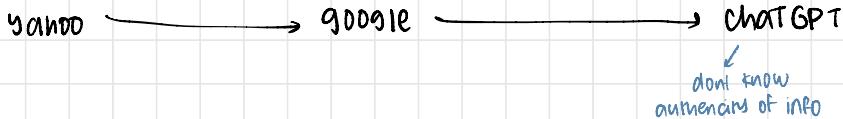
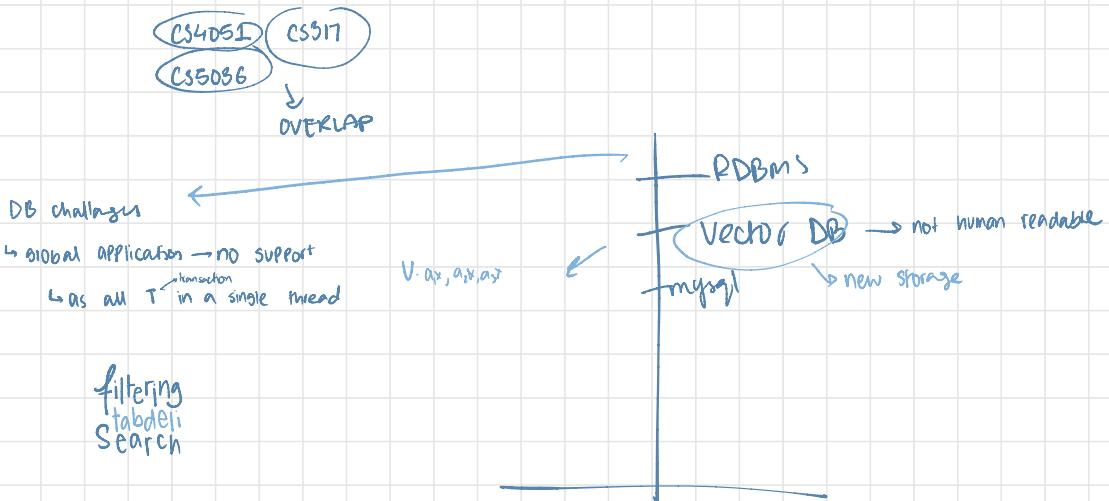


Information Retrieval

Muhammad Rafi - 5 years



↳ data collected by some DB management system

Database Systems

↳ Structured format → clear semantics based on a formal model

↳ Formal queries → SQL

↳ exact results

↳ RDBMS → specific architecture, language

↳ able to diagnose conceptual schema

Information Retrieval

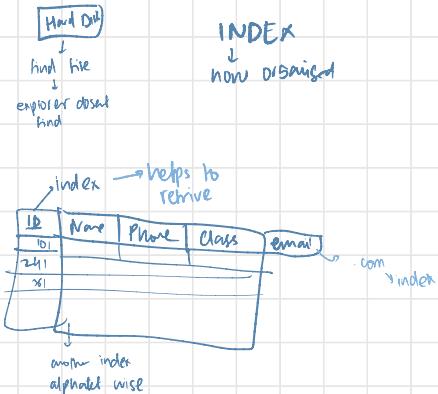
↳ Semi/Unstructured → free text

↳ Natural language → keyword search

↳ sometimes relevant result

↳ You have documents

↳ try to build interface on that



Data mining

↳ analysis step of KDD
knowledge discovery in DB

↳ extracts info from data set and

↳ transforms to an understandable structure

VS

Text mining

↳ exploits info contained in textual documents

in many ways

↳ discovery of patterns and trends

↳ associations among entities

↳ predictive rules

↳ to answer for questions for which the answer is not currently known

Text Vs. Data		
	Data Mining	Text Mining
Data Object	Numerical & categorical data	Textual data
Data structure	Structured	Unstructured & semi-structure
Data representation	Straightforward	Complex
Space dimension	< tens of thousands	> tens of thousands
Methods	Data analysis, machine learning statistic, neural networks	Data mining, information retrieval, NLP,...
Maturity	Broad implementation since 1994	Broad implementation starting 2000
Market	10 ³ analysts at large and mid size companies	10 ³ analysts corporate workers and individual users

natural language processing

Information Retrieval

VS

Text mining

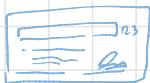
↳ Gets info related to a user query
from unstructured collections of textual data

↳ Activity: crawling, parsing, indexing
evaluating info using generalized methods

↳ finding unstructured relevant info

text is considered
unstructured here

→ e.g. getting credit card no



what is your CNIC NO.? → query

C → look up

Adhoc Retrieval Systems

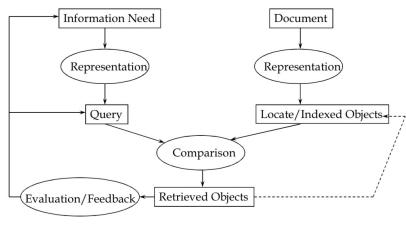
■ It is a system that aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query.

■ Example

- Google
- Bing (later Decision engine)
- Assignment no 1

Westlaw

Basic Information Retrieval Process



Content wise IR Systems

Contents

- ❑ Size
 - Personal / Desktop (Spotlight, Instant Search, anywhere)
 - Web Scale (Google, DuckDuckgo)
 - Enterprise Search (WestLaw, PolicyBazar)
- ❑ Static (Offline) or Dynamic (Online)
- ❑ Type
 - Text
 - Multimedia
 - Mixed
- ❑ Exact Match vs. Best Match

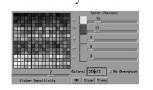
Dimension of IR

Content	Applications	Tasks
Text	Web search	Ad hoc search
Images	Vertical search	Filtering
Video	Enterprise search	Classification
Scanned docs	Desktop search	Question answering
Audio	Forum search	Clustering
Music	P2P search	Literature search

IBM's QBIC

- QBIC – Query by Image Content
- First commercial CBIR system.
- Model system – influenced many others.
- Uses color, texture, shape features
- Text-based search can also be combined.
- Uses R*-trees for indexing

QBIC – Search by color



** Images courtesy : Yong Rao

QBIC – Search by shape



** Images courtesy : Yong Rao

QBIC – Query by sketch

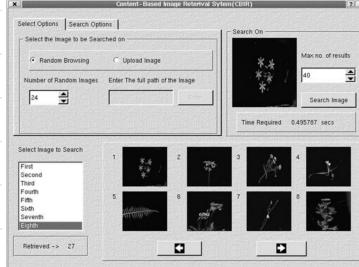
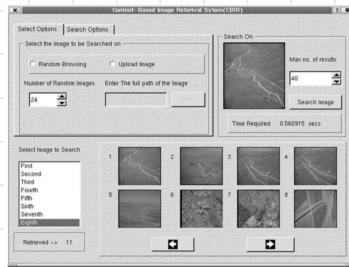


** Images courtesy : Yong Rao

Virage

- Developed by Virage inc.
- Like QBIC, supports queries based on color, layout, texture
- Supports arbitrary combinations of these features with weights attached to each
- This gives users more control over the search process

Search



VisualSEEk

- Research prototype – University of Columbia
- Mainly different because it considers spatial relationships between objects.
- Global features like mean color, color histogram can give many false positives
- Matching spatial relationships between objects and visual features together result in a powerful search.

Music Search Engine

- Search by metadata likes: artists biography, music reviews, new releases, concerts dates
- Search for Lyrics: Lyrics.com, Allmusic.com, musicpedia.com and SearchLyrics.com
- Recommend Similar Music: based on seed elements (artist, track) users are recommended similar tracks or artists.
- User query by Humming
- Generate Playlists: automatic generation of playlists, that satisfy user constraints

Mathematical Search Engines

- The mathematical contents on the web is continuously on rise. The IR perspective of contents are quite low.
- We need to redesign a specialized search engine for it.
- There are few very good attempts:
 - www.wolframalpha.com
 - symbolab.com
 - searchenginewatch.com

VisualSEEk

- Research prototype – University of Columbia
- Mainly different because it considers spatial relationships between objects.
- Global features like mean color, color histogram can give many false positives
- Matching spatial relationships between objects and visual features together result in a powerful search.

Example: Content-based Image Retrieval:

□ <http://wang.ist.psu.edu/IMAGE>



General Search Engines

- www.google.com
- www.visimo.com
- www.clusty.com
- www.bing.com
- www.yahoo.com

Bing

- MSN Search
- Microsoft Search
- Live Search
- Bing
 - ASP.NET Launched June 01, 2009 supported 40 languages now
 - Bing and Decide was the slogan in 2009
 - Bing a decision engine - 2010
 - "Bing is for doing" is in 2012

BOOLEAN RETRIEVAL

MODEL

↳ an abstract representation of a process/object

↳ used

↳ to study properties

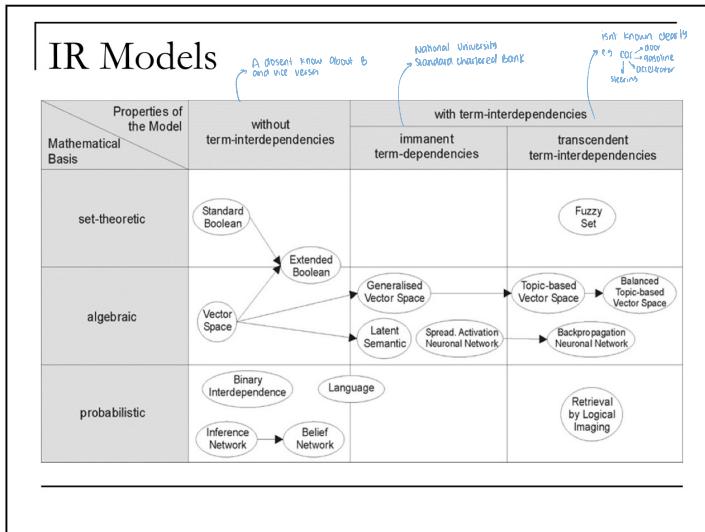
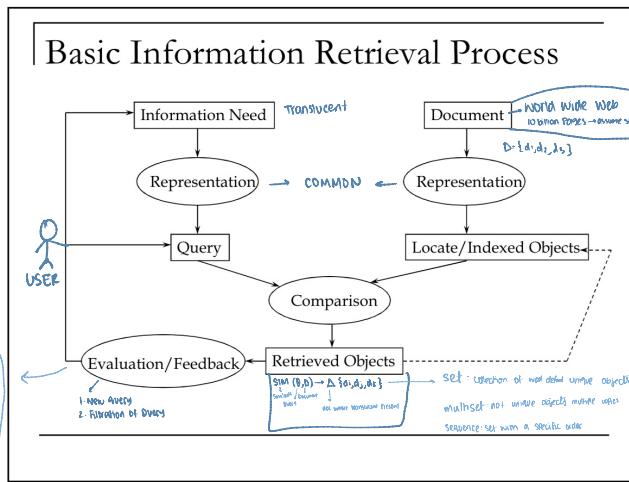
↳ draw conclusions
↳ quality of conclusions depends how much the model is close to reality

↳ make predictions

Retrieval Model

↳ describes the human and computational process involved in ad-hoc retrieval

- Example: A model of human information seeking behavior
- Example: A model of how documents are ranked computationally
- Components: Users, information needs, queries, documents, relevance assessments,
- Retrieval models define relevance, explicitly or implicitly



Retrieval Complexities

Basic Information Retrieval Setting

- Offline / Online
 - Corpus → many diff documents
 - Dynamic / changing collection (www)
- Scale of operation
 - World wide web level → big search → fast, from file explorer
 - Personal / individual level → bounded size
 - Enterprise / domain specific level → diff format, files
 - very specific kind of data
 - size based on collection
- Type of Content
 - Text/multimedia/ mixed content

Sep. 1.1

Unstructured data in 1680

- Shakespeare's work
 - Lets you have all his plays in soft copy with you, in simple text format.
 - 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**?

Example

- Human Method
 - Read the book from start to end, if any play contains Brutus AND Caesar and NOT Calpurnia then the title is returned as result.
- Machine Method
 - Read each file sequentially for Brutus AND Caesar and NOT Calpurnia, maintain a boolean status for each play. Check for play in which Brutus = 1 AND Caesar=1 and Calpurnia=0
 - Problems:
 - Lets discuss

SHAKESPEARE



↓
grep
can tell which files
have Brutus, Caesar
then subtract Calpurnia

Boolean Model CONS

- ↳ no relevance feedback
- ↳ no ranking
- ↳ complex query formation
- ↳ no exact matching
- ↳ sparse matrix →
 - too many null values
 - too large for small data set

Term Matrix Document

- ↳ counts
- ↳ add Brutus, Caesar
- ↳ complement Calpurnia
- ↳ add both

Problems

- ↳ bad for dynamic
 - if uses changes on the go → changing dim
as dim + term both increase
- ↳ sparse matrix
 - too many null values
too large for small data set
- ↳ Exact matching → no best match
- ↳ complex query formulation
- ↳ not efficient in memory consumption

PRO

- ↳ good for non changing dim

Term-Document Matrix

	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	1	0	0	0	
mercy	1	0	1	1	1	1	
wosier	1	0	1	1	1	1	

Figure I.1 A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t , and 0 otherwise.

