Indexing and boolean retrieval

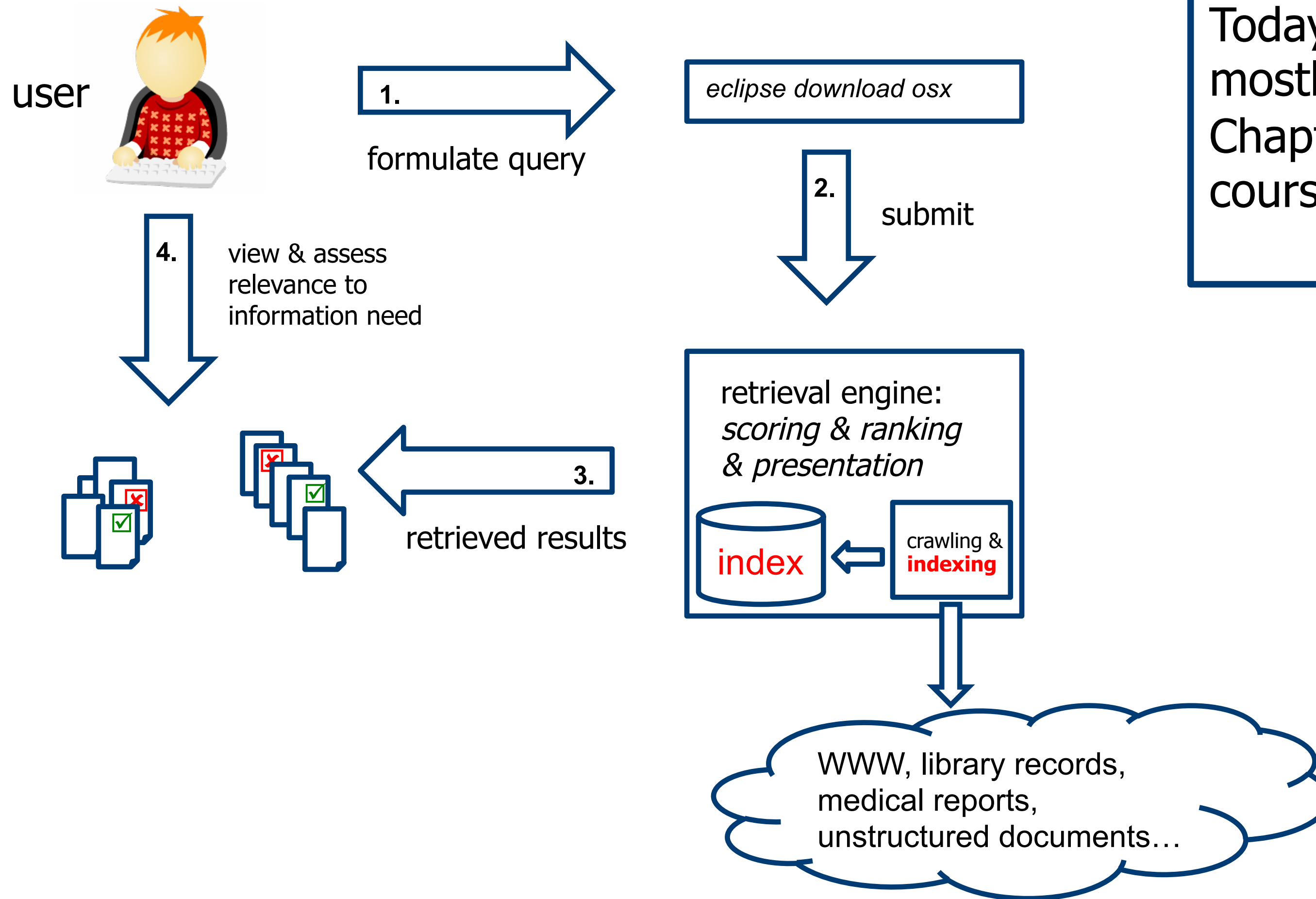
IN4325 – Information Retrieval

TU Delft

Organizational matters

- So far only 17 people emailed me about group enrollment
- A few students have reported problems with the Hadoop installation
  - If you are still struggling please have a chat with me in the break or email me providing:
    - A link to the instructions you are using to install Hadoop
    - The specific step at which you are struggling
    - The error message(s) you get
    - Your operating system and Hadoop version

Today ...



Today's lecture is mostly based on Chapters 1-3 of the course book [1].

Why so complicated? (from lecture #1)

- Searching for the lines in the book *Count of Monte Christo* that contain the terms *Dantes* AND *prison* but NOT *Albert*
- Naïve solution
  - Grep all lines that contain *Dantes*, then grep those containing *prison* and finally strip out lines containing *Albert*

```
more countOfMonteChristo.txt|grep Dantes|grep prison|grep -v Villefort
```

- Problems
  - Proximity operations not easy to implement (e.g. *Dantes* within max. 3 terms of *prison*)
  - *Set of matching results (yes/no decision)*
  - What about approximate/semantic matches (Edmond instead of Dantes, cell instead of prison. etc.)

Why so complicated? (from lecture #1)

- Searching for the lines in the book *Count of Monte Christo* that contain the terms *Dantes* AND *prison* but NOT *Albert*

- Naïve solution
  - Grep all lines that contain *Dantes*, then grep those containing *prison* and finally strip out lines containing *Albert*

```
xt|grep Dantes|grep prison|grep -v Villefort
```

Elaborate queries require the user to anticipate possibly used terms:
(*Edmond* OR *Dantes*) AND (*prison* OR *cell* OR *imprisoned*) NOT *Albert*

ent (e.g. *Dantes* within

  - *Set of matching results (yes/no decision)*
  - What about approximate/semantic matches (Edmond instead of Dantes, cell instead of prison. etc.)

