Information Retrieval

Muhammad Rafi — 5 years

CS4051 CS517 CS5086

↓

OVERLAP

DB challenges

↳ global application → no support

↳ as all T (transaction) in a single thread

RDBMS

Vector DB → not human readable

↳ new storage

mysql

V: a|x, a|x, a|x

filtering
tabdeli
search

yahoo → google → chatGPT

↓

dont know
authencity of info

Database systems

↳ data collected by some DB management system

↳ structured format → clear semantics based on a formal model

↳ Formal queries → SQL

↳ exact results

↳ RDMS → specific architecture, language

↳ able to diagnose conceptual schema

Information Retrival

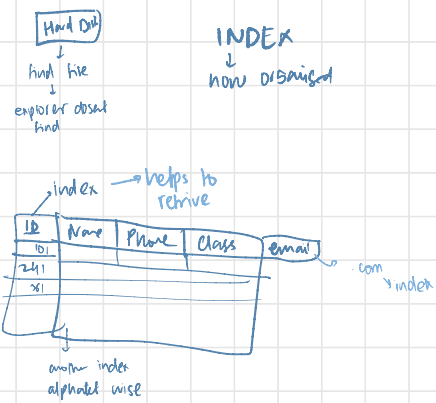
↳ Semi / Un structured → free text

↳ natural language → keyword search

↳ sometimes relavant result

↳ you have documents

↳ try to build interface on that



Data mining

↳ knowledge discovery in DB

↳ analysis step of KDD

↳ extracts info from data set and

↳ transforms to an understandable structure

VS

Text minings

↳ 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 since1994	Broad implementation starting 2000
Market	10^5 analysts at large and mid size companies	10^8 analysts corporate workers and individual users

natural language processor

Information Retrival

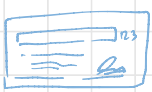
↳ gets info related to a user query from unstructured collections of textual data

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

↳ finding unstructured relavent info

text is considered unstructured here

→ e.g getting credit card no



what is your CNIC NO.? → query

C → look up

AdhOC Retrival 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

VS

Text minins

↳ utilizes NLP based methods for doing activity

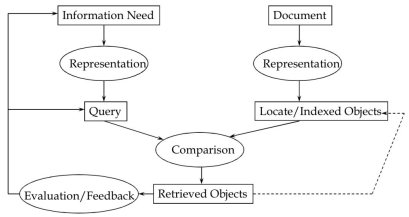
↳ Activity: clustering, classification, summarization, QA

VS

Information Extraction (IE)

↳ automatically extracts structured info from un/semi structured machine readable documents

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	

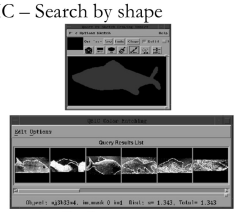
IBM's QBIC

- 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

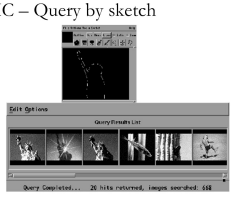
** Images courtesy : Yong Rao

QBIC – Search by shape



** Images courtesy : Yong Rao

QBIC – Query by sketch



** Images courtesy : Yong Rao

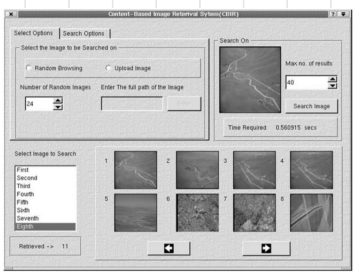
Virage

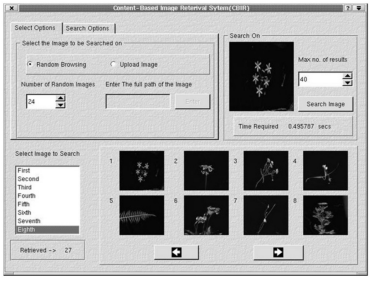
- Developed by Virage inc.
- Like QBIC, supports queries based on color, layout, texture
- Supports arbitrary combinations of these features with weights attached to each
- This gives users more control over the search process

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.

ISearch





Example: Content-based Image Retrieval:

- http://wang.ist.psu.edu/IMAGE

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

Question & Answering Systems

- System that enables the extraction of an answer (or several) to a request (a question) based on a corpus
- An answer or several, possibly a list from one or several documents, an answer of the type Yes/No…,
  - Askjevees.com
  - Easyask.com
  - Answerlogic.com