Answer to the Query

To answer the query Brutus AND Caesar AND NOT Calpurnia, we take the vectors for Brutus, Caesar and Calpurnia, complement the last, and then do a bitwise AND:

$$110100 \text{ AND } 110111 \text{ AND } 101111 = 100100$$

The answers for this query are thus Antony and Cleopatra and Hamlet

THIS IS THE PROBLEM

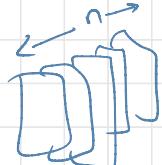
SOLUTION

↳ INVERTED INDEX

↳ POSTING

decide which one goes first

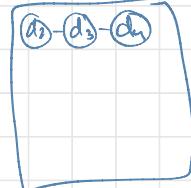
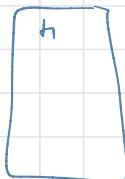
$$\begin{matrix} \uparrow & \uparrow \\ \neg (w_1 \text{ AND } w_2) \text{ OR } (w_3 \text{ AND } w_4) \\ \text{AND} (w_5 \text{ OR } w_6) \end{matrix}$$



→ PRDS

↳ DYNAMIC

↳ PASS TO PROCESS



term frequency
3

document frequency
2

BOOLEAN MODEL

$D = \{d_1, d_2, \dots, d_n\}$

$|D| = n$ is a large number

$d_i = \{t_1, \dots, t_m\}$

$m \lll$ small number compared to n

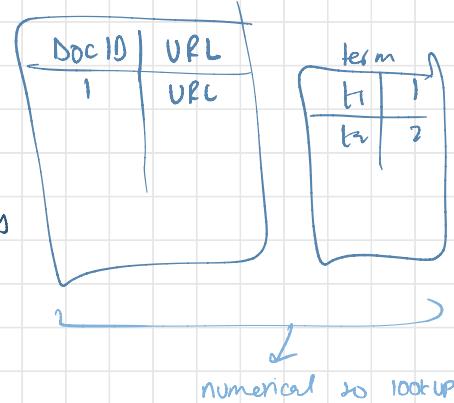
e.g. $D = 1$ million

document size $\leftarrow |d_i| = 5000 \rightarrow$ distinct: 1000 \times million
 \downarrow
term size $\leftarrow |t_i| = 500$ 0.5 billion
 0.1 billion

Term Doc Matrix $\rightarrow 2 \text{ dim}$
 $t_1 \quad d_1 \quad d_2 \dots \dots \dots \rightarrow$ bigger
 \downarrow
smaller
 t_2
 \vdots

80% sparsity

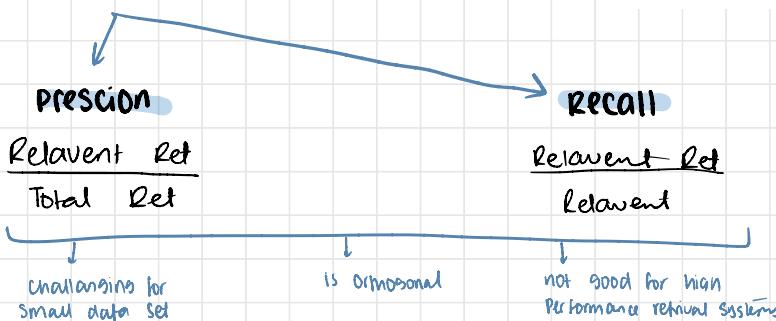
URL (http://Users/test/1.html)



Effectiveness of IR System

↳ based on quality of its search results

↳ TWO factors



100% Precision → gives all data set

100% Recall → Retire 1 document that is also relevant

	Relevant	Irrelevant
Retrieved	Rel+Ret	Irrel+Ret
Non Retrieved	Non-Rel + Rel	Non-Rel + Irrel

Precision = relevant -retrieved / total retrieved

Recall = relevant -retrieved / total Relevant in collection

Example of IR Evaluation

- An IR system returns 8 relevant documents, and 10 irrelevant documents. There are total of 20 relevant documents in the collection. What is the precision of the system, on this search, and what is its recall?

precision = $8/18 = 0.44$
recall = $8/20 = 0.4$

Example:

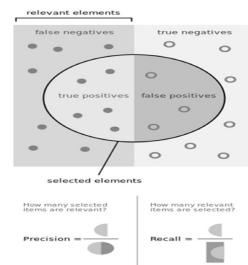


Image source :
Wikipedia

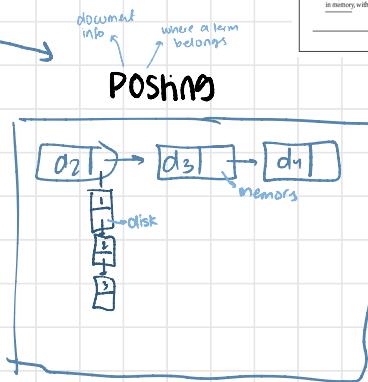
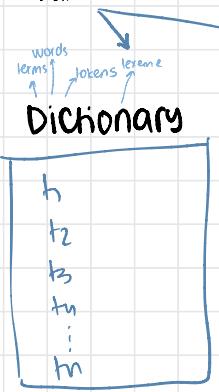
Q) Rel : 20
Ret = 8
IRR = 10
Precision : $\frac{8}{10+8} = 0.44$

Recall : $\frac{8}{20} = 0.4$

Inverted Index

↳ index of vocabulary that occurs in a collection

↳ has 2 parts



Inverted Index	
Brutus	→ [1 2 4 11 31 45 197 194]
Cesar	→ [1 2 4 5 6 16 39 132 ...]
Caesar	→ [2 31 54 101]
⋮	
Dictionary Postings	

Figure 13 The two parts of an inverted index. The dictionary is commonly kept in memory with pointers to each posting list, which is stored on disk.

PROS

↳ less sparsity

↳ quick access

↳ uses NLP to solve problem of complex query formation

Example

Doc 1
I did enact Julius Caesar. I was killed if'the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.

term	docID	term	docID	term	docID	term	freq	→	postings lists
did	1	ambitious	1	be	1	brutus	2	→	[2 2]
enact	1	be	1	brutus	1	capitol	1	→	[1 1]
hus	1	brutus	1	capitol	1	caesar	2	→	[1 2]
caesar	1	capitol	1	caesar	1	caesar	2	→	[1 1]
I	1	caesar	1	did	1	caesar	2	→	[1 1]
was	1	caesar	2	enact	1	caesar	2	→	[1 1]
killed	1	caesar	2	hath	1	caesar	2	→	[2 2]
1	1	did	1	hath	1	caesar	2	→	[2 2]
the	1	enact	1	hath	1	caesar	2	→	[2 2]
the	1	hath	1	it	1	caesar	2	→	[2 2]
capitol	1	hath	1	it	1	caesar	2	→	[2 2]
brutus	1	it	1	it	1	caesar	2	→	[2 2]
killed	1	it	1	let	1	caesar	2	→	[2 2]
me	1	it	1	killed	1	caesar	2	→	[2 2]
I'	1	it	1	me	1	caesar	2	→	[2 2]
me	1	it	1	noble	1	caesar	2	→	[2 2]
the	1	it	1	so	1	caesar	2	→	[2 2]
the	1	it	1	the	2	caesar	2	→	[2 2]
the	1	it	1	the	2	caesar	2	→	[2 2]
the	1	it	1	you	1	caesar	2	→	[2 2]
the	1	it	1	you	2	caesar	2	→	[2 2]
told	2	it	1	was	2	caesar	2	→	[2 2]
you	2	it	1	with	1	caesar	2	→	[2 2]
caesar	2	it	1			ambitious	2	→	
the	2	it	1			ambitious	2	→	
brutus	2	it	1			ambitious	2	→	
bath	2	it	1			ambitious	2	→	
told	2	it	1			ambitious	2	→	
you	2	it	1			ambitious	2	→	
caesar	2	it	1			ambitious	2	→	
the	2	it	1			ambitious	2	→	
was	2	it	1			ambitious	2	→	
was	2	it	1			ambitious	2	→	
ambitious	2	it	1			ambitious	2	→	



ceaser tree

Term freq = 3 → mentioned 3 times in all document

Document freq = 2 → mentioned in 2 documents

Processing Boolean Queries

- Processing Boolean Queries of the form, for example
 - T1 OR T2
 - T1 AND T2
 - T2 AND T3 OR (T4 OR NOT T5)
- Example Query
 - Brutus AND Calpurnia

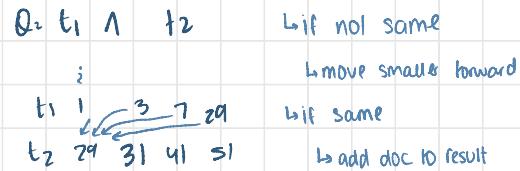
Processing Boolean Queries

- Example Query
 - Brutus AND Calpurnia
 - Locate "Brutus" in the Dictionary
 - Retrieve its Postings
 - Locate "Calpurnia" in the Dictionary
 - Retrieve its Postings
 - Get the intersection of the two posting lists

Brutus → [1]—[2]—[4]—[11]—[31]—[45]—[173]—[174]
Calpurnia → [2]—[31]—[54]—[101]
Intersection ⇒ [2]—[31]

Intersect (Posting List Algorithm)

```
INTERSECT( $p_1, p_2$ )
1  $answer \leftarrow \langle \rangle$ 
2 while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3 do if  $docID(p_1) = docID(p_2)$ 
4     then ADD( $answer, docID(p_1)$ )
5          $p_1 \leftarrow next(p_1)$ 
6          $p_2 \leftarrow next(p_2)$ 
7     else if  $docID(p_1) < docID(p_2)$ 
8         then  $p_1 \leftarrow next(p_1)$ 
9     else  $p_2 \leftarrow next(p_2)$ 
10 return  $answer$ 
```



Good for
big, single term query

Query Optimization

↳ Organise query such that least amount of work need to be done by the system

For Boolean Queries: order in which Postings lists are accessed

■ What is the best order for query processing?

□ Brutus AND Caesar AND Calpurnia

- For each of the t terms, we need to get its postings, then AND them together.
- The standard heuristic is to process terms in order of increasing document frequency: if we start by intersecting the two smallest postings lists, then all intermediate results must be no bigger than the smallest postings list, and we are therefore likely to do the least amount of total work.

Intersect (Optimized)

```
INTERSECT( $\{t_1, \dots, t_n\}$ )
1 terms  $\leftarrow$  SORTBYINCREASINGFREQUENCY( $\{t_1, \dots, t_n\}$ )
2 result  $\leftarrow$  postings(first(terms))
3 terms  $\leftarrow$  rest(terms)
4 while terms  $\neq$  NIL and result  $\neq$  NIL
5 do result  $\leftarrow$  INTERSECT(result, postings(first(terms)))
6 terms  $\leftarrow$  rest(terms)
7 return result
```

Query Processing total work

↳ Parse query

↳ Locate term in dictionary

↳ Collect Postings list

↳ Intersection algo

↳ Sort by increasing freq for optimization

term | doc ID term | doc ID term doc freq → Postings

term	doc ID
t ₁	1, 2, 3, 4, 5
t ₂	2, 3, 4, 5
t ₃	3, 4, 5

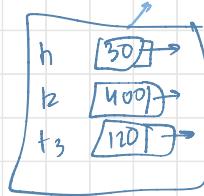
→ SORTED

term	doc ID
t ₁	1, 2, 3, 4, 5
t ₂	2, 3, 4, 5
t ₃	3, 4, 5

term doc freq → Postings

cardinality → no of times came in doc

↳ t₁ ∩ t₂ ∩ t₃
↳ t₁ ∩ t₃ ∩ t₂
↓
less frequency
so better



HOW TO OPTIMIZE QUERY
↳ choose lesser frequency

BOOLEAN MODEL

↳ most simple and used model

↳ It considers that the document contains

↳ features
↳ WORDS
↳ PHRASES
↳ SEQUENCES

↳ user's query is about these features

ADV

↳ clean formation, easy to implement we know Boolean algebra

↳ results are predictable and easily explainable

↳ many diff features can be incorporated

DOCUMENTS: set of terms

QUERIES: Boolean expression on terms

CONS

↳ Exact matching if you don't know the exact word then problem e.g. searchs 'smart' and not getting query for 'intelligent' or 'stamina'