Why so complicated? II (from lecture #1)

Boolean retrieval

- What about using a *term-document (incidence)* matrix?
  - Here: a line is a document

	L1	L2	L3	L4	L5	L6	L7	L8	L9
Edmond	0	0	0	1	1	0	1	0	1
Dantes	1	0	1	0	1	0	1	0	0
Cristo	0	1	0	0	0	1	0	0	0
prison	1	0	0	1	0	0	1	1	0
cell	0	1	0	0	0	0	1	0	0
imprisoned	1	0	1	0	0	0	1	0	0
Albert	1	1	0	0	1	0	0	1	0

1 if *Edmond* occurs in line L9; 0 otherwise

Only feasible for extremely small corpora. The matrix gets too large too quickly.

54,544 lines (#docs) and 19,236 unique terms: more than 1 billion matrix entries; If 1 bit per entry: 125MB memory needed

Dantes AND *prison* ⇨ bitwise AND

```
1 0 1 0 1 0 1 0 0
1 0 0 1 0 0 1 1 0
-----------------
1 0 0 0 0 0 1 0 0
```

⇨ L1 L7

Dantes NOT *Albert* ⇨ bitwise AND complement

```
1 0 1 0 1 0 1 0 0
0 0 1 1 0 1 1 0 1
-----------------
0 0 1 0 0 0 1 0 0
```

⇨ L3 L7

Inverted index

- Maps terms back to the parts of the documents they occur in

```
albert       ---> 1 | 2 | 5 | 8
cell         ---> 2 | 3 | 7
dantes       ---> 1 | 3 | 5 | 7
edmond       ---> 4 | 5 | 7 | 9
imprisoned   ---> 1 | 7
cristo       ---> 2 | 6
prison       ---> 1 | 4 | 7 | 8
```

- Dictionary alphabetically sorted
- Postings ordered by docid

docID
(document identifier)

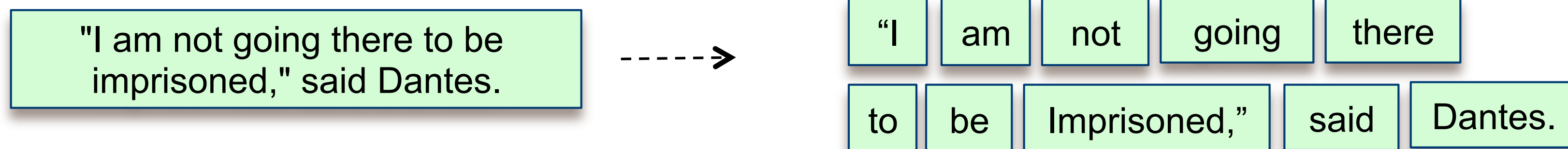
dictionary

posting list

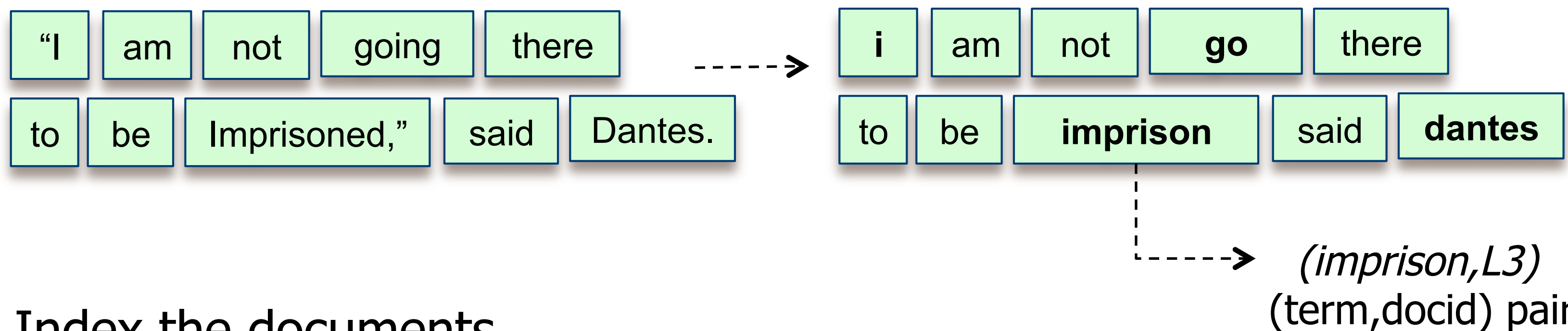
How to build an inverted index

1. Collect the documents to index
2. Tokenize the content: from string to tokens

L3



3. Normalize the tokens (preprocessing)



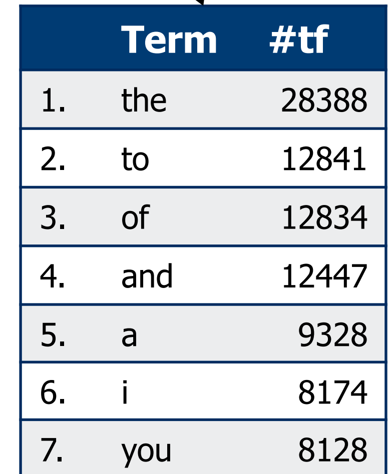
4. Index the documents

Tokenization & normalization I

- Tokenization is not always straight-forward
  - E-mail: *email* or {*e,mail*}*?*
  - It's: *its* or *{it,s}?*
  - What about *O-β-D-galactopyranosyl-(1→4)-D-glucopyranose*?
  - What about documents containing many floats 2.43254534234323234324325…. ?
  - "The sun is shining." in simplified Chinese: 阳光普照。
- Case folding [2]
  - *{the,The,THE,tHE}* are all matched to *the*
  - *General AND Motors* should not retrieve *"general repairs to all kinds of motors" (exception can be handled by a postretrieval scan)*

Tokenization & normalization II

- Stopword removal
  - Term frequencies: *The Count of Monte Cristo*
  - *Stopwords occur with very high frequencies often not adding any value*
  - *What about the query "to be or not to be"?*
  - Standard stopword list vs. corpus-dependent (domain-dependent) lists



	Term	#tf
1.	the	28388
2.	to	12841
3.	of	12834
4.	and	12447
5.	a	9328
6.	i	8174
7.	you	8128

- Stemming
  - Reduce terms to their root form (strip suffixes), e.g.