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

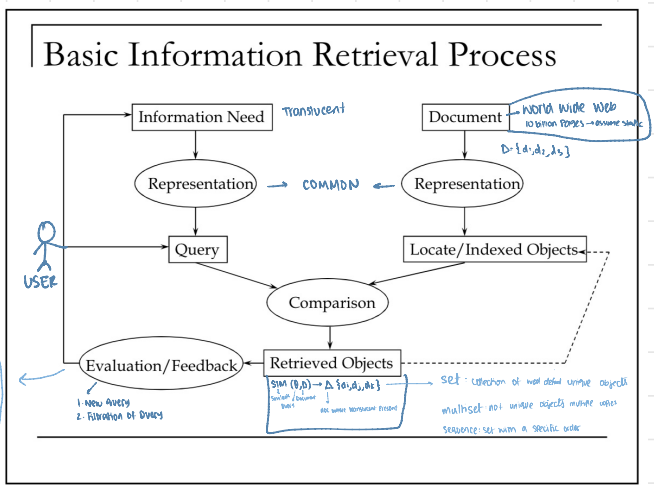
BOOLEAN RETRIVAL

MODEL

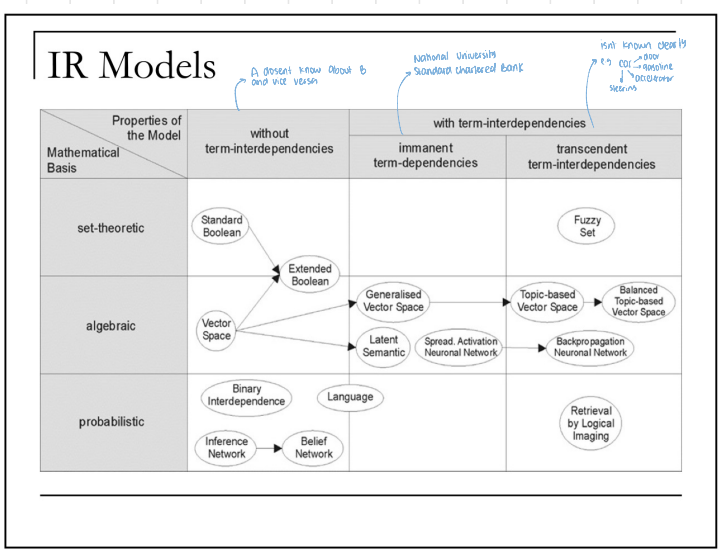
- an abstract representation of a process/object
- used
  - to study properties
  - draw conclusions → quality of conclusions depends how much the model is close to reality
  - make predictions

Retrival Model

- describes the human and computational process involved in ad-hoc retrival
  - **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



eg Jaguar is a car and animal so feedback can be used to bring more accuracy



Retrival Complexies

Basic Information Retrieval Setting

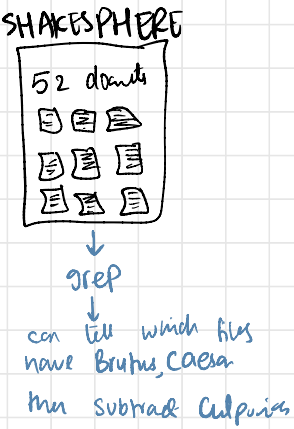
- Offline / Online
  - Corpus → many diff documents
  - Dynamic /changing collection (www)
- Scale of operation
  - World wide web level → as any search → faster than file explorer
  - Personal / individual level → bounded size
  - Enterprise / domain specific level → diff format, files
- Type of Content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

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 the 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 ….



Term matrix document

↳ counts
↳ add Brutus, Caesar
↳ compliment Calpurnia
↳ add both

Problems

↳ bad for dynamic → as dim + term both increase ← if users changes on the go → changing dim
↳ 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

PRD

↳ 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	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
…							

Figure 1.1 A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 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 AND 110111 AND 101111 = 100100

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

THIS IS THE PROBLEM

SOLUTION
↳ INVERTED INDEX

Dictionary

Posting

decide which one goes first

q_i: $(w_1$ AND $w_2)$ OR $(w_3$ AND $w_4)$ AND $(w_5$ OR $w_5)$

→ PROS
↳ Dynamic
↳ easy to process

d1: Ceaser

d2: Ceaser Ceaser

term frequency: 3

document frequency: 2

BOOLEAN MODEL

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

$|D| = n$ is a long number

$d_i = \{t \dots t_m\}$

$m \ll$ small number compound to n

es D: 1 million

document size ← d_i : 5000 → distinc : 1000

term size ← $|t| = 500k$

0.5 billion

X million

↓

0.1 billion

Term Doc Matrix → 2 dim

90% sparsity

t_1 d_1 d_2 . . . . . → bigger

small ↓ t_n

URL (http://users/text/1.html)

Doc ID	URL
1	URL

term	
t_1	1
t_2	2

numerical so lookup

Effectiveness of IR system

↳ based on quality of its search results

↳ TWO factors

- Precision
  - What fraction of the returned results are relevant to the information need?
- Recall
  - What fraction of the relevant documents in the collection were returned by the system?