↳ Query formation is hard into has to be translated into Boolean expression

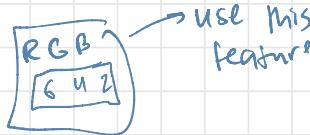
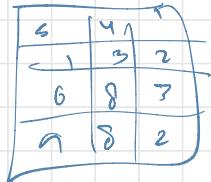
↳ All terms are equally weighted

↳ flat results

↳ returns either too few or too many doc

IMAGES

130x150



Example

- Consider the following document collection:
D1 = "Dogs eat the same things that cats eat"
D2 = "no dog is a mouse"

D3 = "mice eat little things"

D4 = "Cats often play with rats and mice"

D5 = "cats often play, but not with other cats"

Indexed by:

- D1 = dog, eat, cat
- D2 = dog, mouse
- D3 = mouse, eat
- D4 = cat, play, rat, mouse
- D5 = cat, play

Example

Queries

- Query Q1: (cat AND dog) returns D1
- Query Q2: (cat AND mouse) returns D4
- Query Q3: (cat OR play) returns (D4, D5)
- Query Q4: (cat AND play AND mouse) returns D4
- Query Q5: ((cat AND Play) OR (Cat AND dog)) returns (D1, D4, D5)
- (cat OR (dog AND eat)) returns (D1, D4, D5)

Example

Example 1.1: Commercial Boolean searching: Westlaw. Westlaw (<http://www.westlaw.com>) is the largest commercial legal search service (over 10 million users) performing millions of searches per day over tens of terabytes of test data. The service was started in 1975. In 2005, Boolean search (called "Terms and Connectors" by Westlaw) was still the default, and used by a large percentage of users, although ranked free text querying (called "Natural Language" by Westlaw) was added in 1992. Here are some example Boolean queries on Westlaw:

Information need: Information on the legal theories involved in preventing the disclosure of trade secrets by employees formerly employed by a competing company. Query: "trade secret" /s disclose /s prevent /s employee

Information need: Requirements for disabled people to be able to access a workplace. Query: disable! /p access! /s work-site work-place (employment /3 place)

Information need: Cases about a host's responsibility for drunk guests. Query: host! /p (responsib! liab!) /p (intoxicat! drunk!) /p guest

TERM VOCAB & POSTING LISTS

Boolean Model

- Information need has to be translated into a Boolean expression which most users find awkward
- The Boolean queries formulated by the users are most often too simplistic
- The Boolean model imposes a binary criterion for deciding relevance
- The question of how to extend the Boolean model to accommodate partial matching and a ranking has attracted considerable attention in the past
- Two extensions of boolean model:
 - Extended Boolean Model
 - Fuzzy Set Model

Westlaw

Westlaw – Commercial Systems

- Largest commercial legal search service in terms of the number of paying subscribers
- Over half a million subscribers performing millions of searches a day over tens of terabytes of text data
- The service was started in 1975.
- In 2005, Boolean search (called "Terms and Connectors" by Westlaw) was still the default, and used by a large percentage of users . . .
- . . . although ranked retrieval has been available since 1992.
- Information need: Information on the legal theories involved in preventing the disclosure of trade secrets by employees formerly employed by a competing company
- Query: "trade secret" /s disclos! /s prevent /s employ!

Extended Boolean Model

Extended Boolean Model

- Proximity Search
- Ranked Retrieval
- Example

□ WestLaw

Westlaw

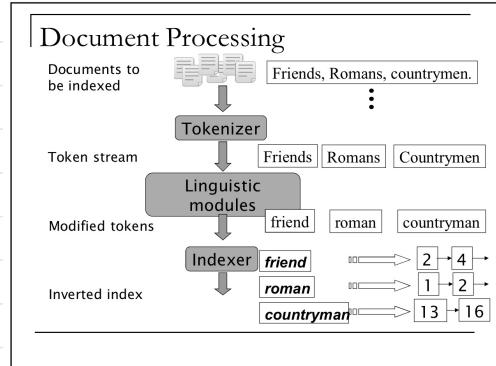
Pros

↳ weighted → freq. of term as a weight
↳ final results → gives ranks

Boolean Retrieval Model

- Last Chapter: Simple Boolean retrieval system
- Our assumptions were:
 - We know what a document is.
 - Documents are only the collection of features.
 - We can "machine-read" each document.
- This can be complex in reality.

Document Processing



in b/w a space

Words: delimited string of characters

Type: same meaning diff word

↳ an equivalence class of tokens

Language: character set = {a,b,...,z}, {A,B,...,Z}, {!, ?, ., , -}

Term: normalized words

↳ an equivalence class of words

Token: instance of word/term

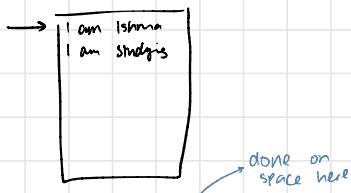
e.g. word token, term token
token may not be your feature



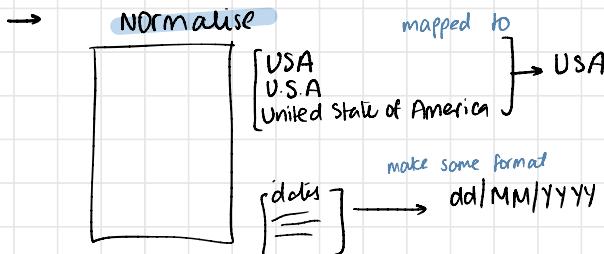
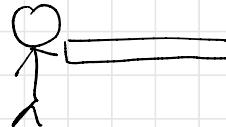
Equivalence class: prefix, suffix of words

e.g. operati, operation, operator

DOCUMENT PROCESSING



→ words: Tokenization(file, "----")
: I, am, Ishma, I, am, studying



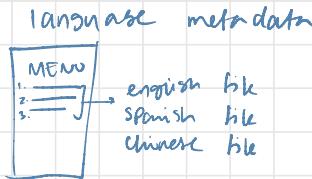
Morphological changes
prefix → add suffix

Noisy runs
run

CHALLENGES IN DOC PROCESSING

↳ what format: PDF/Word/Excel/HTML

↳ what language



↳ what character set: UTF-8/CP1252

↳ Format/Language-Encoding

↳ Documents

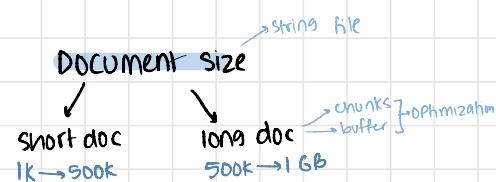
↳ size of document: a file/email/blog/group of files

↳ Tokenization

↳ issues in tokenization

linearization → chop in single/multi byte

classification
Problems



TOKENIZATION

↳ a process through which docs are parsed and a sequence of characters separated

- Tokenization process decide when to emit a token.
- Input: "Friends, Romans and Countrymen"
- Output: Tokens
 - Friends
 - Romans
 - Countrymen

Issues in Tokenization

- Finland's capital
- Hewlett-Packard → 1 word or 2?
- co-education
- San Francisco: one token or two?
- Numbers
 - 3/20/91
 - Mar. 12, 1991
 - 55 B.C.
 - B-52
 - (800) 234-2333
- Languages
 - French
 - German
 - Urdu & Arabic
 - Korean
 - Chinese & Japanese

استقلال الجزائر في سنة 1962 بعد 132 عاماً من الاحتلال الفرنسي.
← → ← → ← START
'Algeria achieved its independence in 1962 after 132 years of French occupation.'

STOP words

↳ exclude from the dic, common words

- They have little semantic content: *the, a, and, to, be*
- There are a lot of them: ~30% of postings for top 30 words
- But the trend is away from doing this:
 - Good compression techniques (lecture 5) means the space for including stopwords in a system is very small
 - Good query optimization techniques (lecture 7) mean you pay little at query time for including stop words.
- You need them for:
 - Phrase queries: "King of Denmark"
 - Various song titles, etc.: "Let it be", "To be or not to be"
 - "Relational" queries: "flights to London"

case folding

↳ reduce all letters to lower case

- exception: upper case in mid-sentence?
 - e.g., General Motors
 - Fed vs. fed
 - SAIL vs. sail
- Often best to lower case everything, since users will use lowercase regardless of 'correct' capitalization...

Normalization

↳ normalize words into same form

↳ remove periods USA → USA

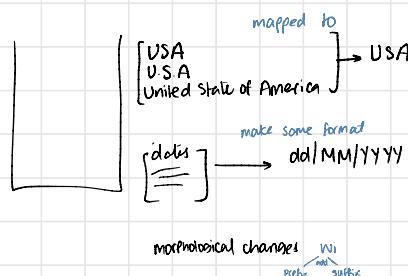
↳ remove hyphens

↳ results in a word present in IR dictionary

→ de-accent a word

Sec. 2.2.3
Normalization: other languages

- Accents: e.g., French **résumé** vs. **resume**.
- Umlauts: e.g., German: **Tübingen** vs. **Tüberlingen**
- Cedilla/diacritic
- Most important criterion:
 - How are your users like to write their queries for these words?
- Even in languages that standardly have accents, users often may not type them
 - Often best to normalize to a de-accented term
 - **Tuebingen, Tübingen, Tubingen \ Tübigen**



→ expand words

Sec. 2.2.3
Normalization to terms

- An alternative to equivalence classing is to do asymmetric expansion
 - **Search: window, windows**
 - **Search: Windows, windows, window**
 - **Search: Windows**
- An example of where this may be useful
 - Enter: **window** Search: **window, windows**
 - Enter: **windows** Search: **Windows, windows, window**
 - Enter: **Windows** Search: **Windows**
- Potentially more powerful, but less efficient

Thesauri and soundex

- Do we handle synonyms and homonyms?
 - E.g., by hand-constructed equivalence classes
 - car = automobile color = colour
 - We can rewrite to form equivalence-class terms
 - When the document contains automobile, index it under car-automobile (and vice-versa)
 - Or we can expand a query
 - When the query contains automobile, look under car as well
- What about spelling mistakes?
 - One approach is S^{under}oundex, which forms equivalence classes of words based on phonetic heuristics

Dictionay

- ↳ alphabetical order
- ↳ through details on
 - ↳ meaning
 - ↳ definition
 - ↳ usage
 - ↳ etymology → Origins
 - ↳ pronunciation
- ↳ used to learn word and its meanings

Thesauras

- ↳ conceptual order
- ↳ gives
 - ↳ synonyms
 - ↳ antonyms
 - ↳ relations with words
 - ↳ language usage
- ↳ used to know a diff word for a given word

20 sec

→ undo lexical and morphological changes

STEMMING

↳ a rule based process

↳ transforms diff form of words to an expected root word

↳ has various algos

↳ Porter's Stemmer

↳ Lovins Stemmer

↳ KROVETZ Stemmer

↳ REGEXP

CANDIDATE APPROACH

↳ to undo morphological changes

CONS

↳ sometimes stem is not human readable

↳ overstemming → sometimes chop too much

↳ low precision
→ more recall

PROS

↳ extremely fast

↳ higher recall

Porter Stemmer

- An incoming word is cleaned up in the initialization phase, one prefix trimming phase then takes place and then five suffix trimming phases occur.
- Note: The entire algorithm will not be covered
 - we will leave out some obscure rules.

Steps

1. Convert to lower case letters or digits are kept F-16 => f16

2. REMOVE prefixes kilo,micro,milli,mega,nano,pseudo so megabyte, kilobyte => byte

$\lambda = \{a, b, \dots, z\}$

word = {abort}

vowel = {a,e,i,o,u}

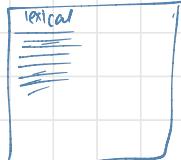
semi-vowel = {y,w}

constant = {the rest}

$[c|v]^* [vc]^* [c|v]^*$ → Vowel
m lenient words

if string is of 0 length
m=0 so need to make
an exception for it

Inverted Index



Tree
ccvv
m=0

rotation
cvcvcv
m=3

Porter Step 1

- Examples:
 - Remove "es" from words that end in "sses" or "ies"
 - passes --> pass, cries --> cri
 - Remove "s" from words whose next to last letter is not an "s"
 - runs --> run, fuss --> fuss
 - If word has a vowel and ends with "eed" remove the "ed"
 - agreed --> agre, freed --> freed
 - Replace trailing "y" with an "l" if word has a vowel
 - satisfy --> satisfi, fly --> fli

Porter Step 3

- With what is left, replace any suffix on the left with suffix on the right ...
 - icate ic fabricate --> fabric (*Think about this one!*)
 - ative -- combativ --> comb (*another good one!*)
 - alize al nationalize --> national
 - iciti ic tropical --> tropic
 - ical -- faithful --> faith
 - ful -- inventiveness --> inventive
 - ness -- harness --> har

Porter Step 2

- With what is left, replace any suffix on the left with suffix on the right ...

tional	tion	conditional --> condition
ization	ize	nationalization --> nationalize
iveness	ive	effectiveness --> effective
fulness	ful	usefulness --> useful
ousness	ous	nervousness --> nervous
ousli	ous	nervously --> nervous
entli	ent	fervently --> fervent
iveness	ive	inventiveness --> inventive
biliti	ble	sensibility --> sensible

Porter Step 4

- Remove remaining standard suffixes
 - al, ance, ence, er, ic, able, ible, ant, ement, ment, ent, sion, tion, ou, ism, ate, iti, ous, ive, ize, ise

Porter Step 5

- Remove trailing "e" if word does not end in a vowel
 - hinge --> hing
 - free --> free

Porter Stemmer: Experimental Results

Suffix stripping of a vocabulary of 10,000 words

Number of words reduced in step 1: 3597
 step 2: 766
 step 3: 327
 step 4: 2424
 step 5: 1373

Number of words not reduced: 3650

The resulting vocabulary of stems contained 6370 distinct entries. Thus the suffix stripping process reduced the size of the vocabulary by about one third.