    {compressed,compression} ➔ *compress*

    {walking,walked,walks} ➔ *walk*

Tokenization & normalization III

- Stemming cont.
  - It is not appropriate for all types of documents or parts of documents
    - Author names in scientific papers or book catalogues, etc.
  - Two standard stemmers (for English): Krovetz (1993) and Porter stemmer (1979) [3,4]

Clear sky, swift-flitting boats, and brilliant sunshine disappeared; the heavens were hung with black, and the gigantic structure of the Chateau d'If seemed like the phantom of a mortal enemy.

Clear sky swift flit boat and brilliant sunshin disappear the heaven were hung with black and the gigant structur of the Chateau d If seem like the phantom of a mortal enemi

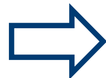
Porter stemmed

Let's focus on step 4

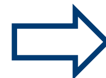
Doc 1: "I am not going there to be imprisoned," said Dantes.

Doc 2: "You are Edmond Dantes," cried Villefort, seizing the count by the wrist; "then come here!"

term	docID
i	1
am	1
not	1
go	1
there	1
to	1
be	1
imprison	1
said	1
dantes	1
you	2
are	2
edmon	2
dantes	2
cri	2
villefort	2
seiz	2
the	2
count	2
by	2
the	2
wrist	2
then	2
come	2
here	2



term	docID
am	1
are	2
be	1
by	2
come	2
count	2
cri	2
dantes	1
dantes	2
edmond	2
go	1
here	2
i	1
imprison	1
not	1
said	1
seiz	2
the	2
the	2
then	2
there	1
to	1
villefort	2
wrist	2
you	2



term doc.freq.		
am 1	------->	1
are 1	------->	2
be 1	------->	1
by 1	------->	2
come 1	------->	2
count 1	------->	2
cri 1	------->	2
dantes 2	------->	1 2
edmond 1	------->	2
go 1	------->	1
here 1	------->	2
i 1	------->	1
imprison 1	------->	1
not 1	------->	1
said 1	------->	1
seiz 1	------->	2
the 2	------->	2
then 1	------->	2
there 1	------->	1
to 1	------->	1
villefort1	------->	2
wrist 1	------->	2
you 1	------->	2

dictionary

posting list

Boolean retrieval over posting lists

Dantes AND Albert

1) Process the query in the same manner as the corpus
2) Determine whether both query terms exist
3) Locate pointers to the respective posting lists

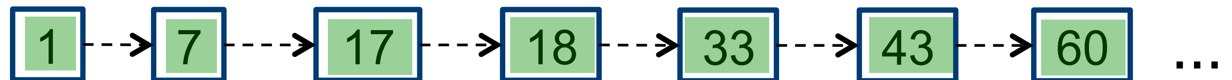
Dantes 1 → 7 → 17 → 18 → 33 → 43 → 60 ...

Albert 4 → 7 → 54 → 60 → 61 → 82 → 96 ...

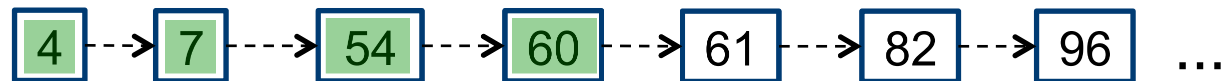
Boolean retrieval over posting lists

Dantes AND Albert

Dantes



Albert



result

Boolean retrieval over posting lists

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

posting lists

common docID found in both lists

increase the posting list counters

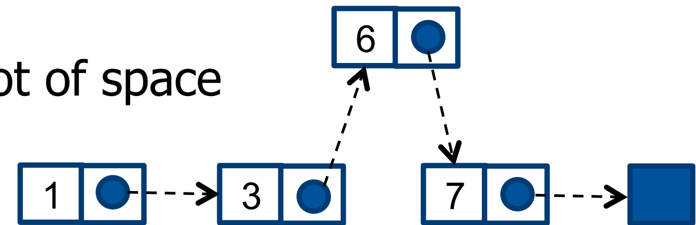
Source: [1]

Posting lists data structures

Implementation

- Index needs to be optimized for
  - Storage & access efficiency
  - Today: fast CPUs and slow disk-access (reducing posting list sizes has priority)

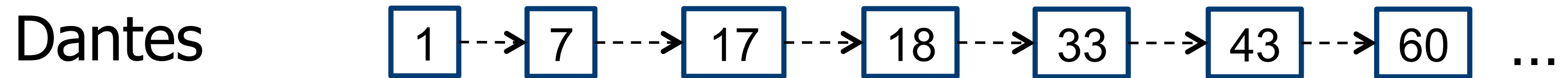
- How to implement posting lists?
  - Fixed length array: easy, wastes a lot of space
  - Singly linked list: cheap insertion
  - Variable length arrays
    - Require less space than linked lists (no pointers)
    - Allow faster access (contiguous memory increases)
    - Good if few updates are required