Prescion

$$\frac{\text{Relavent Ret}}{\text{Total Ret}}$$

challanging for small data set

Recall

$$\frac{\text{Relavent Ret}}{\text{Relavent}}$$

not good for high Performance retrival systems

is orthogonal

100% Precision → gives all data set

100% Recall → Retrive 1 documat that is also relavent

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

Precision = relevant –retrieved / total retrieved

Recall = relevant –retrieved / total Relevant in collection

Example:

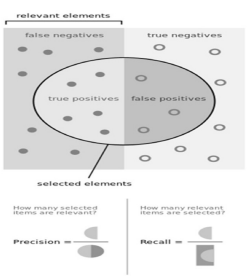


Image source : Wikipedia

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

8) Rel : 20
Ret = 8
IRR = 10

$$\text{Precision} = \frac{8}{10+8} = 0.44$$

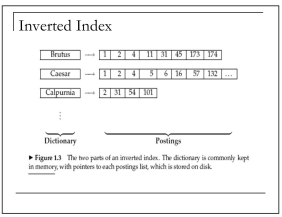
$$\text{Recall} = \frac{8}{20} = 0.4$$

Inverted Index

www

↳ index of vocabulary that occurs in a collection

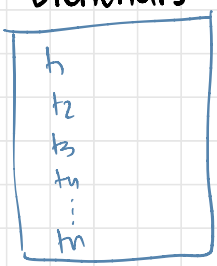
↳ has 2 parts



▶ Figure 1.3 The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

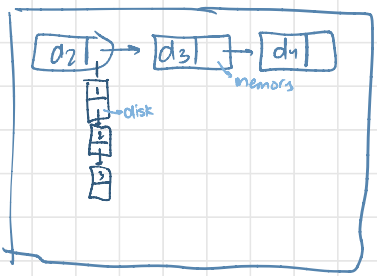
Dictionary

words / terms / tokens / lexeme



Posting

document info / where a term belongs



$\log_2 n$

$\log_2 500k$

$\approx (18 \text{ to } 19)$

PROS

↳ less sparcity

↳ quick acess

↳ uses NLP to solve problem of complex query formation

Example

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

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

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

⟹

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

=

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

ceaser freq

Term freq = 3 → mentioned 3 times in all documents

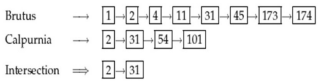
document freq = 2 → mentioned in 2 documents

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



Intersect (Posting List Algorithm)

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

Q: t1 ∧ t2

↳ if not same
  ↳ move smaller forward
↳ if same
  ↳ add doc to result

t1: 1 3 7 29
t2: 29 31 41 51

Good for bi, 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 posting 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.

Query Processing total work

↳ Parse Query

↳ locate term in dictionaricy

↳ collect posting list

↳ intersection algo

Intersect (Optimized)

```
INTERSECT(⟨t1, ..., tn⟩)
1 terms ← SORTBYINCREASINGFREQUENCY(⟨t1, ..., tn⟩)
2 result ← postings(first(terms))
3 terms ← rest(terms)
4 while terms ≠ NIL and result ≠ NIL
5 do result ← INTERSECT(result, postings(first(terms)))
6    terms ← rest(terms)
7 return result
```

↳ sort by increasing freq for optimization

→ SORTED

term	doc ID

term	doc ID

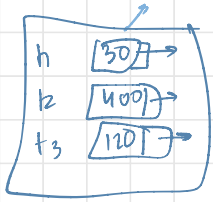
term	doc freq	→	Postings

↳ $t_1 \wedge t_2 \wedge t_3$

↳ $t_1 \wedge t_3 \wedge t_2$

less frequency so better

cardinality → no of times came in doc



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

Documents: set of terms

Queries: Boolean expression on terms

ADV

↳ clean formation, easy to implement → we know boolean algebra

↳ results are predictable and easily explainable

↳ many diff features can be incorporated

CONS

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

↳ Query formation is hard → info has to be translated by Boolean expression

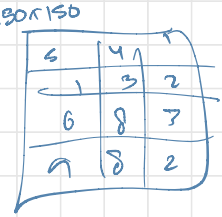
↳ All terms are equally weighted

↳ flat results

↳ returns either too few or too many doc

Images

150 × 150



→ use this feature

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 (in terms of the number of paying subscribers), with 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 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 disclos! /s prevent /s employe!

Information need: Requirements for disabled people to be able to access a workplace.
Query: disab! /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

CHP 2

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 accomodate partial matching and a ranking has attracted considerable attention in the past
- Two extensions of boolean model:
  - Extended Boolean Model
  - Fuzzy Set Model

Extended Boolean Model

Extended Boolean Model

- Proximity Search
- Ranked Retrieval
- Example
  - WestLaw

Westlaw

Pros

unranked Boolean results → freq of term as a weight → gives ranks

Westlaw

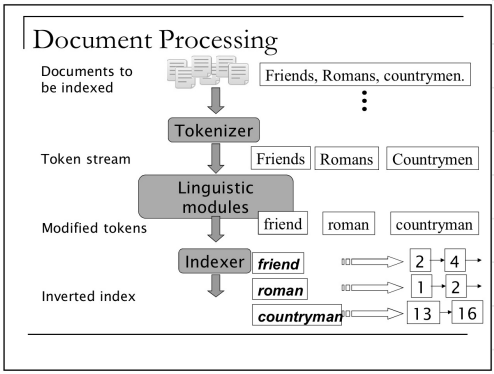
Westlaw – Commercial Systems

- 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 employe!

