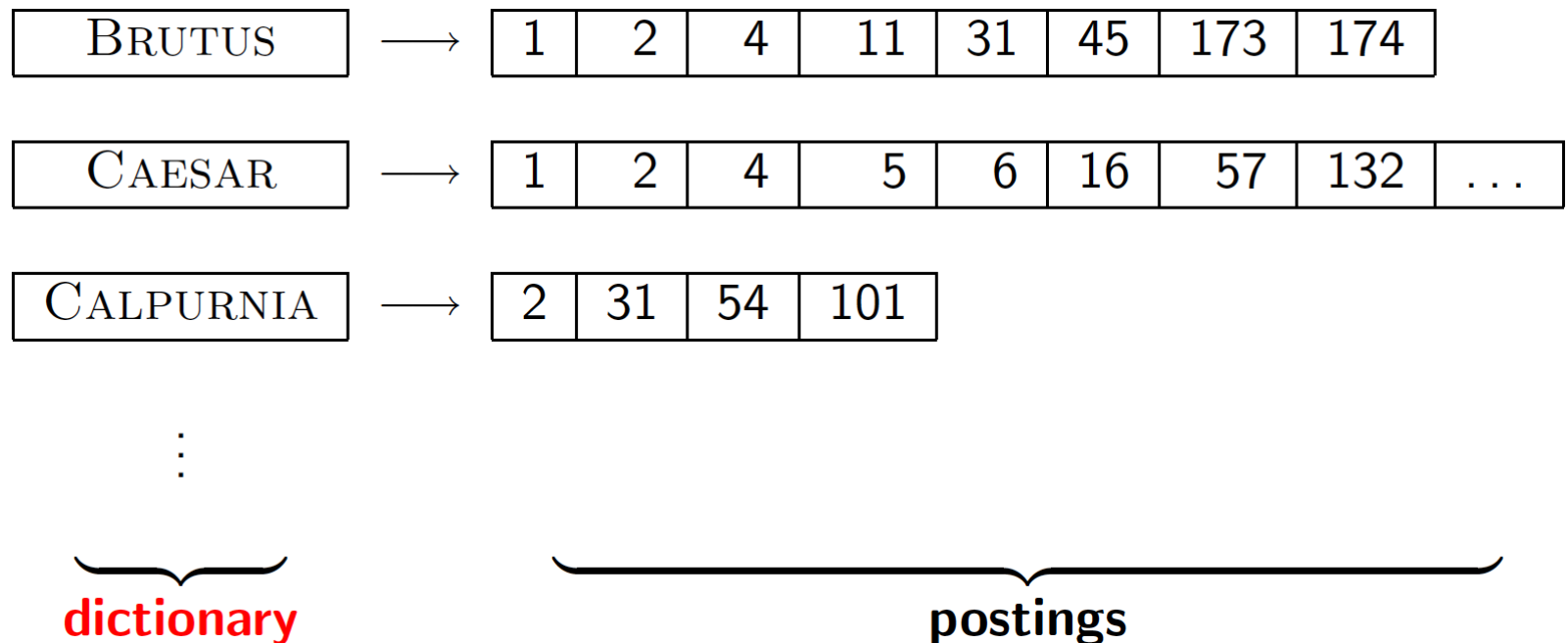


Recap of the previous lecture

- The type/token/term distinction
 - Terms are normalized types(tokens) to be put in the dictionary
- Tokenization problems:
 - Hyphens, apostrophes, compounds
- Term equivalence classing:
 - stemming, lemmatization
- Skip pointers
- Biword indexes for phrases
- Positional indexes for phrases/proximity queries

Dictionary data structures for inverted indexes

The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list ... **in what data structure?**



- “Tolerant” retrieval
 - Spelling correction
- Index Construction with large collection

Spell correction

Two principal uses

- Correcting document(s) being indexed
- Correcting user queries to retrieve “right” answers

Two main flavors:

- Isolated word: Check each word on its own for misspelling. Will not catch typos resulting in correctly spelled words
e.g., ***from** → form*
- Context-sensitive: Look at surrounding words,
e.g., ***I fly form Heathrow to Narita.***



<http://www.google.com/jobs/archive/britney.html>

Google reports that the following are all treated as misspellings of the query **britney spears**:
britian spears, britney's spears, brandy spears and prittany spears.