Example

Sample text: Such an analysis can reveal features that are not easily 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 analysis can reveal features that are not easily 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

Lovins stemmer: such an analysis can reveal features that are not easily 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

Paice stemmer: such an analysis can reveal features that are not easily 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 Summary

- Do stemming and other normalizations help?
 - English: very mixed results. Helps recall but harms precision
 - operative (dentistry) --> oper
 - operational (research) --> oper
 - operating (systems) --> oper
 - Definitely useful for Spanish, German, Finnish, ...
 - 30% performance gains for Finnish!
- Full morphological analysis – at most modest benefits for retrieval

2 min

Lemmatization

- ↳ takes every word to dictionary to match term
- ↳ output is human readable

↳ doing proper reduction

↳ reduce variant forms to base forms

e.g. am, are, is → be

cars, car, car's, car's' → car

- the boy's cars are different colors → the boy
car be different color

↳ take tokens to dictionary

↳ which tells which word to replace token with
base form

- For example, in English, the verb 'to walk' may appear as 'walk', 'walked', 'walks', 'walking'. The base form, 'walk', that one might look up in a dictionary, is called the lemma for the word.

Word (lexeme) of a Language

- The formation process is based on

- Root or Origin
- Antonyms
- Synonyms
- Prefixes / Suffixes uses
- Part of speech
- Function
- Pronunciation
- Spelling
- Meaning

CONS

- ↳ very slow
- ↳ dictionary has coverage problem

PROS

- ↳ output is in human form
- ↳ better precision at the expense of lower recall

↙
more
stemming

Stemming

VS

Lemmatization

- ↳ operates on a single term without context
- ↳ faster
- ↳ higher recall
- ↳ low precision
- ↳ tokens not human readable

↳ uses dic and context to determine lemma

↳ slow

↳ lower recall

↳ higher precision

↳ token human readable

↳ dict size smaller → faster

↳ can fetch semantically similar words

MORPHOLOGY

↳ Study of the formations of word in a language

Inflectional

↳ Adding a suffix/prefix doesn't change its grammatical categories

↳ Modification of word

↳ Use of suffix common

↳ Semantically regular

e.g. adding -ing, -ed, -s

big, bigger, biggest

V>

Derivational

↳ Adding a suffix/prefix changes its grammatical categories

↳ Formation of new word

↳ Use of prefix/suffix common

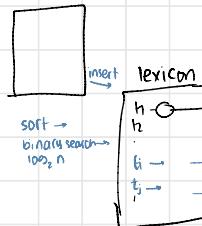
↳ Semantically irregular

e.g. Activation → reactivation

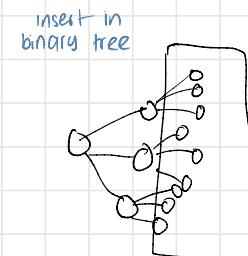
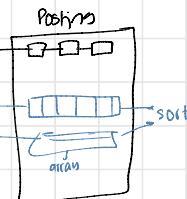
National → antinational

MORPHOLOGY

Inflectional



Q: $t_1 \wedge t_2$



Implementation Issues

- Inverted Index
 - Lists
 - Hashmap
 - Trees
- SkipList

Problem

- Are the following statements true or false?
 - In a Boolean retrieval system, stemming never lowers precision.
 - In a Boolean retrieval system, stemming never lowers recall.
 - Stemming increases the size of the vocabulary.
 - Stemming should be invoked at indexing time but not while processing a query.

Phrase Query

a sequence of words

e.g. "Stanford University"
↳ 10910M → contains 2 terms

1. Stanford, University
2. locate term in inverted index
3. Fetch Postings list
4. Intersection to get Stanford University adjacent to each other in a doc
5. Post Processing

Phrase queries

- Want to be able to answer queries such as "**stanford university**" – as a phrase
- Thus the sentence "*I went to university at Stanford*" is not a match.
- The concept of phrase queries has proven easily understood by users; one of the few "advanced search" ideas that works
- Many more queries are *implicit phrase queries*
- For this, it no longer suffices to store only <term : docs> entries

CON

↳ too many implicit phrase queries

↳ the bigger the data set result of intersection algo
the more the processing

↳ Post Processing

Has 2 Solutions

SOL 1

Bi-word indexes

↳ 2 word pairs indexed as a Phrase

↳ use 2 words as a feature

e.g. Stanford University
Ishma Mafeez

CON

↳ false positives

↳ index blow up → bigger dictionary by ~40% for English

PROS

Avoids Post Processing

Simple Phrase Queries: 2 words

General Phrase Queries: n words

h. What do we mean by a query post-processing in IR? What is its complexity?

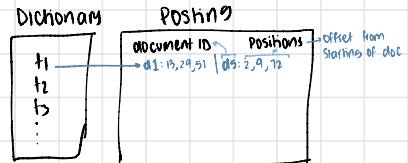
The post processing in IR is a specialized phase of result-set processing. In some model of IR there is a chance of getting false positive documents against a given query due to the processing approach. The post processing tries to filter all the false positive documents. The complexity of this phase is directly proportional to number of documents in the result-set.

Complexity \propto no of docs in resultset

Positional Index

↳ an inverted index

↳ Postings list has document ID and positions



2. locate term in inverted index

3. Fetch Postings list

4. find adjacent terms 15, 16, 17

CON

↳ Postings list storage increase substantially

↳ intersection algo changed

PRO

↳ no false positive

↳ quicker general Phrase query processing

Solution 1: Bi-word indexes

- Index every consecutive pair of terms in the text as a phrase
- 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.

Sec. 2.4.1

Solution 2: Positional indexes

- In the postings, store for each **term** the position(s) in which tokens of it appear:

<**term**, number of docs containing **term**;
doc1: position1, position2 ... ;
doc2: position1, position2 ... ;
etc.>

Sec. 2.4.2

Positional index example

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

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
- But we now need to deal with more than just equality

Sec. 2.4.2

Processing a phrase query

- Extract inverted index entries for each distinct term: **to**, **be**, **or**, **not**.
- Merge their **doc:position** lists to enumerate all positions with "**to be or not to be**".
- **to**:
 - 2:1,17,74,222,551; 4:8,16,190,429,433;
 7:13,23,191; ...
- **be**:
 - 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...
- Same general method for proximity searches

Positional Intersect

```

POSITIONALINTERSECT( $p_1, p_2, k$ )
1  answer  $\leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3    do if docID( $p_1$ ) = docID( $p_2$ )
4      then  $I \leftarrow \langle \rangle$ 
5       $p_{p1} \leftarrow \text{positions}(p_1)$ 
6       $p_{p2} \leftarrow \text{positions}(p_2)$ 
7      while  $p_{p1} \neq \text{NIL}$ 
8        do while  $p_{p2} \neq \text{NIL}$ 
9          do if  $|\text{pos}(p_{p1}) - \text{pos}(p_{p2})| \leq k$ 
10            then ADD( $I, \text{pos}(p_{p2})$ )
11          else if  $|\text{pos}(p_{p1}) - \text{pos}(p_{p2})| > pos(p_{p1})$ 
12            then break
13           $p_{p2} \leftarrow \text{next}(p_{p2})$ 
14        while  $p_{p2} \neq \langle \rangle$  and  $|I[0] - \text{pos}(p_{p1})| > k$ 
15        do DELETE( $I[0]$ )
16        for each  $ps \in I$ 
17        do ADD(answer, (docID( $p_1$ ), pos( $p_{p1}$ ),  $ps$ ))
18         $p_{p1} \leftarrow \text{next}(p_{p1})$ 
19         $p_{p2} \leftarrow \text{next}(p_{p2})$ 
20      else if docID( $p_1$ ) < docID( $p_2$ )
21        then  $p_1 \leftarrow \text{next}(p_1)$ 
22        else  $p_2 \leftarrow \text{next}(p_2)$ 
23
24  return answer

```

Longer phrase Queries → General phrases

↳ no limit on number of terms

t_1, t_2, t_3, t_4, t_5

↓
break into biword phrase queries

$(t_1 t_2)$ $(t_2 t_3)$ $(t_3 t_4)$ $(t_4 t_5)$

Query Expansion

CON

↳ can have false positives so need to do post processing

Sec. 2.4.1 Longer phrase queries

■ Longer phrases are processed as we did with wild-cards:

■ **stanford university palo alto** can be broken into the Boolean query on biwords:

stanford university AND university palo AND palo alto

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.

Can have false positives!

Extended bi-word index

↳ performs parts of speech tagging (POST)

↳ bucket terms into NOUNS and PREPOSITIONS

↳ $N \times N$ strings are extended biword

↳ it is now a term in dictionary

e.g. $\underline{\text{coin}} \ \underline{\text{in}} \ \underline{\text{the}} \ \underline{\text{pocket}}$

$t_1 \quad t_2 \rightarrow \text{feature}$

→ ① coin are there in pocket

■ Example: **coins are in pocket**

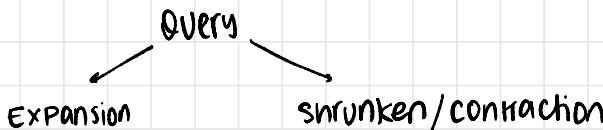
N X X N

■ Query processing: parse it into N's and X's

□ Segment query into enhanced biwords

□ Look up in index: **coin pocket**

* QODDIE only includes 10 terms in the query



↳ add more words synonyms, ontologies
e.g. astronaut + cosmo

CON
might bring what user didn't ask for

CON/might not show anything

Proximity Queries

1. $W_1 W_2 / 2$
2. $W_1 W_2 / 5$

- ↳ uses Positional index
- ↳ can't use biword indexes

Sec. 2.4.2 Proximity queries

- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
 - Again, here, $/k$ means "within k words of".
- Clearly, positional indexes can be used for such queries; biword indexes cannot.
- Exercise: Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of k ?
 - This is a little tricky to do correctly and efficiently
 - See Figure 2.12 of IR
 - There's likely to be a problem on it!

Positional Index size

- ↳ Positional index expands postings storage substantially

↳ it is standardly used because of
of the power and usefulness of
phrase and proximity queries

Sec. 2.4.2

Positional index size

- Need an entry for each occurrence, not just once per document
- Index size depends on average document size
 - Average web page has <1000 terms Why?
 - SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%

Document size	Postings	Positional postings
1000	1	1
100,000	1	100

Sec. 2.4.2

Rules of thumb

- A positional index is 2–4 as large as a non-positional index
- Positional index size 35–50% of volume of original text
- Caveat: all of this holds for "English-like" languages

if query term frequent → bi-word

if query term not frequent → Positional index

→ to scale performance

Sec. 2.4.3

Combination schemes

- These two approaches can be profitably combined
 - For particular phrases ("Michael Jackson", "Britney Spears") it is inefficient to keep merging positional postings lists
 - Even more so for phrases like "The Who"
- Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme
 - A typical web query mixture was executed in 1/4 of the time of using just a positional index
 - It required 26% more space

Postman is just like a linkedlist

Skiplist

→ Probabilistic data structure

↳ specialized link list

↳ improved searching

↳ is of two types

↳ homogenous [Node]

↳ heterogeneous

no compromise on storage

CON

↳ bad for $t_1 \text{ OR } t_2$

PRO

↳ good for $t_1 \text{ AND } t_2$

Singly Linked List

1. unordered SLL

insertion: $O(1)$

selection: $O(n)$

2. ordered SLL

insertion: $O(n)$

selection: $O(g_n)$

Head



↓



CONS

↳ searching takes time → as start from top of the list
so if list large it takes too much time

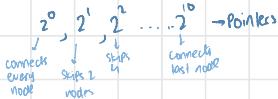
insertion: $O(n)$ searching: $10^{g_2} n$

HOMOGENIOUS NODE

↳ sorted order

↳ all nodes identical

↳ searching faster by skipping pointers



[1000]

$2^n \leq 1000$

$n=10$

1000 elements

1 integer: 4 bytes

$1000 \times 4 =$

CONS

↳ wasteful memory → every node identical and every node needs 10 pointers

SOLUTION

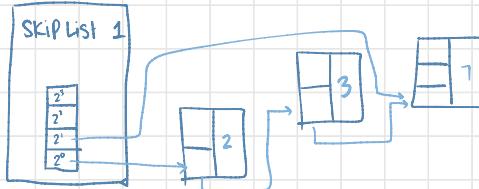
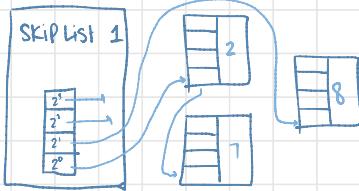
Probabilistic Node Structure

↳ give every node atleast 1 pointer

↳ 50% nodes 2 pointers

↳ 25% nodes 3 pointers

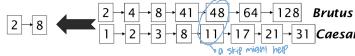
↳ 12.5% nodes 4 pointers



Sec. 2.3.