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



Words: delimited string of characters (in b/w a space)

Type: same meaning diff word

↳ an equivalance 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

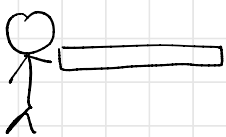
e.g word token, term token may not be your feature

token

Equivalance class: prefix, suffix of words

e.g operati, operation, operator

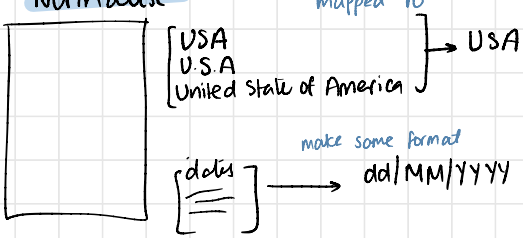
Document Processing



→ words = Tokenization(file, "-----") (done on space here)

= I, am, Ishma, I, am, Studying

→ Normalise



morphological changes: Wi + add (prefix, suffix)

noise → noisy

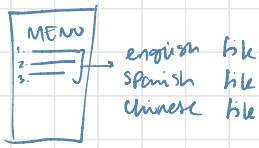
run → runs

Challanges in DOC PROCESSING

↳ what format : PDF/word/excel/HTML

↳ what language

language meta data



linearization → chop in single/multi byte

classification Problems

↳ what character set : UTF-8/CP1252

↳ Format/Language/Encoding

↳ Documents

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

↳ Tokenization

↳ issues in Tokenization

Document size → string file

short doc 1k → 500k

long doc 500k → 1 GB → chunks / buffer → optimization

Tokenization

↳ a process through which docs are parsed and a seqeuence of characters seperated

- 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

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

استقلت الجزائر في سنة 1962 بعد 132 عاما 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 foldings

↳ 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…

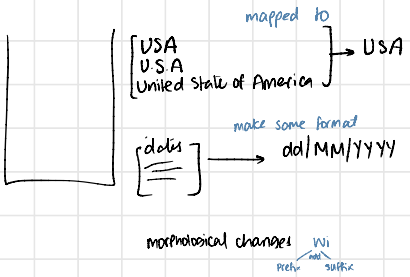
Normalization

↳ normalize words into same form

↳ remove periods U.S.A → USA

↳ remove hypens

↳ 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: ***Tuebingen*** vs. ***Tübingen***
- 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*** → ***Tubingen***

expand words

Sec. 2.2.3

Normalization to terms

- An alternative to equivalence classing is to do asymmetric expansion (suffix, prefix of a word)
- 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 Soundex, which forms equivalence classes of words based on phonetic heuristics

Sounder

Dictionary

↳ alphabetical order

↳ through details on

- ↳ meaning
- ↳ definition
- ↳ usage
- ↳ etymology → origins
- ↳ pronounce

↳ used to learn word and its meaning

Thesauras

↳ conceptual order

↳ gives

- ↳ synonyms
- ↳ antonyms
- ↳ relations with words
- ↳ language usage

↳ used to know a diff word for a given word

20 sec

Stemming

→ undo lexical and morphological changes

- 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
- algorithmic approach
- to undo morphological changes

CONS

- sometimes stem is not human readable
- overstemming → sometimes chop too much
  - lower precision
  - more recall

PROS

- extremly fast

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

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

word = {abort}

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

semi-vowel = {y, w}

constant = {the rest}

$[c|v]^* \underbrace{[vc]}_{m}^* [c|v]^*$ → verified lexical words

Tree
c c v v
m=0

rotation
c vc vc v v c
m=3

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

Inverted Index

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 “I” if word has a vowel
  - satisfy --> satisfi, fly --> fli

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 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	
ical		ic tropical --> tropic
ful	--	faithful --> faith
iveness	ive	inventiveness --> inventive
ness	--	harness --> har

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

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

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

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

Lemmatization

2 min

↳ takes every word to dictionary to match term
↳ output is human redable
↳ doing proper reduction
↳ reduce variant forms to base forms

eg am, are, is ⟶ be
cars, car, car's, cars' ⟶ 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

↙ than stemming

Stemming VS Lemmatization

↳ operate 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

Morphology

↳ Study of the formations of word in a language

Inflectional

↳ adding a suffix/prefix doesn't changes it grammatical category

↳ 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 it grammatical category

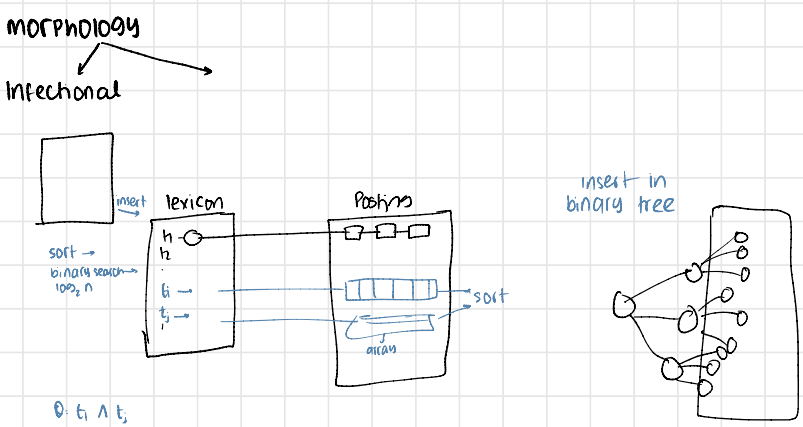
↳ formation of new word

↳ use of prefix/suffix common

↳ semantically irregular

eg Activation → reactivation

National → antinational



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

eg "Stanford University"

bigram → contains 2 terms

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

Sec. 2.4

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 also the more the processing