Posting list data structures

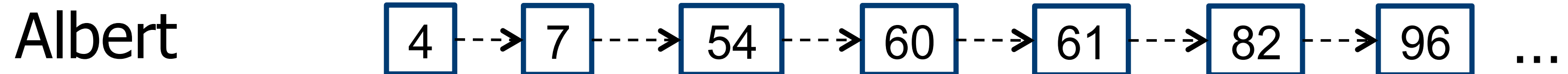
Skip pointers

Dantes



AND

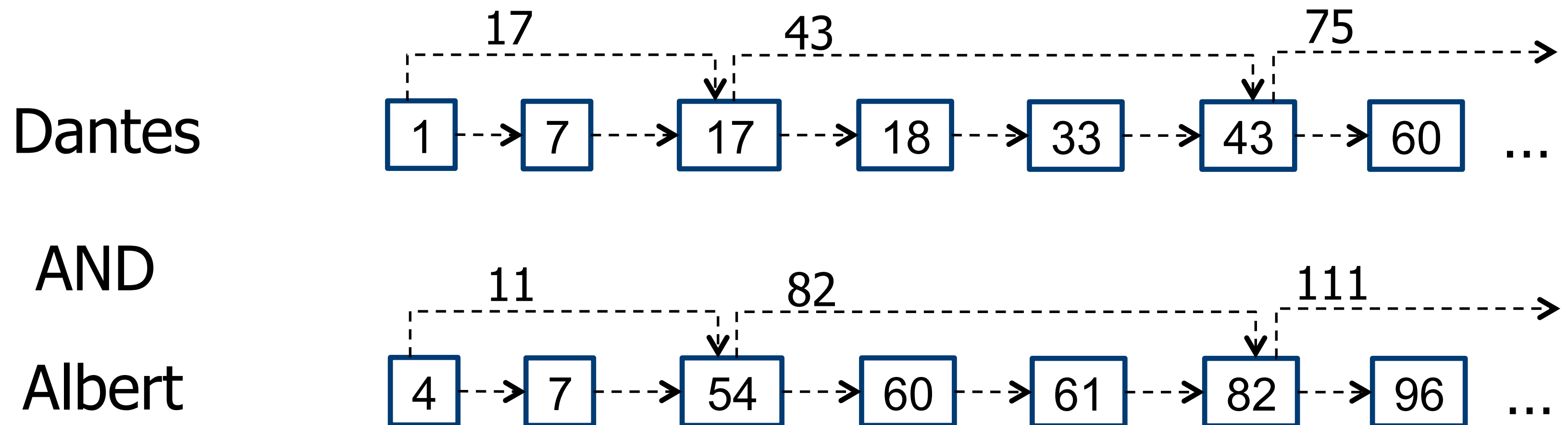
Albert



List intersection without skip pointers: O(n+m)

Posting list data structures I

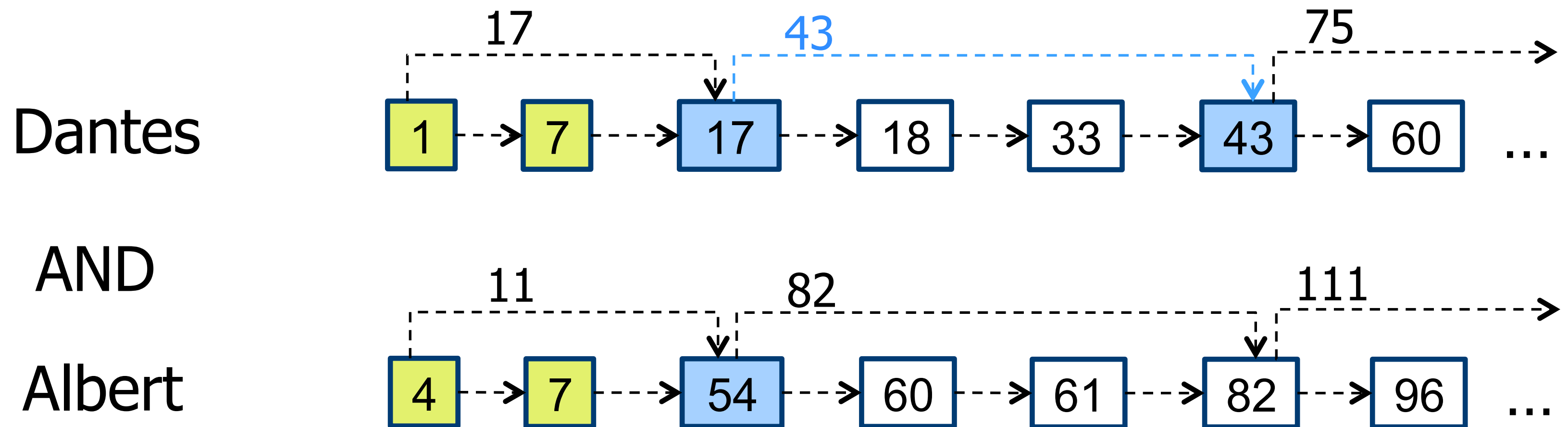
Skip pointers



List intersection without skip pointers: O(n+m)

Posting list data structures II

Skip pointers



List intersection without skip pointers: O(n+m)
List intersection with skip pointers: sub-linear

Question: What about OR queries?

Posting list data structures III

Skip pointers

```
INTERSECTWITHSKIPS(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4       then ADD(answer, docID(p1))
 5            p1 ← next(p1)
 6            p2 ← next(p2)
 7       else if docID(p1) < docID(p2)
 8              then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
 9                     then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
10                            do p1 ← skip(p1)
11                     else p1 ← next(p1)
12              else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
13                     then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
14                            do p2 ← skip(p2)
15                     else p2 ← next(p2)
16  return answer
```

posting lists

common docID found in both lists

increase the posting list counters, skip if possible

Source: [1]

Posting list data structures IV

Skip pointers: where to place them

- Tradeoff
  - More skips yield shorter skip spans; more skips are likely
    - Requires many skip pointer comparisons and pointer storage
  - Fewer skips yield larger skip spans; few skips are likely
    - Requires less comparisons, fewer space
- Heuristic: for posting lists of length ℓ, use $sqrt(\ell)$ evenly spaced skip pointers
  - Ignores particularities of the **query** terms distribution
- Effective skip pointers are easy to create in static indices, harder when the posting lists are frequently updated