Spelling correction

- Isolated term correction
 - Edit distance
 - n-gram overlap
- Context-sensitive correction

Isolated word correction

- Fundamental premise 1: – there is a lexicon (or a list of correct words) from which the correct spellings come.
- Fundamental premise 2: we can compute the **distance** between a misspelled word and a correct word.

Isolated word correction

Given a lexicon and a character sequence Q , return the words in the lexicon closest to Q .

What's "closest"?

We'll study several alternatives

- Edit distance (Levenshtein distance)
- Weighted edit distance
- n-gram overlap

Edit distance

Edit distance: Given two strings S_1 and S_2 , the minimum number of operations to convert one to the other.

- Operations are typically character-level

Insert, Delete, Replace, (Transposition)

E.g., the edit distance from **dof** to **dog** is 1

From **cat** to **act** is 2 (Just 1 with transpose.)

from **cat** to **dog** is 3.



Edit distance

EDITDISTANCE(s_1, s_2)

```

1  int  $m[i, j] = 0$ 
2  for  $i \leftarrow 1$  to  $|s_1|$ 
3  do  $m[i, 0] = i$ 
4  for  $j \leftarrow 1$  to  $|s_2|$ 
5  do  $m[0, j] = j$ 
6  for  $i \leftarrow 1$  to  $|s_1|$ 
7  do for  $j \leftarrow 1$  to  $|s_2|$ 
8      do  $m[i, j] = \min\{m[i-1, j-1] + \text{if } (s_1[i] = s_2[j]) \text{ then } 0 \text{ else } 1, \text{fi},$ 
9           $m[i-1, j] + 1,$ 
10          $m[i, j-1] + 1\}$ 
11 return  $m[|s_1|, |s_2|]$ 

```

The three quantities correspond to substituting a character in , inserting a character in , inserting a character in.

► **Figure 3.5** Dynamic programming algorithm for computing the edit distance between strings s_1 and s_2 .

Edit distance

			f	a	s	t
		0	1 1	2 2	3 3	4 4
c		1 1	<i>1 2</i> <i>2 1</i>	<i>2 3</i> <i>2 2</i>	<i>3 4</i> <i>3 3</i>	<i>4 5</i> <i>4 4</i>
a		2 2	<i>2 2</i> <i>3 2</i>	<i>1 3</i> <i>3 1</i>	<i>3 4</i> <i>2 2</i>	<i>4 5</i> <i>3 3</i>
t		3 3	<i>3 3</i> <i>4 3</i>	<i>3 2</i> <i>4 2</i>	<i>2 3</i> <i>3 2</i>	<i>2 4</i> <i>3 2</i>
s		4 4	<i>4 4</i> <i>5 4</i>	<i>4 3</i> <i>5 3</i>	<i>2 3</i> <i>4 2</i>	<i>3 3</i> <i>3 3</i>

► **Figure 3.6** Example Levenshtein distance computation. The 2×2 cell in the $[i, j]$ entry of the table shows the three numbers whose minimum yields the fourth. The cells in italics determine the edit distance in this example.

Exercise

Compute the edit distance between *paris* and *alice*.
Write down the 5×5 array of distances between all prefixes as computed by the algorithm in Figure 3.5.

Solution

		a	l	i	c	e
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$	$\frac{5}{5}$
p	$\frac{\quad}{1}$	$\frac{1}{2}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$	$\frac{5}{5}$
a	$\frac{2}{2}$	$\frac{1}{3}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$	$\frac{5}{5}$
r	$\frac{3}{3}$	$\frac{3}{4}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$	$\frac{5}{5}$
i	$\frac{4}{4}$	$\frac{4}{5}$	$\frac{3}{4}$	$\frac{2}{4}$	$\frac{4}{3}$	$\frac{5}{4}$
s	$\frac{5}{5}$	$\frac{5}{6}$	$\frac{4}{5}$	$\frac{4}{5}$	$\frac{3}{4}$	$\frac{4}{4}$



Weighted edit distance

As edit distance, but **the weight** of an operation depends on the character(s) involved

Example: ***m*** more likely to be mis-typed as ***n*** than as ***q***

Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***

This may be formulated as a probability model

- Requires weight matrix as input

Context-sensitive spell correction

Text: *I flew from Heathrow to Narita.*

Consider the phrase query “*flew form Heathrow*”

We’d like to respond

Did you mean “*flew from Heathrow*”?
because no docs matched the query phrase.