↳ 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 ∝ no of docs in result set

Has 2 solutions

SOL 1

Bi-word indexes

simple phrases

↳ 2 word pairs indexed as a Phrase

↳ use 2 words as a feature

eg Stanford University

Ishma Hafeez

CON

↳ false positives

↳ index blow up → bigger dictionary → by 40% for english

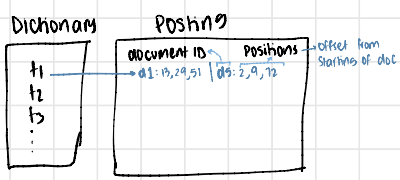
PROS avoids Post Processing

SOL 2

Positional Index

↳ an inverted index

↳ Posting list has document ID and Positions



2. locate term in inverted index

3. Fetch Postings list

4. find adjacent terms 15,16,17

CON

↳ Posting list storage increase substantially

↳ intersection also 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.

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.>

Positional index example

<***be***: 993427;
1: 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

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(p1, p2, k)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then l ← ⟨ ⟩
 5             pp1 ← positions(p1)
 6             pp2 ← positions(p2)
 7             while pp1 ≠ NIL
 8             do while pp2 ≠ NIL
 9                do if |pos(pp1) − pos(pp2)| ≤ k
10                      then ADD(l, pos(pp2))
11                      else if pos(pp2) > pos(pp1)
12                              then break
13                   pp2 ← next(pp2)
14                while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
15                do DELETE(l[0])
16                for each ps ∈ l
17                do ADD(answer, ⟨docID(p1), pos(pp1), ps⟩)
18                pp1 ← next(pp1)
19             p1 ← next(p1)
20             p2 ← next(p2)
21        else if docID(p1) < docID(p2)
22                then p1 ← next(p1)
23                else p2 ← next(p2)
24  return answer
```

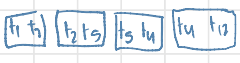
Longer phrase Queries → general phrases

↳ no limit on number of terms

$t_1, t_2, t_3, t_4, t_{12}$

↓

break into bi word phrase queries → Query Expansion



CON

↳ can have false positives → so need to do post processing

Extended bi-word index

↳ performs parts of speech tagging (POST)

↳ bucket terms into N NOUNS and X Prepositions

↳ N X* N strings are extended biword

↳ it is now a term in dictionary

e.g coin (t_1) in the pocket (t_2) → feature

→ Q 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***

Sec. 2.4.1

Longer phrase queries

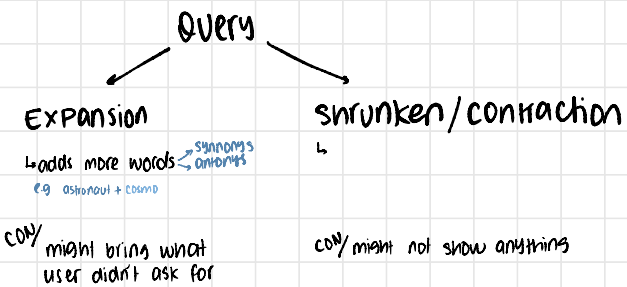
- Longer phrases are processed as we did with wild-cards:
- ***stanford university palo alto*** can be broken into the Boolean query on biwords:

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!

* google only includes 10 terms in the query



911 → delta index → last 12 hours news

Proximity Queries

1. $w_1 w_2 / 2$ → $w_1 w_2$, $w_1 x w_2$, $w_1 x y w_2$
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 IIR
  - There's likely to be a problem on it!

Positional index size

↳ Positional index expands postings storage substantially

↳ it is standardly used because 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 on 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 ¼ of the time of using just a positional index
  - It required 26% more space

Posting is just like a linklist

Skiplist → Probabilistic data structure

↳ specialized link list
↳ improved searching
↳ is of TWO Types
  ↳ Homogenious
  ↳ Herrogenious
} Node

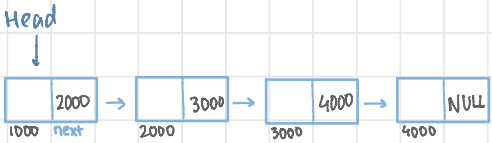
CON
↳ bad for t_1 **OR** t_2

PRO
↳ good for t_1 **AND** t_2

no compromise on storage

Singly Linked List

	insertion	searchon
1. unordered SLL	O(1)	O(n)
2. ordered SLL	O(n)	$\log_2 n$



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

insersion: O(n) searching: $\log_2 n$

Homogenious node

↳ sorted order
↳ all nodes identical
↳ searching faster by skipping pointers

2^0, 2^1, 2^2 ... 2^{10} → pointers
2^0 connects every node
2^1 skips 2 nodes
2^2 skips 4
2^{10} connects last node

[1000]
$2^n \leq 1000$
$n = 10$

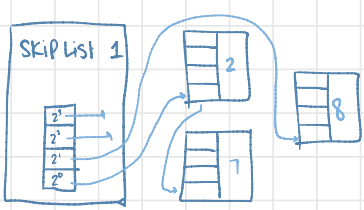
1000 elements
1 integer: 4 bytes
1000 × 4 =

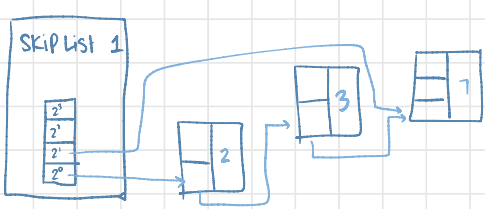
cons
↳ wastefull memory → every node identical and every node needs 10 pointers

SOLUTION

Probalistic node structure

↳ give every node atleas 1 pointer
↳ 50% nodes 2 pointer
↳ 25% nodes 3 pointers
↳ 12.5% nodes 4 pointers

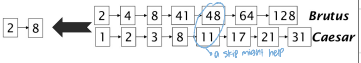




Sec. 2.3

Recall basic merge

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