Recall basic merge

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

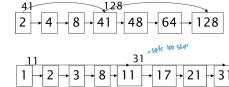


If the list lengths are m and n , the merge takes $O(m+n)$ operations.

Can we do better?
Yes (if the index isn't changing too fast).

Sec. 2.3.

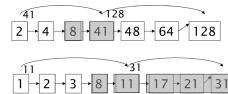
Augment postings with skip pointers (at indexing time)



- Why?
- To skip postings that will not figure in the search results.
- How?
- Where do we place skip pointers?

Sec. 2.3.

Query processing with skip pointers



Suppose we've stepped through the lists until we process 8 on each list. We match it and advance.

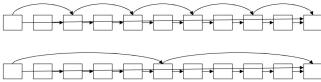
We then have 41 and 11 on the lower. 11 is smaller. But the skip successor of 11 on the lower list is 31, so we can skip ahead past the intervening postings.

Sec. 2.3.

Where do we place skips?

Tradeoff:

- More skips \rightarrow shorter skip spans \Rightarrow more likely to skip. But lots of comparisons to skip pointers.
- Fewer skips \rightarrow few pointer comparison, but then long skip spans \Rightarrow few successful skips.



MORE SKIPS \rightarrow shorter skips

\hookrightarrow too many pointer comparisons

Fewer skips \rightarrow longer skips

\hookrightarrow less pointer comparisons

\hookrightarrow few successful skips

Faster Skip Lists

```
INTERSECTWITHSKIP(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 while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
9       do p1 ← skip(p1)
10    else p1 ← next(p1)
11  else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
12    then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
13      do p2 ← skip(p2)
14    else p2 ← next(p2)
15 return answer
```

Figure 2.10 Postings lists intersection with skip pointers.

Sec. 2.3.

Placing skips

- Simple heuristic: for postings of length L , use \sqrt{L} evenly-spaced skip pointers [Moffat and Zobel 1996].
- This ignores the distribution of query terms.
- Easy if the index is relatively static; harder if L keeps changing because of updates.
- This definitely used to help; with modern hardware it may not unless you're memory-based [Bahle et al. 2002]
- The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging!

inverted index $\begin{cases} \rightarrow \text{Boolean queries} \\ \rightarrow \text{Proximity queries} \end{cases}$

Positional indexes \rightarrow general Phrase queries
 \rightarrow Proximity queries

modify intersection of postings list to speed up in generating result set

\hookrightarrow Sort by increasing freq

Data Structures for Dictionary

Hashable

- ↳ each vocab term is hashed into an integer

PROS

- ↳ look up faster than a tree $O(1)$

CONS

- ↳ no easy way to find minor variants
e.g. judgment / judgement
- ↳ no prefix search
- ↳ rebalancing needed if vocab keeps growing

Tree

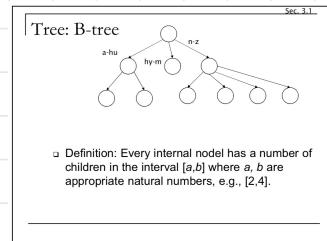
- ↳

PROS

- ↳ solves Prefix problem

CONS

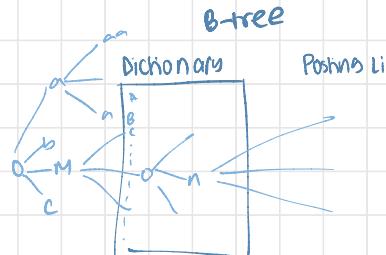
- ↳ slower
- ↳ Rebalancing binary tree is expensive



B-tree: internal nodes have data

B+tree: leafs have data

↳ used for memory



Wild card Queries (*)

SITUATIONS

- ↳ user is uncertain of spelling e.g. Sidney instead of Sydney
↳ wild card query: Sydney*
- ↳ user aware of multiple variants e.g. colour vs color
- ↳ user wants variants of the term e.g. judica*
- ↳ user uncertain of correct rededentation of foreign word/phrase

Trailing Wildcard Queries

mon*: words starting with mon

↳ binary tree

↳ B-tree

$$mon \leq w < mon$$

Leading Wildcard Queries

*mon: words ending with mon

↳ maintain another reverse B-tree

$$mon \leq w < non$$

General Wildcard Queries

co*tim , re*xta*hon

↳ look up co*t and *tim in a B-tree
↳ then intersect the two terms

CONS

↳ too many false positives → ?

↳ too expensive



SOLUTION

↳ make subcases

↳ no of queries and cost increases

↳ false positives reduced



SOLUTION

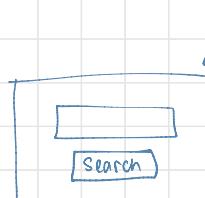
1. Permuterm Index

2. Bigram index

WHAT?

General Wild Card Query

- We now study two techniques for handling general wildcard queries.
 - Both techniques share a common strategy: express the given wildcard query q_w as a Boolean query Q on a specially constructed index, such that the answer to Q is a superset of the set of vocabulary terms matching q_w .
 - Then, we check each term in the answer to Q against q_w , discarding those vocabulary terms that do not match q_w . At this point we have the vocabulary terms matching q_w and can resort to the standard inverted index.



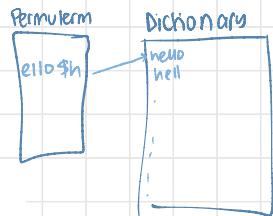
Advance Search
↳ hidden by google
↳ cost heavy

Permuterm index

- ↳ transform wild card queries so that the *'s occurs at the end

Hello
Hello \$
e llo g h
llo g he
llo g hel

he * o



Permuterm Query Processing

- ↳ rotate wild card query to the right

- ↳ use B-tree for look up

CON

increase dictionary size

5 character word → size quadrupled

PROS

- ↳ reduces false positives

Bigram (K-gram) index (\$)

- ↳ enumerate all k-grams in any word **moon** ↳ in sorted order
- ↳ maintain a 2nd inverted index from bigrams to dictionary terms that match each bigram
- ↳ \$ is a word boundary

→ bigram → k=2

K-gram



- e.g., from text "April is the cruellest month" we get the 2-grams (bigrams)

\$a,ap,pr,i,il,S,l,i,s,S,\$t,he,e\$,Sc,cr,ru,
ue,e,le,es,st,t\$, Sm,mo,on,n,h\$

- ↳ \$ is a special word boundary symbol

↓
sort it

Sec. 3.2.2

Processing wild-cards

- Query **mon*** can now be run as \$m AND mo AND on
- Gets terms that match AND version of our wildcard query.
- But we'd enumerate **moon**.
- Must post-filter these terms against query.
- Surviving enumerated terms are then looked up in the term-document inverted index.
- Fast, space efficient (compared to permuterm).
- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution (very large disjunctions...)
 - ↳ python AND prog*
- If you encourage "laziness" people will respond!

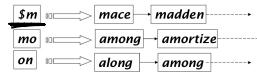
Sec. 3.2.2

Type your search terms, use '*' if you need to.
E.g., Alex* will match Alexander.

Search

Bigram index example

- The k-gram index finds terms based on a query consisting of k-grams (here k=2).



↓
sort it

Sec. 3.2.2

PROS

- ↳ fast

- ↳ space efficient

CON

- ↳ expensive query execution

Spelling Corrections

↳ TWO USES

1. correcting documents being indexed → identifying misspellings
2. Correcting user queries → to retrieve right answers → fetch correct word

↳ Two main flavours

1. Isolated word

- ↳ checks each words misspelling individually e.g. fail → fail
- ↳ won't correct typos spelled right e.g. from → form

2. Context Sensitive

- ↳ looks at surrounding words e.g. karachi to lahore

I flew from

Document Correction

- Especially needed for OCR'ed documents
 - Correction algorithms are tuned for this: "m" / "m"
 - Can use domain-specific knowledge
 - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).
- But also: web pages and even printed material has typos
- Goal: the dictionary contains fewer misspellings

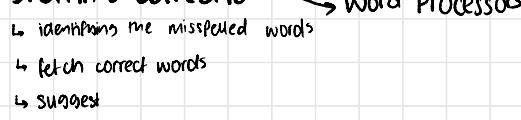
Spelling Corrections

Return to Google's Job page

48501 history spans	29 history spans	1 history spans	3 history spans	2 history spans
61218 history spans	29 history spans	1 history spans	3 history spans	2 history spans
48501 history spans	29 history spans	1 history spans	3 history spans	2 history spans
48501 history spans	29 history spans	1 history spans	3 history spans	2 history spans
48501 history spans	29 history spans	1 history spans	3 history spans	2 history spans

Isolated Word Correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this
 - A standard lexicon such as
 - Webster's English Dictionary
 - An "industry-specific" lexicon – hand-maintained → make our own dictionary
 - The lexicon of the indexed corpus
 - E.g., all words on the web
 - All names, acronyms etc.
 - (Including the mis-spellings)
- 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



- ↳ check if in dictionary
- ↳ if incorrect word
check closest word in dictionary

Find closest word

1. Edit distance → cost of diff ops are same
2. weighted edit distance → cost of diff operations is diff
3. n-gram overlap

Heuristics suggest

- ↳ assume first char correct
- ↳ assume word length is ± 1
- ↳ check search log, and which has most hits
- ↳

Edit distance

tells you how to convert 1 string into another

↳ min no of ops to convert s_1 to s_2

soft issue



Operations are

↳ Insert

- E.g., the edit distance from **dog** to **dog** is 1
- From **cat** to **act** is 2 (Just 1 with transpose.)
- From **cat** to **dog** is 3.

↳ Replace

↳ Delete

A: "horse"

B: "ros"

1 Replace = "orse"
1 Delete = "rose"
1 Delete = "ros"
3 operations

Edit Distance – Levenshtein

```
EDITDISTANCE( $s_1, s_2$ )
1 int m[i, j] = 0
2 for i ← 1 to | $s_1$ |
3 do m[i, 0] = i
4 for j ← 1 to | $s_2$ |
5 do m[0, j] = j
6 for i ← 1 to | $s_1$ |
7 do for j ← 1 to | $s_2$ |
8   do m[i, j] = min(m[i - 1, j - 1] + if ( $s_1[i] = s_2[j]$ ) then 0 else 1 if,
9                      m[i - 1, j] + 1,
10                     m[i, j - 1] + 1) cost for
11    insert, replace, delete
```

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

	f	a	s	t
	0	1 1	2 2	3 3 4 4
c	1	1 2	2 3	3 4 4 5
	2	2 1	2 2	3 3 4 4
a	1	2 2	1 3	3 4 4 5
	2	3 2	3 1	2 3 3 3
s	1	3 3	3 2	2 3 2 4
	2	4 3	4 2	3 2 3 2
t	3	4 4	4 3	3 2 3 2
	4	5 4	5 3	4 2 3 3
s	4	5 5 4	5 3	4 2 3 3

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

		f	a	s	t
		0	1	2	3
		0	0	1	2
		0	1	2	3 4 5 6
r	1	1	2	3	4 5
e	2	2	2	3	4 5
p	3	3	3	3	4 5
h	4	4	4	4	4 5
r	5	5	5	5	5 6
e	6	5	5	6	6 5
m	7	5	6	6	6 5

edit distance b/w "f" and "fr" → as it is

edit distance b/w "s" and "n" → as it is

④ → the edit distance b/w these two strings

Using Edit Distance

↳ enumerate all char sequence within a present edit distance eg 2

↳ check how many of these are present in lexicon dataset

↳ then choose which has most hits in corpus

↳ show found terms to user as suggestions

■ Alternatively,

- We can look up all possible corrections in our inverted index and return all docs ... slow
- We can run with a single most likely correction

A						
" " b e n y a m						
	0	1	2	3	4	5
"	0	1	2	3	4	5
"	1	1	2	3	4	5
b	2	2	2	3	4	5
e	3	3	3	3	4	5
p	4	4	4	4	4	5
h	5	5	5	5	5	5
r	6	5	6	6	6	5
e	7	5	6	6	6	5
m	8	5	6	6	6	5

write as is in starting

write as is in string

Operations

R	I
D	min of 3 cells ↓ if doesn't match

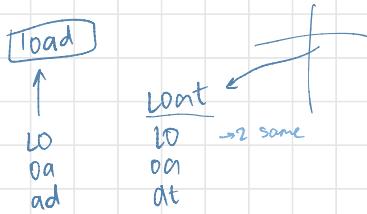
min of 3 cells
↓
if matches

→ no of operations = 5

N-gram Overlaps

- Enumerate all the n-gram in a query
- Use n-gram index to retrieve all lexicon terms matching any of the query n-grams
- Threshold by no of matching n-grams

Variants: weight by keyboard layout, etc



n-gram overlap: Jackson

$$S_i \cap S_j : \frac{2}{4}$$

Context Sensitive Spelling

- lets say words are entered and they are correct and in dic
- BUT when entered together no doc

- 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.

w1 w2 w3

↳ correct but not found any phrase

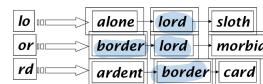
1. incorrect

$2 \times 1 \times 3$

2. wrong order

2-grams for match

- Consider the query **lord** – we wish to identify words matching 2 of its 3 bigrams (**lo, or, rd**)



Standard postings "merge" will enumerate ...