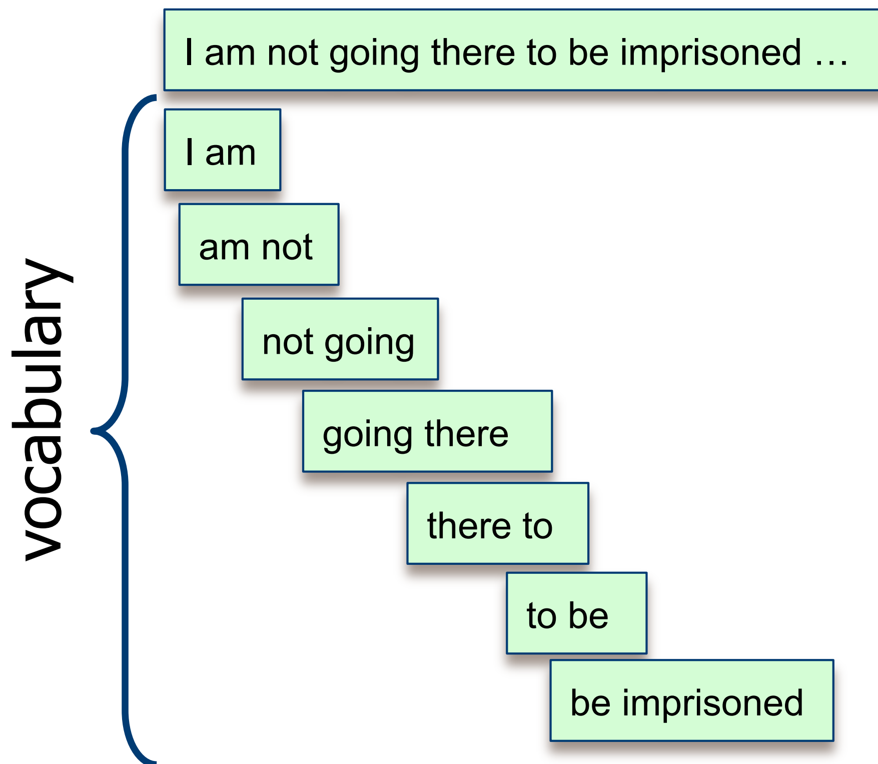
Positional postings

- Concepts and names may be multiword compounds, e.g. "*Edmond Dantes*"
  - If treated as a phrase, it should not return the sentence: "*Edmond went to the town of Dantes.*"
  - Web search engines introduced the "…." syntax for phrase queries (~10% of posed queries are explicit phrase queries [1])

  - Posting lists of the form `term`➔`d1|d2|d3|..` do not provide sufficient granularity
    - Would require a lot of postretrieval filtering

Biword indices

- **Biword** = every **pair** of consecutive words



- Each biword is one vocabulary term
- Two-word phrase queries can be handled immediately
- Longer phrase queries are broken down, e.g. "*Count of Monte Cristo*"
  - *"Count of"* AND *"of Monte"* AND *"Monte Cristo"*
  - Not 100% correct

Biword indices II

- Not all phrases are proper nouns ("Edmond Dantes")
  - *negotiation of the treaty*
  - *the year of the rabbit*

- Part-of-speech tagging labels words according to their lexical categories

The negotiation of the treaty took many years.

DT/ The NN/ negotiation IN/ of DT/ the NN/ treaty VBD/ took JJ/ many NNS/ years ./ .

singular noun

verb

Try it yourself (demo): http://cogcomp.cs.illinois.edu/demo/pos/

Stanford POS Tagger http://nlp.stanford.edu/software/tagger.shtml

Biword indices III

cost overruns on a power plant
“cost overruns” AND “overruns power” AND “power plant”

The negotiation of the treaty took many years.

DT/ The NN/ negotiation IN/ of DT/ the NN/ treaty VBD/ took JJ/ many NNS/ years ./ .

DT/ The NN/ negotiation IN/ of DT/ the NN/ treaty VBD/ took JJ/ many NNS/ years ./ .

negotiation of the treaty

N X X N

Extended biword: any string of the form **NX*N** is a vocabulary term

Query: needs to be POS-tagged & converted to extended biwords!

Biword indices IV

- This concept can be extended to longer sequences (phrase indices)
- Single term queries are not handled naturally in biword indices (entire index scan is necessary)
    - Add a single-term index

- Arbitrary phrases are usually not indexed
    - Vocabulary sizes increases greatly

	Vocabulary size
Single term index	19,236
Biword index	866,914
Triword index	6,425,444

The Count of Monte Cristo
~50,000 lines of text

Positional indices

- Most common index type
- For each term postings are stored with frequency values

to, 993427:
⟨ 1, 6: ⟨7, 18, 33, 72, 86, 231⟩;
2, 5: ⟨1, 17, 74, 222, 255⟩;
4, 5: ⟨8, 16, 190, 429, 433⟩;
5, 2: ⟨363, 367⟩;
7, 3: ⟨13, 23, 191⟩; . . . ⟩

to occurs 993,427 times in the corpus

to occurs 6 times in document 1

to occurs at positions 7, 18, 33 …

be, 178239:
⟨ 1, 2: ⟨17, 25⟩;
4, 5: ⟨17, 191, 291, 430, 434⟩;
5, 3: ⟨14, 19, 101⟩; . . . ⟩

Source: [1]

Positional indices

Querying the inverted index

- To process a phrase query: "to be or not to be"
  - Access the inverted list for each term
  - When merging (intersecting) the result list, check if the positions of the terms match the phrase query
    - Calculate offsets between words
    - Start with the least frequent query term