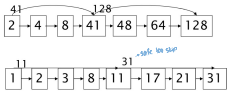


If the list lengths are *m* and *n*, the merge takes O(*m*+*n*) operations.

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

Sec. 2.3

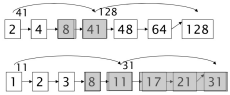
Augment postings with skip pointers (at indexing time)



- 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



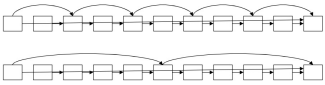
Suppose we've stepped through the lists until we process **8** on each list. We match it and advance.

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 → shorter skip spans ⇒ more likely to skip. But lots of comparisons to skip pointers.
  - Fewer skips → few pointer comparison, but then long skip spans ⇒ few successful skips.



Faster Skip Lists

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

► **Figure 2.10** Postings lists intersection with skip pointers.

More skips → shorter skips
↳ too many pointer comparisions

Fewer skips → long skips
↳ less pointer comparisions
↳ few successfull skips

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 → Boolean queries
→ Proximity queries

Positional indexes → genral Phrase queries
→ Proximity queries

modify intersection of posting list to speed up in generating result set
↳ 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

↳ rehashing needed if vocab keeps growing

Tree

↳

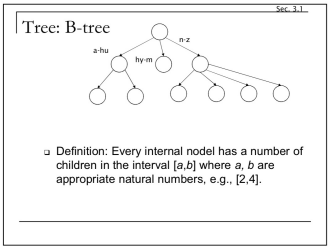
PROS

↳ solves prefix problem

CONS

↳ slower

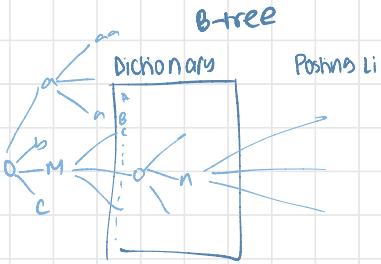
↳ Rebalancing binary tree is expensive



B-tree: interal 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 pronunciation of foreign word/phrase

Trailing Wildcard Queries

mon*: words starting with mon

↳ Binary tree

↳ B-tree

mon ≤ w < moo

Leading wildcard Queries

*mon: words ending with mon

↳ maintain another reverse B-tree

nom ≤ w < non

General Wildcard Queries

co*tion , re*ta*tion

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

→ Solution
Premuterm Index

WHAT?

CONS

↳ too many false positives → ?

↳ too expensive

⇓

SOLUTION

↳ make subcases

↳ no of queries and cost increases

↳ false positives reduced

→ ?

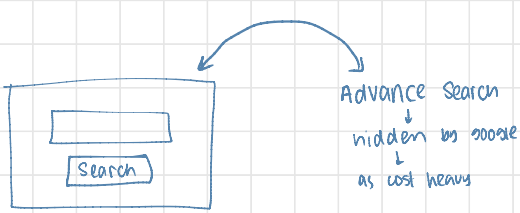
⇓

SOLUTION

1. Permuterm index
2. Bigram index

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.

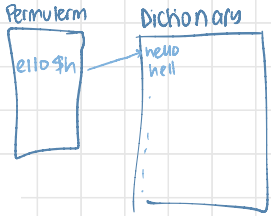


Permuterm index

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

Hello
hello $
ello $h
llo $he
lo $hel
o $hell

he * o



Permuterm Query Processing

↳ rotate wildcard query to the right

↳ use B-tree for look up

- Queries:
  - X lookup on X$ X* lookup on $X*
  - *X lookup on X$* *X* lookup on X*
  - X*Y lookup on Y$X* X*Y*Z ??? Exercise!

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

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 → mo, on → bigram → k=2

↳ maintain a 2nd inverted index from bigrams to dictionary terms that match each bigram

↳ $ is a word boundary

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

  a,ap,pr,ri,il,l,i,is,s,t,th,he,e,c,cr,ru,ue,el,le,es,st,t, m,mo,on,nt,h

  - $ is a special word boundary symbol

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…)
  - pyth* AND prog*
- If you encourage "laziness" people will respond!

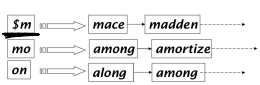
Search

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

Sec. 3.2.2

Bigram index example

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



PROS

↳ fast

↳ space efficient

CON

↳ expensive query execution

Spelling Corrections

↳ Two uses

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

↳ Two main flavours

1. Isolated word
   - ↳ checks each words misspelling indivisually e.g fait → fail
   - ↳ won't correct typos spelled right e.g from → form

Spelling Corrections

Return to Google's jobs pages

488941 britney spears	29 britent spears	9 brinttany spears	5 brney spears	3 britiy spears	2 brirreny spears
40134 brittany spears	29 brittnany spears	9 britanay spears	5 broitney spears	3 britmeny spears	2 brirtany spears
36315 brittney spears	29 britttany spears	9 britinany spears	5 brotny spears	3 britneeey spears	2 brirttany spears
24342 britany spears	29 btiney spears	9 britn spears	5 bruteny spears	3 britnehy spears	2 brirttney spears
7331 britny spears	26 birttney spears	9 britnew spears	5 btiyney spears	3 britnely spears	2 britain spears
6633 briteny spears	26 breitney spears	9 britneyn spears	5 btrittney spears	3 britnesy spears	2 britane spears