Answer list based on threshold
e.g. threshold overlap 2

we define

?

→ use weight

Context Sensitive Spelling Corrections

- Retrive all dic terms close 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.

Issues in Spelling Corrections

- We enumerate multiple alternatives for "Did you mean?"
- Need to figure out which to present to the user
- Use heuristics
 - The alternative hitting most docs
 - Query log analysis + tweaking
 - For especially popular, topical queries
- Spell-correction is computationally expensive
 - Avoid running routinely on every query?
 - Run only on queries that matched few docs

Soundex

↳ groups together terms that sound similar when pronounced into the same equivalence class

↳ mostly used for names

Soundex

→ assume sound same

- Class of heuristics to expand a query into phonetic equivalents
 - Language specific – mainly for names
 - E.g., *chebyshev* → *tchebycheff*
- Invented for the U.S. census ... in 1918

Soundex Algorithm

1. Retain the first letter of the word. **FLOOR**
2. Change all occurrences of the following letters to '0' (zero):
'A', 'E', 'I', 'O', 'U', 'H', 'W', 'Y'. FLOOR
3. Change letters to digits as follows:
 - B, F, P, V → 1
 - C, G, J, K, Q, S, X, Z → 2
 - D, T → 3
 - L → 4
 - M, N → 5
 - R → 6**F4006**
4. Remove all pairs of consecutive digits. **F406**
5. Remove all zeros from the resulting string. **F46**
6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>. **F 4 0 6**

E.g., *Herman* becomes H655.

Soundex

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, ...)
- How useful is soundex?
 - Not very – for information retrieval
- Zobel and Dart (1996) show that other algorithms for phonetic matching perform much better in the context of IR

CONS

↳ lowers precision

as many things will be mapped to the same posting list

→ mapped to
the same posting list

Soundex Exercise

- Find two differently spelled proper nouns (different to the course example) whose soundex codes are the same and give their soundex code.
 - Mary, Nira (Soundex code = 5600).
- Find two phonetically similar proper nouns whose soundex codes are different.
 - Chebyshev, Tchebycheff
 - Rafi, Rafee

CRS + ALEXANDER → 2 days

Hardware Basics

- ↳ Access to data in memory is much faster than access to data on disk

Disk Seek: 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

- Servers used in IR systems now typically have several GB of main memory, sometimes tens of GB.
- Available disk space is several (2–3) orders of magnitude larger.
- Fault tolerance is very expensive; it's much cheaper to use many regular machines rather than one fault tolerant machine.

Basic Inverted Index

- The basic steps in constructing a non-positional index as discussed in chapter 1.
- We first make a pass through the collection assembling all term-docID pairs.
- We then sort the pairs with the term as the dominant key and docID as the secondary key.
- 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.
- For large collections (most now in IR)

Hardware Basics (2007)

► Table 4.1 Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

Symbol	Statistic	Value
s	average seek time	$5 \text{ ms} = 5 \times 10^{-3} \text{ s}$
b	transfer time per byte	$0.02 \mu\text{s} = 2 \times 10^{-8} \text{ s}$
	processor's clock rate	10^9 s^{-1}
p	lowlevel operation (e.g., compare & swap a word)	$0.01 \mu\text{s} = 10^{-8} \text{ s}$
	size of main memory	several GB
	size of disk space	1 TB or more

Seek time: time needed to position the disk head in a new position

Transfer time: rate of transfer from disk to memory when head is in the right position

intended. It is a perfect example where isolation correction failed.

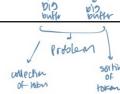
- j. Why do we want to compress the dictionary in the context of IR?

One of the primary factors in determining the response time of an IR system is the number of disk seeks necessary to process a query. If parts of the dictionary are on disk, then many more disk seeks are necessary in query evaluation. Thus, the main goal of compressing the dictionary is to fit it in main memory, or at least a large portion of it, to support high query throughput.

Example

Doc 1 I did enact Julius Caesar. I was killed by the Capitol; Brutus killed me.
Doc 2 So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.

term	docID	term	docID	term	doc freq.	postings lists
did	1	ambitious	1	ambitious	1	1
enact	1	brutus	1	brutus	2	1, 2
passus	1	brutus	2	brutus	2	1, 2
caesar	1	capitol	1	capitol	1	1
i	1	caesar	2	caesar	2	1, 2
was	1	caesar	2	caesar	2	1, 2
killed	1	caesar	2	caesar	2	1, 2
the	1	did	1	did	1	1
capitol	1	did	1	did	1	1
brutus	1	both	1	both	1	1
killed	1	i	1	i	1	1
me	1	it	1	it	1	1
so	1	palius	1	palius	1	1
let	1	killed	1	killed	1	1
the	1	it	1	it	1	1
with	1	killed	2	it	1	1
caesar	1	not	1	not	1	1
this	1	no	1	no	1	1
those	1	so	1	so	1	1
brutus	1	told	1	told	1	1
hath	1	the	1	the	1	1
told	1	you	1	you	1	1
you	1	were	1	were	1	1
caesar	1	was	1	was	1	1
was	2	with	1	with	1	1

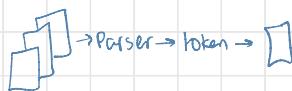


Token N = 10 million
· 4 words

Reuters- RCV1 Collection

► **Table 4.2** Collection statistics for Reuters-RCV1. Values are rounded for the computations in this book. The unrounded values are 806,791 documents, 222 tokens per document, 391,523 (distinct) terms, 6.04 bytes per token with spaces and punctuation, 4.5 bytes per token without spaces and punctuation, 7.5 bytes per term, and 96,969,056 tokens. The numbers in this table correspond to the third line ("case folding") in Table 5.1 (page 87).

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



soft all terms

challenge: keep huge data in 1 buffer
and soft that buffer

Term ID Term Doc ID

4 bytes 4 bytes

1 term = 12 bytes

RCV1 has 100,000 terms so 48 million bytes

To solve challenge

1. BSB1
2. SPIMI

→ sorting with fewer disk seeks

Blocked Sort Based Indexing (BSBI)

↳ Take buffer into mem

↳ Process and sort terms in that buffer

↳ then write in disk, disk will have blocks

↳ then merge blocks into 1 and so on...

▫ Accumulate postings for each block, sort, write to disk.

▫ Then merge the blocks into one long sorted order.

Algorithm (BSBI)

- Segments the collection into parts of equal size,
- Sorts the termID-docID pairs of each part in memory,
- Stores intermediate sorted results on disk, and
- Merges all intermediate results into the final index.

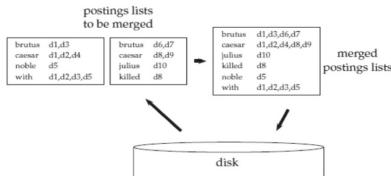
Blocked Sort-Based Indexing

BSBICONSTRUCTION()

- 1 $n \leftarrow 0$
- 2 while (all documents have not been processed)
- 3 do $n \leftarrow n + 1$
- 4 block \leftarrow PARSENEXTBLOCK()
- 5 BSB1-INVERT(block)
- 6 WRITEBLOCKTODISK(block, f_n)
- 7 MERGEBLOCKS($f_1, \dots, f_n, f_{merged}$)

► **Figure 4.2** Blocked sort-based indexing. The algorithm stores inverted blocks in files f_1, \dots, f_n and the merged index in f_{merged} .

Blocked Sort-Based Indexing



► **Figure 4.3** Merging in blocked sort-based indexing. Two blocks ("postings lists to be merged") are loaded from disk into memory, merged in memory ("merged postings lists") and written back to disk. We show terms instead of termIDs for better readability.

Blocked Sort-Based Indexing

■ BSB1 Complexity

- How expensive is BSB1?
- Its time complexity is $O(T \log T)$ because the step with the highest time complexity is sorting and T is an upper bound for the number of items we must sort (i.e., the number of termID-docID pairs).
- The actual indexing time is usually dominated by the time it takes to parse the documents (PARSENEXTBLOCK) and to do the final merge (MERGEBLOCKS).

PROS

↳ doesn't sort big collection but smaller chunks into big collection

TIME Complexity

↳ $O(T \log T)$

↑ upper bound of
no. of items to sort ↑ no. of
termID-docID pairs

2nd Solution

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 indexes can then be merged into one big index.

every block will be an inverted index

PROS

- ↳ No need to sort
- ↳ Will be complete inverted index

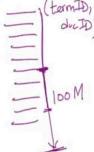
BSBI

- ↳ first collects all termID-docID pairs and then sorts them

termID - DocID

↳ fewer disk seeks

↳ records (term, doc ID)
iterated as we page docs.
↳ 100M such 2-byte records by termID
↳ 10M such records
↳ a couple into memory.
↳ such blocks to start with.
↳ algorithm:
posting for each block, sort, write to disk.
the blocks into one long sorted order.



Block-Sort Based Indexing (BSBI)

- BSBI uses continuous disk space to collect all terms from document collections by dividing collection into equal parts, iteratively.
- It uses a data structures to collect termID and docID into memory.
- The running time is proportional to $T \log T$ where T is Number of terms in the collection. Dominated by sorting of terms in a collection.

Single-Pass In Memory Indexing (SPIMI)

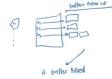
- SPIMI add posting directly to posting list and small posting list are stored into the continuous disk blocks.
- There is no need to map termID and docID pairs and hence no sorting is required. Faster and efficient.
- The running time is linear in term of $T(\text{number of terms in the collection})$.
- SPIMI also support compression of posting lists.

placement new()