- Index size increases (positions need to be stored)
  - Between 2-4 times larger than a non-positional index

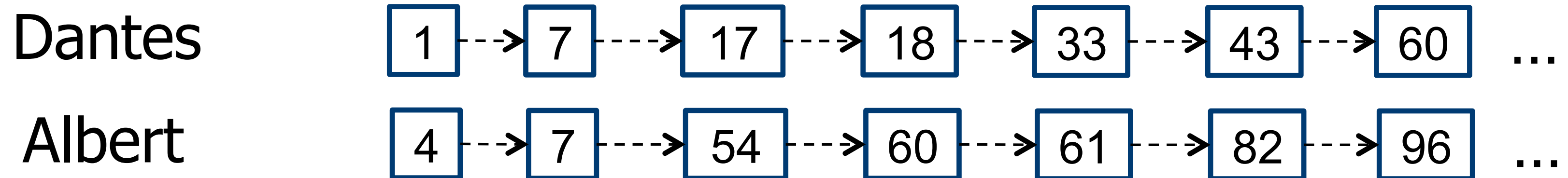
Combining biword and positional indices

- For common biword queries (e.g. "Britney Spears") it is inefficient to keep merging positional lists
- Idea: use a biword index for certain queries and a positional index for all others
  - Most expensive are those queries where the individual words are common ("The Who"); having those in a biword index yields considerable speed-ups
  - What queries to execute on the biword index can be learned from looking at the query log

Query time	userID	query	itemRank	clickURL
02.03.2010 04:15:03	23543535	chess software	5	http://www.chessbase.com/
02.03.2010 04:15:15	23543535		9	http://en.wikipedia.org/wiki/Computer_chess
02.03.2010 04:23:15	1243232	britney spears	1	http://www.britneyspears.com/
02.03.2010 05:06:15	53443223	hadoop cygwin	4	http://wiki.apache.org/hadoop/FAQ

Vocabulary lookup I

1) Determine whether both query terms exist
2) Locate pointers to the respective posting lists

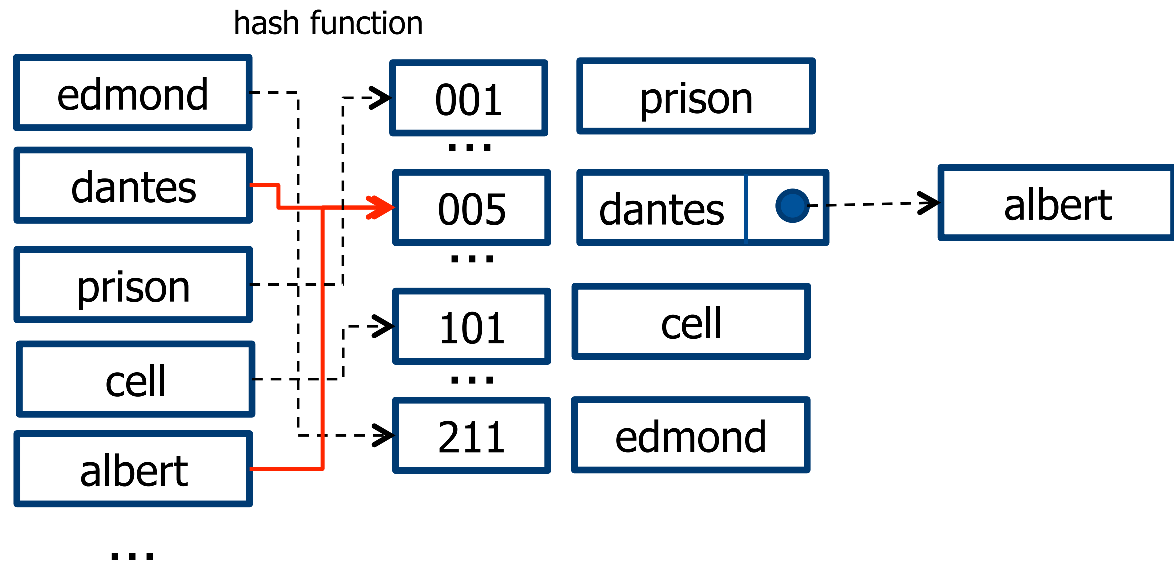


- Implementation options: **hashes** and **search trees**
  - Choice depends on
    - Number of terms (keys)
    - Frequency of and type of changes (key insert/delete) in the index
    - Frequency of key accesses

Vocabulary lookup II

Hashing

- Each vocabulary term is hashed into an integer (avoid hash collisions if possible)



- Unable to react to slight differences in query terms (e.g. *Dantes* vs. *Dantès*)
- Unable to seek for all terms with a particular prefix (e.g. *Dant)*

- Querying: hash each query term separately, follow pointer to corresponding postings list

Vocabulary lookup III

Binary search trees overcome many of the hashing disadvantages

root

a-m

n-z

a-hu

hv-m

n-sh

si-z

a-de

df-hu

....

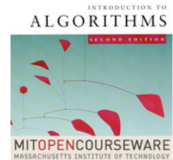
....

....

aardvark

zygot

Introduction to Algorithms. Cormen et al., 2009.