2696 britteny spears	26 brinity spears	9 britrney spears	5 gritney spears	3 britnetty spears	2 britaneny spears
1807 briney spears	26 britenay spears	9 brtiny spears	5 spritney spears	3 britnex spears	2 britania spears
1635 brittny spears	26 britneyt spears	9 brtittney spears	4 bittny spears	3 britneyxxx spears	2 britann spears
1479 brintey spears	26 brittan spears	9 brtny spears	4 bnritney spears	3 britnity spears	2 britanna spears
1479 britanny spears	26 brittne spears	9 brytny spears	4 brandy spears	3 britntey spears	2 britannie spears
1338 britiny spears	26 btittany spears	9 rbitney spears	4 brbritney spears	3 britnyey spears	2 britannt spears
1211 britnet spears	24 beitney spears	8 birtiny spears	4 breatiny spears	3 britterny spears	2 britannu spears
1096 britiney spears	24 birteny spears	8 bithney spears	4 breetney spears	3 brittneey spears	2 britanyl spears
991 britaney spears	24 brightney spears	8 brattany spears	4 bretiney spears	3 brittnney spears	2 britanyt spears
991 britnay spears	24 brintiny spears	8 breitny spears	4 brfitney spears	3 brittnyey spears	2 briteeny spears
811 brithney spears	24 britanty spears	8 breteny spears	4 briattany spears	3 brityen spears	2 britenany spears
811 brtiney spears	24 britenny spears	8 brightny spears	4 brieteny spears	3 briytney spears	2 britenet spears
664 birtney spears	24 britini spears	8 brintay spears	4 briety spears	3 brltney spears	2 briteniy spears
664 brintney spears	24 britnwy spears	8 brinttey spears	4 briitny spears	3 broteny spears	2 britenys spears
664 briteney spears	24 brittni spears	8 briotney spears	4 briittany spears	3 brtaney spears	2 britianey spears
601 bitney spears	24 brittnie spears	8 britanys spears	4 brinie spears	3 brtiiany spears	2 britin spears
601 brinty spears	21 biritney spears	8 britley spears	4 brinteney spears	3 brtinay spears	2 britinary spears
544 brittaney spears	21 birtany spears	8 britneyb spears	4 brintne spears	3 brtinney spears	2 britmy spears
544 brittnay spears	21 biteny spears	8 britnrey spears	4 britaby spears	3 brtitany spears	2 britnaney spears
364 britey spears	21 bratney spears	8 britnty spears	4 britaey spears	3 brtiteny spears	2 britnat spears
364 brittiny spears	21 britani spears	8 brittner spears	4 britainey spears	3 brtnet spears	2 britnbey spears
329 brtney spears	21 britanie spears	8 brottany spears	4 britinie spears	3 brytiny spears	2 britndy spears
269 bretney spears	21 briteany spears	7 baritney spears	4 britinney spears	3 btney spears	2 britneh spears
269 britneys spears	21 brittay spears	7 birntey spears	4 britmney spears	3 drittney spears	2 britneney spears
244 britne spears	21 brittinay spears	7 biteney spears	4 britnear spears	3 pretney spears	2 britney6 spears
244 brytney spears	21 brtany spears	7 bitiny spears	4 britnel spears	3 rbritney spears	2 britneye spears
220 breatney spears	21 brtiany spears	7 breateny spears	4 britneuy spears	2 barittany spears	2 britneyh spears
220 britiany spears	19 birney spears	7 brianty spears	4 britnewy spears	2 bbbritney spears	2 britneym spears
199 britnney spears	19 brirtney spears	7 brintye spears	4 britnmey spears	2 bbitney spears	2 britneyyy spears
163 britnry spears	19 britnaey spears	7 britianny spears	4 brittaby spears	2 bbritny spears	2 britnhey spears
147 breatny spears	19 britnee spears	7 britly spears	4 brittery spears	2 bbrittany spears	2 britnjey spears
147 brittiney spears	19 britony spears	7 britnej spears	4 britthey spears	2 beitany spears	2 britnne spears
147 britty spears	19 brittanty spears	7 britneyu spears	4 brittnaey spears	2 beitny spears	2 britnu spears

2. Context sensitive
   - ↳ looks at surrounding words e.g I flew form Karachi to Lahore

Document Correction

- Especially needed for OCR'ed documents
  - Correction algorithms are tuned for this: "rn" / "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

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

Spelling correction → search
Spelling correction → word processors

- ↳ identifying the misspelled words
- ↳ fetch correct words
- ↳ suggest

↳ 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
- ↳

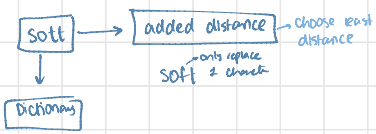
Edit distance

↳ tells you how to convert 1 string into another
↳ min no of ops to convert S1 to S2

Operations are
↳ Insert
↳ Replace
↳ Delete

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

soft tissue



A: "horse"
B: "ros"

1 Replace: "rorse"
1 Delete: "rose"
1 Delete: "ros"
3 operations

Edit Distance – Levenshtein