Spelling correction

- Isolated term correction
 - Edit distance
 - n-gram overlap
- Context-sensitive correction

Context-sensitive correction

First idea: retrieve dictionary terms close (in weighted edit distance) to each query term

Now try all possible resulting phrases with one word “fixed” at a time

flew from heathrow

fled form heathrow

flea form heathrow

Hit-based spelling correction: Suggest the alternative that has lots of hits.

- “Tolerant” retrieval
 - Spelling correction
- Index Construction with large collection

Hardware basics

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

We begin by reviewing hardware basics

Access to data in memory is ***much*** faster than access to data on disk.

Hardware basics

Consequently, we want to keep as much data as possible in memory, especially those data that we need to access frequently.

We call the technique of keeping frequently used disk data in main memory *caching*.

Hardware basics

When doing a disk read or write, it takes a while for the disk head to move to the part of the disk where the data are located.

This time is called the *seek time* and it averages 5 ms for typical disks.

Hardware basics

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

Data transfers from disk to memory are handled by the system bus, not by the processor.

The processor is available to process data during disk I/O.

We can exploit this fact to speed up data transfers by storing compressed data on disk.

Hardware basics

Assuming an efficient decompression algorithm, the total time of reading and then decompressing compressed data is usually less than reading uncompressed data.

Index Construction

RECAP:

1. We first make a collection assembling all term–docID pairs.
2. We then sort the pairs with the term as the dominant key and docID as the secondary key.
3. Finally, we organize the docIDs for each term into a postings list and compute statistics like term and document frequency.

For small collections, all this can be done in memory.
But what if the collection is very large?

Reuters RCV1: Our collection for this lecture

A typical document is shown in next page, but note that we ignore multimedia information like images and are only concerned with text.

Reuters-RCV1 covers a wide range of international topics, including politics, business, sports, and (as in this example) science.

Reuters RCV1 statistics

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	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
T	tokens	100,000,000



A Reuters RCV1 document



You are here: [Home](#) > [News](#) > [Science](#) > [Article](#)

Go to a Section: [U.S.](#) [International](#) [Business](#) [Markets](#) [Politics](#) [Entertainment](#) [Technology](#) [Sports](#) [Oddly Enough](#)

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: Our collection for this lecture

The collection isn't large enough either, but it's publicly available and is at least a more plausible example.

To describe methods for large collections that require the use of secondary storage.

Recall IIR 1 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 tokens to sort for Reuters RCV1 .

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




Bottleneck

Parse and build postings entries one doc at a time

Now sort postings entries by term (then by doc within each term)

Doing this with random disk seeks would be too slow
– must sort 100M records for RCV1



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



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

With main memory insufficient, we need to use *an external sorting algorithm*, that is, one that uses disk.

For acceptable speed, the central requirement of such an algorithm is that it minimize the number of random disk seeks during sorting – sequential disk reads are far faster than seeks

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

- (i) segments the collection into parts of equal size,
- (ii) sorts the termID–docID pairs of each part in memory,
- (iii) stores intermediate sorted results on disk
- (iv) merges all intermediate results into the final index.

BSBINDEXCONSTRUCTION()

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

BSBI: Blocked sort-based Indexing

PARSENEXTBLOCK: The algorithm parses documents into termID–docID pairs and accumulates the pairs in memory until a block of a fixed size is full.

We choose the block size to fit comfortably into memory to permit a fast in-memory sort. The block is then inverted and written to disk.

Inversion (BSBI_Invert)

- we sort the termID–docID pairs.
- Next, we merge all termID–docID pairs with the same termID into a postings list, where a posting is simply a list of docID.

BSBI: Blocked sort-based Indexing

The result, an inverted index for the block we have just read, is then written to disk.

In the final step, the algorithm simultaneously merges all the blocks into one large merged index.

BSBI: Blocked sort-based Indexing

we open all block files simultaneously, and maintain small read buffers for all the blocks we are reading and a write buffer for the final merged index we are writing.

In each iteration, we select **the lowest termID** that has not been processed yet using a priority queue or a similar data structure.

All postings lists for this termID are read and merged, and the merged list is written back to disk.

Each read buffer is refilled from its file when necessary.

Dynamic indexing

Up to now, we have assumed that collections are static.

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

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

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

Exercise

Draw yourself a diagram showing the various indexes in a search engine incorporating all the functionality we have talked about.

Identify some of the key design choices in the search engine pipeline.