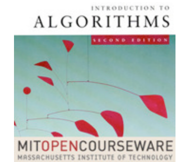


- For efficient search, the tree needs to be balanced ($log(|V|)$ comparisons)
- When terms are added/deleted from the tree, **rebalancing** is required

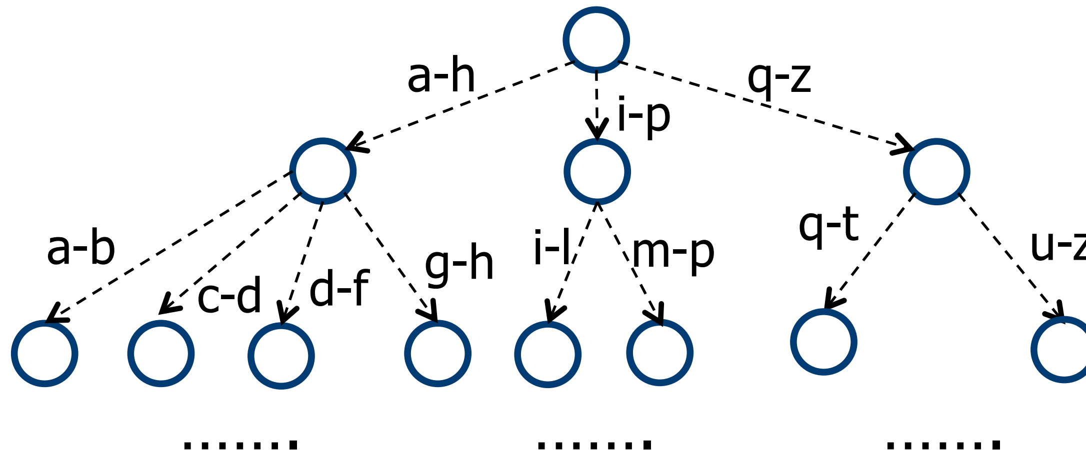
Vocabulary lookup IV

B-trees

Introduction to Algorithms.
Cormen et al., 2009.



- Commonly used for dictionaries
- Number of sub-trees in an internal node varies in a fixed interval (e.g. [2,4]), leading to less frequent rebalancing
- All leaf nodes are at the same depth



Wildcard queries I

- Commonly employed when
  - There is uncertainty about the spelling of a term (Dantes vs. Dantès)
  - Multiple spelling variants of a term exist (labour vs. labor)
  - All terms with the same stem are sought (restoration and restore)
- Trailing wildcard query: *restor**
  - Search trees are perfect in such situations: walk along the edges and enumerate the W terms with prefix *restor*; followed by $|W|$ lookups of the respective posting lists to retrieve all docIDs

Wildcard queries II

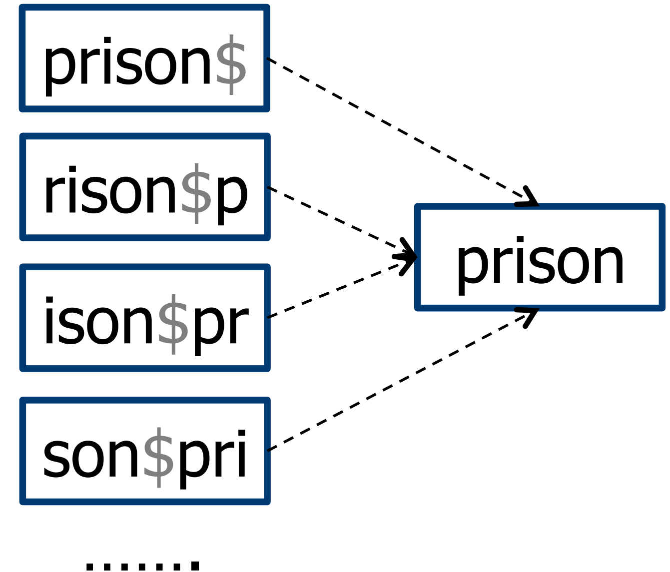
- Leading wildcard query: **building* (building vs. rebuilding)
  - **Reverse** dictionary B-tree: constructed by reading each term in the vocabulary **backwards**
  - Solved analogously to the trailing wildcard query on a b-tree
    - reverse b-tree is traversed with *building backwards: g-n-i-d-l-i-u-b

- Single wildcard query: analy*ed (analysed vs. analyzed)
  - Traverse the regular b-tree to find the W terms with prefix *analy*
  - Traverse the reverse b-tree to find the R terms with suffix *ed*
  - Final result:intersect W and R

General wildcard queries I

Permuterm index

- Query *pr*son*➔*pr*son$*
  - Move * to the end: *son$pr**
  - Look up the term in the permuterm index (search tree)
  - Look up the found terms in the standard inverted index
- Query *pr*s*n*
  - Start with *n$pr**
  - Filter out all results not containing 's' in the middle (exhaustive)
  - Look up the found terms in the standard inverted index



Dictionary increases substantially in size!!

General wildcard queries II

N-gram index

Beginning/end of term character

- N-gram: sequence of *N* characters
  - 3-grams of prison: *$pr, pri, ris, iso, son, on$*
  - 4-grams of prison: *$pri, pris, riso, ison, son$*
- N-gram index: contains all N-grams that occur in any of the terms

	Vocabulary size
3-gram index	5,885
Single term index (term)	19,236
4-gram index	22,264
Biword index (term)	866,914
Triword index (term)	6,425,444

The Count of Monte Cristo
~50,000 lines of text

General wildcard queries II

N-gram index

- each N-gram in the dictionary points to all terms containing the N-gram

lexicographical ordering

- Wildcard query: *pr*on*
  - Boolean query *$pr AND on$*
  - Look up in a 3-gram index yields a list of matching terms
  - Look up the matching terms in a standard inverted index
- Wildcard query: red*
  - Boolean query *$re AND red* (also retrieves *retired*)
  - Post-filtering step to ensure enumerated terms match red*

General wildcard queries III

- Processing of wildcard queries is expensive
- Added lookup in the special index, filtering and finally the lookup in the standard inverted index

Spelling correction I

```
488941 britney spears
 40134 brittany spears
 36315 brittney spears
 24342 britany spears
  7331 britny spears
  6633 briteny spears
  2696 britteny spears
  1807 briney spears
  1635 brittny spears
  1479 brintey spears
  1479 britanny spears
  1338 britiny spears
  1211 britnet spears
  1096 britiney spears
```

http://www.google.com/jobs/britney.html

- **Isolated-term** correction considers each query term individually
- **Context-sensitive** correction
  - *animals form Australia* is corrected to *animals **from** Australia*