Single-Pass In-Memory Indexing

```
SPMI-INVERT(token_stream)
1 output_file = NEWFILE()
2 dictionary = NEWHASH()
3 while (free memory available)
4   do token ← next(token_stream) ← link list
5   if term(token) ∉ dictionary
6     then postings_list = ADDTODICTIONARY(dictionary, term(token))
7     else postings_list = GETPOSTINGSLIST(dictionary, term(token))
8   if full(postings_list)
9     then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10    ADDTOPINGLISHT(postings_list, docID(token))
11    sorted_terms = SORTTERMS(dictionary)
12    WRITERBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```



► Figure 4.4 Inversion of a block in single-pass in-memory indexing

VS

SMIMI

- ↳ adds a postings directly to its postings list

↳ NO sorting

↳ faster → AS no sorting

↳ saves memory

Each Postings list is dynamic

- Each postings list is dynamic (i.e., its size is adjusted as it grows) and it is immediately available to collect postings.

- This has two advantages:

- It is faster because there is no sorting required
- it saves memory because we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored. As a result, the blocks that individual calls of SPIMI-INVERT can process are much larger and the index construction process as a whole is more efficient.

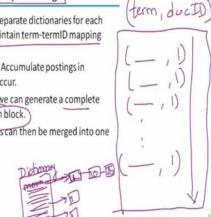
in-memory indexing

Generate separate dictionaries for each need to maintain term-termID mapping

BSBI
Don't sort. Accumulate postings in as they occur

two ideas we can generate a complete lex for each block

late indexes can then be merged into one



Distributed Indexing

- ↳ maintain a master machine directing the indexing job
- ↳ break up indexing into sets of parallel tasks
- ↳ master machine assigns each task to an idle machine from a pool

...
...
...

- Web search data centers (Google, Bing, Baidu) mainly contain commodity machines.

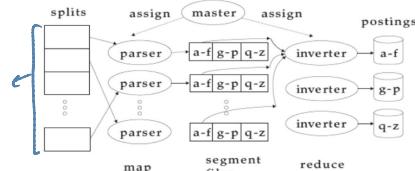
Distributed Indexing

PRO
• no disk require

- Distributed Indexing
 - Parallel Tasks
 - Parser
 - Master assigns a split to an idle parser machine
 - Parser reads a document at a time and emits (term, doc) pairs *generates a stream of normalized (term,doc) pairs*
 - Parser writes pairs into / partitions
 - Inverters
 - An inverter collects all (term, doc) pairs (= postings) for one term-partition.
 - Sorts and writes to postings lists

Distributed Indexing

indexing
split into
sets of
parallel tasks



► Figure 4.5 An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).

Map-Reduce Based Indexing

Schema of map and reduce functions

map: input
reduce: $\langle \emptyset, \text{list}(\cdot) \rangle$

$\rightarrow \text{list}(k, v)$
 $\rightarrow \text{output}$

Instantiation of the schema for index construction

map: web collection
reduce: $\langle \text{termID}_1, \text{list}(\text{docID}) \rangle, \langle \text{termID}_2, \text{list}(\text{docID}) \rangle, \dots \rightarrow \text{postings}, \text{list}, \text{postings_list}_1, \dots \rangle$

$\rightarrow \text{list}(\text{termID}, \text{docID})$

Example for index construction

map: $d_1 : \text{C died}, d_1 : \text{C came}, \text{C cod}$
reduce: $\langle (\text{C}, d_2), (\text{died}, d_1), (\text{C}, d_1), (\text{came}, d_1), (\text{cod}, d_1) \rangle \rightarrow \langle (\text{C}, d_1, d_2, \dots), (\text{died}, d_1), (\text{came}, d_1), (\text{cod}, d_1) \rangle \rightarrow \langle (\text{C}, d_1, d_2, \dots), (\text{died}, d_1), (\text{came}, d_1), (\text{cod}, d_1) \rangle$

$\rightarrow \langle (\text{C}, d_1, d_2, \dots), (\text{died}, d_1), (\text{came}, d_1), (\text{cod}, d_1) \rangle \rightarrow \langle (\text{C}, d_1, d_2, \dots), (\text{died}, d_1), (\text{came}, d_1), (\text{cod}, d_1) \rangle$

► Figure 4.6 Map and reduce functions in MapReduce. In general, the map function produces a list of key-value pairs. All values for a key are collected into one list in the reduce phase. This list is then processed further. The instantiations of the two functions and an example are shown for index construction. Because the map phase processes documents in a distributed fashion, termID-docID pairs need not be ordered correctly initially as in this example. The example shows terms instead of termIDs for better readability. We abbreviate Caesar as C and conquered as cod.

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

if corpus is continuously changing
docs are constantly being added, modified, deleted
hence dictionary and posting list needs to be updated

Deletions

indicates which docs have been deleted

- ↳ maintain a bit vector for deleted docs
- ↳ when the doc is deleted
- ↳ we mark the bit vector
- ↳ then filter out the invalid bit vectors from search result
- ↳ size of bit vector: no of docs in corpus

Modifications

↳ treat as deletions then insertions

DYNAMIC INDEX

In insertions

- ↳ maintains big main index

- ↳ new docs go into small auxiliary index

- ↳ when query comes we send it to both small and large index

- ↳ take results from both

- ↳ then combine them

newly appearing index
when becomes
too large merge
with main index

Google Dance

- Approach is same as in dynamic indexing
- Search engines do incremental (auxiliary posting list) plus some time full indexing in background.

Conclusion

- Indexing text is a challenging task, it requires to consider hardware, OS and Disk IO
- Original publication on MapReduce: Dean and Ghemawat (2004)
- Original publication on SPIMI: Heinz and Zobel (2003)

DISTRIBUTED COMPUTING

Cluster vs. Grid vs. Cloud

	Cluster Computing	Grid Computing	Cloud Computing
Basic Idea	Aggregation of resources.	Segregation of Resources.	Consolidation of Resources.
Running Processes	Same processes run on all computers over the cluster at the same time.	Job is divided into sub-jobs each is assigned to an idle CPU so they all run concurrently.	Depends on service provisioning. Which computer offers a service and provisions it to the requesting clients.
Operating System	All nodes must run the same operating system.	No restriction is made on the operating system.	No restriction is made on the operating system.
Job Execution	Execution depends on job scheduling. So, jobs wait until it's assigned a runtime.	Execution is scalable in a way that moves the execution of a job to an idle processor (node).	Self-Managed.
Suitable for Apps	Cascading tasks. If one task depends on another one.	Not suitable for cascading tasks.	On-demand service provisioning.
Location of nodes	Physically in the same location	Distributed geographically all over the globe.	Location doesn't matter
Homo/Heterogeneity	Homogenous	Heterogeneous	Heterogeneous

Cluster vs. Grid vs. Cloud

	Cluster Computing	Grid Computing	Cloud Computing
Virtualization	None	None	Virtualization is a key
Transparency	Yes	Yes	Yes
Security	High	High, but doesn't reach the level of cluster computing.	Lower than both types.
Interoperability	Yes	Yes	No
Application Domains	industrial sector, research centers, health care, and centers that offer services on the nation-wide level	industrial sector, research centers, health care, and centers that offer services on the nation-wide level	Banking, Insurance, Weather Forecasting, Space Exploration, Business, IaaS, PaaS, SaaS
Implementation	Easy	Difficult	Difficult – need to be done by the host.
Management	Easy	Difficult	Difficult
Resource Management	Centralized (locally)	Distributed	Both centralized and distributed.
Internet	No internet access is required	Required	Required

Why Compression

↳ saves money → uses less disk space

↳ keep more stuff in memory → increases speed

↳ increase speed of data transfer from disk to memory

- In most cases, retrieval system runs faster on compressed postings lists than on uncompressed postings lists.

Why Compression

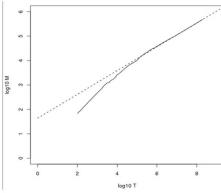
- First, we will consider space for dictionary
 - Make it small enough to keep in main memory
- Then the postings
 - Reduce disk space needed, decrease time to read from disk
 - Large search engines keep a significant part of postings in memory
- (Each postings entry is a docID)

Heap's Law

- HEAPS' LAW helps in estimates vocabulary size as a function of collection size

$$M = kT^b$$

- Vocabulary size M as a function of collection size T
- For RCV1, the dashed line $\log_{10}M = 0.49 \log_{10}T + 1.64$ is the best least squares fit.
Thus, $M = 10^{1.64} T^{0.49}$
so $k = 10^{1.64} \approx 44$
and $b = 0.49$.



Heap's Law

- The parameter k is quite variable because vocabulary growth depends a lot on the nature of the collection and how it is processed.
- Case-folding and stemming reduce the growth rate of the vocabulary, whereas including numbers and spelling errors increase it.
- Regardless of the values of the parameters
 - for a particular collection, Heaps' law suggests that
 - The dictionary size continues to increase with more documents in the collection, rather than a maximum vocabulary size being reached.
 - The size of the dictionary is quite large for large collections.

→ larger vocab has
large dictionary
+ dictionary is continuous

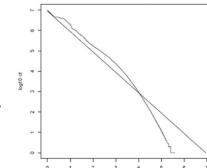
Zipf's law

most freq word → based on freq

- Heaps' Law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In a natural language, there are very few very frequent terms and very many very rare terms.
- Zipf's law: The i th most frequent term has frequency proportional to $1/i$.
- $cf_i \propto 1/i$ where a is a normalizing constant
- cf_i is collection frequency: the number of occurrences of the term i in the collection.

Zipf consequences

- If the most frequent term (*the*) occurs cf_1 times, then
 - the second most frequent term (*of*) occurs $cf_1/2$ times
 - the third most frequent term (*and*) occurs $cf_1/3$ times
 - ...
- Equivalent: $cf_i = a/i$, so
 - $\log cf_i = \log a - \log i$
 - Linear relationship between $\log cf_i$ and $\log i$



↳ there will be a few freq

↳ most freq → rank 1

↳ 2nd most freq → rank 2

tells something about posits

Zipf's law: rank \times frequency \sim constant

English:	Rank R	Word	Frequency f	$R \times f$
	10	he	877	8770
	20	but	410	8200
	30	be	294	8820
	800	friends	10	8000
	1000	family	8	8000

German:	Rank R	Word	Frequency f	$R \times f$
	10	sich	1,680,106	16,801,060
	100	immer	197,502	19,750,200
	500	Mio	36,116	18,059,500
	1,000	Medien	19,041	19,041,000
	5,000	Miete	3,755	19,041,000
	10,000	vorläufige	1,664	16,640,000

CORPUS

Zipf's law examples

Top 10 most frequent words in a large language sample.

English	German	Spanish	Italian	Dutch
the	der	que	non	de
of	die	32,994	25,757	2,709
and	und	32,116	22,988	2,699
to	zu	22,999	22,988	2,699
a	ein	22,313	22,313	2,699
in	in	21,127	17,600	1,999
is	ist	19,214	16,404	1,995
to	zu	16,284	16,404	1,995
it	es	10,875	14,765	1,875
on	an	9,343	14,765	1,875
was	war	9,236	13,915	1,637

rank \times frequency

Posing size estimate \rightarrow Zipf's Law \rightarrow both use
vocab size estimate \rightarrow Heaps Law \rightarrow inverted index

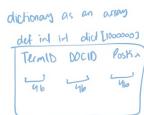
$$M = K T^b \rightarrow 0.4$$

- Is there any relation between Heaps Law and Zipfs Law? Explain

Zipf's law leads to heaps law. The zipf's law concerning the frequencies of individual words(token) in a collection. It state that, if t_1 is the most common term in the collection, t_2 is the next most common, and so on, then the collection frequency $c_{f(i)}$ of the i th most common term is proportional to $1/i$. Mathematically, $c_{f(i)}$ directly proportion to $1/i$. Under mild assumptions, the Heaps law is asymptotically equivalent to Zipf's law concerning the frequencies of individual words within a text. Heaps Law help us in determining how many terms will be there in a collection. It states that the vocabulary size can be estimated as a function of collection size $M = kT^b$ where k is a constant and $b=0.4$ as an exponent. This is a consequence of the fact that the type-token relation (in general) of a homogenous text can be derived from the distribution of its types.

Dictionary Compression

- Written English- 4.5 character / words
- Ave. dictionary word in English: ~8 characters
- Fixed length words for dictionary (20 bytes)
- Fixed length term fields with arrays are certainly wasteful.
- Short words dominate token counts but not type average.



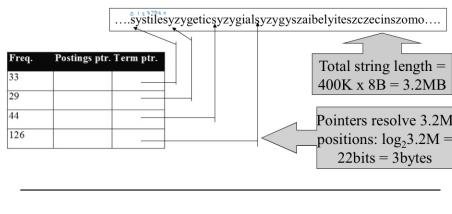
Dictionary Compression –RVC 1

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 20 bytes for each term
- 400K terms x 28 \Rightarrow 11.2 MB

*↓
TO COMPRESS MORE
Save entire array as string*

Dictionary Compression

- Store dictionary as a (long) string of characters:
 - Pointer to next word shows end of current word
 - Hope to save up to 60% of dictionary space.



file : 3-5

*BETTER
use blocks on the strings*

Dictionary Compression –RVC 1

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 400K terms x 19 \Rightarrow 7.6 MB (against 11.2MB for fixed width)

*↓
TO COMPRESS BETTER
as array
as string
as block
as format encoding in block*

Why compression for inverted indexes?

- Dictionary
 - Make it small enough to keep in main memory
 - Make it so small that you can keep some postings lists in main memory too
- Postings file(s)
 - Reduce disk space needed
 - Decrease time needed to read postings lists from disk
 - Large search engines keep a significant part of the postings in memory.
 - Compression lets you keep more in memory
- We will devise various IR-specific compression schemes

Posting Compression

code in bit instead of integer

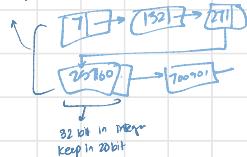
- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800,000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

↳ integers into bits representation

↳ reduce doc ID into bits → To reduce Posting list

↳ use 20 bits per Doc ID

all positive



$$\text{bits: } 4 \times 4 \times 8 \text{ latch} = 32$$

$$\text{now: } 20 \text{ latch} \times 8 \text{ latch} = 20$$

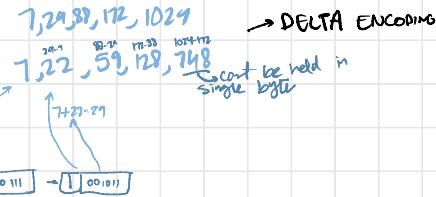
Integers into bit representation

Posting Compression

- - - # - # record ↲

Variable length Encoding

- Aim: For *arachnacentric*, we will use ~20 bits/gap entry.
- For *the*, we will use ~1 bit/gap entry.
- If the average gap for a term is G, we want to use $\sim \log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a *variable length encoding*
- Variable length codes achieve this by using short codes for small numbers



Gamma code ~ 0000 100

9
↓
6 code

12
↓
1 digit
code 1 to most sig fig

1110001 → code 1 at most sig fig

10000010
5
continuation bit

-7- -1-
[0] [1]

137

?

Index Compression

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Only 4% of the total size of the collection
- Only 10-15% of the total size of the text in the collection
- However, we've ignored positional information
- Hence, space savings are less for indexes used in practice
 - But techniques substantially the same.

Summary

- Compression plays an important role for large scale IR systems.

We can create index for high efficient Boolean Retrieval

Agenda

- Hardware Basics
- Blocked Sort-Based Indexing
- Single-Pass In-Memory Indexing
- Distributed indexing
- Dynamic indexing
- Conclusion

4A

Agenda

- Why Compression?
- Heap's Law
- Zipf's Law
- Dictionary compression
- Postings compression
- Summary

4B

Boolean Retrieval Model CONS

- AND ↳ too few or too many results
- OR ↳ every formation is hard

One of the limitation of Boolean Model is the similarity measure it offers.

- Query 1: "standard user dlink 650" → 200,000 hits
- Query 2: "standard user dlink 650 no card found": 0 hits
- It takes a lot of skill to come up with a query that produces a manageable number of hits.
- AND gives too few; OR gives too many

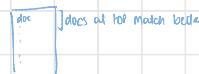
Ranked Retrieval Model

- ↳ returns an ordering over the (TOP) documents in the collection for a query
- ↳ free text queries: users query is just words in human language → not operators or expressions

TWO CHOICES

e.g. India Capital

- ↳ will give documents talking about India



PROS

- ↳ free text queries
- ↳ return of large result set isn't an issue → as JMF show top 10 results

Score Ranking

- ↳ returns documents in order of most useful

- ↳ assign a score [0,1] to each doc

measures how well doc and query match
no match best match

How to assign score to docs

Query Matching

1. One term query

↳ if query term doesn't occur in doc Score = 0

↳ if query term freq in docs Score: higher

1. Jaccard Coefficient

↳ Scoring Scheme

↳ measures overlap of 2 sets

- $\text{jaccard}(A, B) = |A \cap B| / |A \cup B|$
- $\text{jaccard}(A, A) = 1 \rightarrow$ complete overlap
- $\text{jaccard}(A, B) = 0$ if $A \cap B = 0 \rightarrow$ no overlap

- ↳ A and B can be diff size

- ↳ assigns no. b/w '0' and '1'

Query: ides of march

- Document 1: caesar died in march
- Document 2: the long march

$$JC(O, D1) = \frac{1}{6} \quad JC(O, D2) = \frac{1}{5}$$

common terms
6 total terms
lower score

CONS

↳ doesn't consider term freq

↳ term of doc affects score → to need to normalize it

↳ rare terms are more informative than freq terms, JC ignores this

↓ SOLUTION

VECTOR SPACE MODEL

capital of India



2. Parametric Search

Parametric Search

- Documents
 - Data
 - Metadata
- A parametric search provides IR based on Parameters.
- Example:
 - Find documents which contain "computational linguistics" in title, "Manning" in author and "Parameters" in the body of the text.

Example

Search category	Value
Author	Example: Widom, J or Garcia-Molina Also a part of the title possible
Title	Example: 1987-01-01/1987-01-01 limits the search to the documents appeared in, before and after 1987 respectively
Date of publication	Example: 1987-01-01/1987-01-01
Language	Language the document was written in
Project	English
Type	ANY
Subject group	ANY
Sorted by	Date of publication

3. Weighted zone Scoring

- Different fields/zones may have different importance in evaluating how a document matches a query
- For a query q and a document d , weighted zone scoring assigns to pair (q, d) a score in range $[0, 1]$ by computing a linear combination of zone scores
- Suppose each document has I zones, let $g_1, \dots, g_I \in [0, 1]$ such that $\sum_{i=1}^I g_i = 1$
 - Each field/zone of the document contributes a Boolean value – let s_i be the Boolean score denoting a match or absence between q and the i -th zone
 - The weighted zone score is $\sum_{i=1}^I g_i \times s_i$

$$\text{score}(d, q) = g \cdot ST + SB(1-g)$$

Exercise 6.1

When using weighted zone scoring, is it necessary for all zones to use the same Boolean match function? *No, diff zones can employ diff match functions*

Exercise 6.2

In Example 6.1 above with weights $g_1 = 0.2, g_2 = 0.31$ and $g_3 = 0.49$, what are all the distinct score values a document may get?

Example 6.1: Consider the query *shakespeare* in a collection in which each document has three zones: *author*, *title* and *body*. The Boolean score function for a zone takes on the value 1 if the query term *shakespeare* is present in the zone, and zero otherwise. Weighted zone scoring in such a collection would require three weights g_1, g_2 and g_3 , respectively corresponding to the *author*, *title* and *body* zones. Suppose we set $g_1 = 0.2, g_2 = 0.3$ and $g_3 = 0.5$ (so that the three weights add up to 1); this corresponds to an application in which a match in the *author* zone is least important to the overall score, the *title* zone somewhat more, and the *body* contributes even more.

Thus if the term *shakespeare* were to appear in the *title* and *body* zones but not the *author* zone of a document, the score of this document would be 0.8.

ST	SB
0	0
0	1
1	0
1	1

$$g_1 = 0.2$$

$$\text{For } g = 0.2$$

$$\text{for } g = 0.31$$

$$\text{for } g = 0.49$$

$$g_2 = 0.31$$

$$\hookrightarrow 0.2(0) + 0(0.8) = 0$$

$$\hookrightarrow 0.31(0) + 0(0.69) = 0$$

$$\hookrightarrow 0.49(0) + 0(0.51) = 0$$

$$= 0.51$$

$$g_3 = 0.49$$

$$\hookrightarrow 0.2(1) + 0(0.8) = 0.2$$

$$\hookrightarrow 0.31(1) + 0(0.69) = 0.31$$

$$= 0.49$$

$$+$$

$$1$$

$$\hookrightarrow 0.2(1) + 0(0.8) = 1$$

$$\hookrightarrow 0.31(1) + 0(0.69) = 1$$

$$= 1$$

$$\text{ANS: } 0, 0.2, 0.31, 0.49, 0.51, 0.69, 0.8, 1$$

Exercise 6.5

Apply Equation 6.6 to the sample training set in Figure 6.5 to estimate the best value of g for this sample.

Exercise 6.6

For the value of g estimated in Exercise 6.5, compute the weighted zone score for each (query, document) example. How do these scores relate to the relevance judgments in Figure 6.5 (quantized to 0/1)?

Example	DocID	Query	s_T	s_B	Judgment
Φ_1	37	linux	1	1	Relevant
Φ_2	37	penguin	0	1	Non-relevant
Φ_3	238	system	0	1	Relevant
Φ_4	238	penguin	0	0	Non-relevant
Φ_5	1741	kernel	1	1	Relevant
Φ_6	2094	driver	0	1	Relevant
Φ_7	3191	driver	1	0	Non-relevant

► Figure 6.5 An illustration of training examples.

9:02S

$$\Phi_1, s_T=1, s_B=1, R$$

$$S = 0.25(1) + 1(0.75)$$

= 1 Relevant

$$\Phi_2 S = 0 + 0.75$$

$$= 0.75 NR$$

$$\Phi_3 S = 0 + 0.75$$

= 0.75 Relevant

$$\Phi_4 = 0 + 0$$

$$= 0 NR$$

$$\Phi_5 S = 0.25 + 0.75$$

$$= 1 R$$

$$\Phi_6 S = 0 + 0.75$$

$$= 0.75 R$$

$$\Phi_7: S = 0.25 + 0$$

$$= 0.25 NR$$

Learning Weights for Zone Scoring

- Using training examples that have been judged editorially
- Each training example is a tuple consisting of a query q and a document d , and a relevance judgment for d on q
 - The judgment can be binary – relevant or not
 - A judgment score can also be used
- Compute the weights such that the learned scores approximate the relevance judgments as much as possible
 - An optimization problem

Learning Weights for Zone Scoring

We now consider a simple case of weighted zone scoring, where each document has a *title* zone and a *body* zone. Given a query q and a document d , we use the given Boolean match function to compute Boolean variables $s_T(d_j, q)$ and $s_B(d_j, q)$, depending on whether the title (respectively, body) zone of d matches query q . For instance, the algorithm in Figure 6.4 uses an AND of the query terms for this Boolean function. We will compute a score between 0 and 1 for each (document, query) pair using $s_T(d_j, q)$ and $s_B(d_j, q)$ by using a constant $g \in [0, 1]$, as follows:

$$\text{score}(d, q) = g \cdot s_T(d, q) + (1 - g)s_B(d, q).$$

We now describe how to determine the constant g from a set of *training examples*, each of which is a triple of the form $\Phi_j = (d_j, q_j, r(d_j, q_j))$. In each training example, a given training document d_j and a given training query q_j are assessed by a human editor who delivers a relevance judgment $r(d_j, q_j)$ that is either *relevant* or *nonrelevant*. This is illustrated in Figure 6.5, where seven training examples are shown.

Example – Learning Weights

Example	DocID	Query	s_T	s_B	Judgment
Φ_1	37	linux	1	1	Relevant
Φ_2	37	penguin	0	1	Non-relevant
Φ_3	238	system	0	1	Relevant
Φ_4	238	penguin	0	0	Non-relevant
Φ_5	1741	kernel	1	1	Relevant
Φ_6	2094	driver	0	1	Relevant
Φ_7	3191	driver	1	0	Non-relevant

Figure 6.5 An illustration of training examples.

Learning Weights for Zone Scoring

For each training example Φ_j we have Boolean values $s_T(d_j, q_j)$ and $s_B(d_j, q_j)$ that we use to compute a score from (6.2)

$$\text{score}(d_j, q_j) = g \cdot s_T(d_j, q_j) + (1 - g)s_B(d_j, q_j).$$

We now compare this computed score with the human relevance judgment for the same document–query pair (d_j, q_j) ; to this end, we quantize each *relevant* judgment as a 1 and each *nonrelevant* judgment as a 0. Suppose that we define the error of the scoring function with weight g as

$$\varepsilon(g, \Phi_j) = (r(d_j, q_j) - \text{score}(d_j, q_j))^2,$$

where we have quantized the editorial relevance judgment $r(d_j, q_j)$ to 0 or 1. Then, the total error of a set of training examples is given by

$$\sum_j \varepsilon(g, \Phi_j).$$

Learning Weights for Zone Scoring

We begin by noting that for any training example Φ_j for which $s_T(d_j, q_j) = 0$ and $s_B(d_j, q_j) = 1$, the score computed by Equation (6.2) is $1 - g$. In similar fashion, we may write down the score computed by Equation (6.2) for the three other possible combinations of $s_T(d_j, q_j)$ and $s_B(d_j, q_j)$; this is summarized in Figure 6.6.

Let n_{00r} (respectively, n_{01n}) denote the number of training examples for which $s_T(d_j, q_j) = 0$ and $s_B(d_j, q_j) = 1$ and the editorial judgment is *relevant* (respectively, *nonrelevant*). Then the contribution to the total error in Equation (6.4) from training examples for which $s_T(d_j, q_j) = 0$ and $s_B(d_j, q_j) = 1$ is

$$[1 - (1 - g)]^2 n_{00r} + [0 - (1 - g)]^2 n_{01n}.$$

By writing in similar fashion the error contributions from training examples of the other three combinations of values for $s_T(d_j, q_j)$ and $s_B(d_j, q_j)$ (and extending the notation in the obvious manner), the total error corresponding to Equation (6.4) is

Learning Weights for Zone Scoring

$$(n_{00r} + n_{10n})g^2 + (n_{10r} + n_{01n})(1 - g)^2 + n_{00r} + n_{11n}.$$

By differentiating Equation (6.5) with respect to g and setting the result to 0, it follows that the optimal value of g is

$$\frac{n_{10r} + n_{01n}}{n_{10r} + n_{10n} + n_{01r} + n_{01n}}.$$

ishma hafeez
notes
repsht
treeet