Spelling correction

Edit distance

- **Levenshtein distance** between strings *s1* and *s2*: number of operations to transform *s1* into *s2*
  - Insert a character (hod ➔ hood)
  - Delete a character (brittney ➔ britney)
  - Replace a character (analyzis ➔ analysis)

rise ➔prison
0. rise
1. prise [insertion]
2. priso [substitution]
3. prison [insertion]

foot➔fast
0. foot
1. faot [substitution]
2. fast [substitution]

TU Delft

Spelling correction

Edit distance

- **Levenshtein distance** between strings *s1* and *s2*: number of operations to transform *s1* into *s2*
  - Insert a character (hod ➔ hood)
  - Delete a character (brittney ➔ britney)
  - Replace a character (analyzis ➔ analysis)

(i,j) contains the edit distance of the first *i* chars of *s1* and the first *j* chars of *s2*

		p	r	i	s	o	n
	0	1	2	3	4	5	6
r	1						
i	2						
s	3						
e	4						

Spelling correction

Edit distance

- **Levenshtein distance** between strings *s1* and *s2*: number of operations to transform *s1* into *s2*
    - Insert a character (hod ➔ hood)
    - Delete a character (brittney ➔ britney)
    - Replace a character (analyzis ➔ analysis)

		p	r	i	s	o	n
	0	1	2	3	4	5	6
r	1	1	1				
i	2	2					
s	3	3					
e	4	4					

$d(i,j)=d(i-1,j-1)$ *if* $s1[i]==s2[j]$

Spelling correction

Edit distance

- **Levenshtein distance** between strings *s1* and *s2*: number of operations to transform *s1* into *s2*
  - Insert a character (hod ➔ hood)
  - Delete a character (brittney ➔ britney)
  - Replace a character (analyzis ➔ analysis)

		p	r	i			n
	0	1	2	3			6
r	1	1	1				
i	2	2	**2**				
s	3	3					
e	4	4					

Minimum of
(i-1,j)+1,
(i-1,j-1)+1,
(i,j-1)+1

Spelling correction

Edit distance

- **Levenshtein distance** between strings *s1* and *s2*: number of operations to transform *s1* into *s2*
  - Insert a character (hod ➔ hood)
  - Delete a character (brittney ➔ britney)
  - Replace a character (analyzis ➔ analysis)

		p	r	i	s	o	n
	0	1	2	3	4	5	6
r	1	1	1	2	3	4	5
i	2	2	**2**	1	2	3	4
s	3	3	3	2	1	2	3
e	4	4	4	3	2	2	**3**

edit distance

Spelling correction III

Edit distance: how to apply in practice

- Naïve approach
  - Edit distance between query terms and all terms in the dictionary (vocabulary) are calculated
  - The most similar (smallest distance) terms are considered as spelling correction
  - Computationally too expensive
- Heuristics
  - Restrict search to vocabulary terms with the same starting letter

Spelling correction IV

Context sensitive spelling correction

- If a query phrase yields a small set of retrieved documents, search engines often offer potential corrections
  - *animals form Australia* is corrected to *animals* ***from*** *Australia*

- Approach
  - Enumerate all possible corrections of each query term
  - Substitute each correction into the phrase
  - Run a query against the index, find number of matching documents
  - Offer most common phrasings

```
    8 animals form australia
    6 animal form australia
    0 animal form austria
  155 animal from austria
 3850 animals from austria
55500 animals from australia
```

#Google hits

Spelling correction V

Context sensitive spelling correction

- Approach
  - Enumerate all possible corrections of each query term
  - Substitute each correction into the phrase
  - Run a query against the index, find number of matching documents
  - Offer most common phrasings
  - Can be very expensive!

- Heuristics
  - Retain only the most common combinations in the documents or query log (query reformulations)

Phonetic correction I

Soundex algorithm

- Misspelled queries that sound like the target term
  - Mostly applicable to proper nouns, in particular people's names (may be spelled differently in different countries)

- Idea: phonetic hashing
  - Similar sounding terms hash to the same value

- Soundex algorithm
  1. Turn every term to be indexed into a reduced form with 4 characters; build an inverted index from these reduced terms (soundex index)
  2. Apply the same to the query terms before searching the index

  (when a query contains a term from the soundex index, expand the query to include all variations an search in the standard inverted index)

Phonetic correction II

Soundex algorithm

- Reducing terms to 4 characters
    1. Keep the first letter of the term
    2. Change letters to digits as follows (vowels are ignored)
        a. b,v,p,v ➔ 1
        b. c,g,j,k,q,s,x,z ➔ 2
        c. d,t ➔ 3
        d. l ➔ 4
        e. m,n ➔ 5
        f. r ➔ 6
    3. Consecutive identical digits are reduced to one
    4. If there are less than three digits after the conversion, pad with '0' values (e.g. last name "Lee" is reduced to *L000*)

```
Levenshtein
L 1 52 3  5
L152

Levensjtejn
L 1 5223 25
L152
```

- Based on phonetic observations (language dependent)
    - Vowels are interchangeably
    - Consonants with similar sounds are in equivalence classes

Summary

- Indexing is not a trivial task
- Many data structures and algorithms exist
  - A lot depends on the type of queries that should be supported by the search system
- This high-level overview is followed by a bit more implementation-oriented lecture on Wednesday

Sources

1. Introduction to Information Retrieval. Manning et al., 2008.
2. Managing gigabytes. Witten et al., 1999.
3. http://tartarus.org/~martin/PorterStemmer/
4. http://tartarus.org/martin/PorterStemmer/def.txt