```
EDITDISTANCE(s1, s2)
 1  int m[i, j] = 0
 2  for i ← 1 to |s1|
 3  do m[i, 0] = i
 4  for j ← 1 to |s2|
 5  do m[0, j] = j
 6  for i ← 1 to |s1|
 7  do for j ← 1 to |s2|
 8       do m[i, j] = min{m[i − 1, j − 1] + if (s1[i] = s2[j]) then 0 else 1fi,
 9                        m[i − 1, j] + 1,
10                        m[i, j − 1] + 1}
11  return m[|s1|, |s2|]
```

cost for 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	1 2	2 3	3 4	4 5
a	2 2	2 1	2 2	3 3	4 4
t	2 2	2 2	1 3	3 4	4 5
s	3 3	3 3	3 2	2 3	2 4
	4 4	4 4	4 3	2 3	3 3
	4 4	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 three numbers whose minimum yields the fourth. The cells in italics determine the edit distance in this example.

		¢	f	a	s	t
	¢	0	1 (min)	2	3	4 → as it is
	f	0	0	0		
	r	1	•			
	e	2				
	s	3				
	n	4				

edit distance b/w "f" an "fr"

↓ as it is

◎ → the edit distance b/w these two strings

Using Edit Distance

↳ enumerate all char seqs within a preset edit distance eg 2
(mentions one by one separately)
↳ check how many of these are present in lexicon
↳ then choose which has most hits in corpus (dataset)
↳ 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	" "	b	e	n	y	a	m
" "	0	1	2	3	4	5	6 → write as is in string
e	1	1	1 (same) → 2	3	4	5	
p	2	2 (same)	2	2 → 3	4	5	
h	3	3	3	3	3 → 4	5	
r	4	4	4	4	4	4 → 5	
e	5	4 (match)	5	5	5	5	
m	6	5	5	6	6	6	5 (match)

↓ write as is in string

↳ no of operation = 5

Operations
R I
D
min of 3 cells → if doesn't match
min of 3 cells → if matches

N-gram Overlaps

↳ enumerate all the n-gram in query

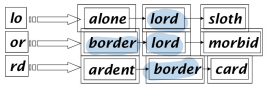
↳ use n-gram index to retrieve all lexicon terms matching any of the query n-grams → wild card query search k-gram

↳ Threshold by no of matching n-grams

variants: weight by keyboard layout, etc

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 …

we define

Answer hit based on threshold

border, lord

eg threshold overlap 2

load

lo
oa
ad

Loat

lo
oa
at

→ 2 same

n-gram overlap: Jackard

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

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 w2

↳ correct but not found any phrase

1. incorrect
2. wrong order

$2 \times 1 \times 3$

Context Sensitive Spelling Corrections

↳ retrive all dic terms close to each query term → use weight edit distance

- 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 F400R
   - 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>. F460

F 4 6 0

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 → not only names

as many things will be 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

→ mapped to the same posting list

CHP 4

ORS + asynmox → 2 days

Hardware Basics

↳ acess to data in memory is much faster than acess to data on disk

Disk seeks: no data is transferred from disk while the disk head is being positioned

- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
- Block sizes: 8KB to 256 KB.

Hardware Basics (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	$5\text{ ms} = 5 \times 10^{-3}$ s
b	transfer time per byte	$0.02\ \mu\text{s} = 2 \times 10^{-8}$ s
	processor's clock rate	$10^9\ \text{s}^{-1}$
p	lowlevel operation (e.g., compare & swap a word)	$0.01\ \mu\text{s} = 10^{-8}$ 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

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.

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.

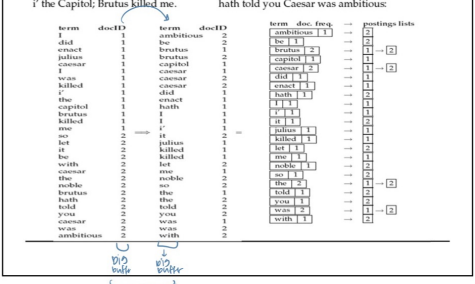
Basic Inverted Index

- 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)

Example

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

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



Token = 10 million : 4 lakh

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

→ Parser → token →

Sort all terms

Challange: keep huge data in 1 buffer and sort that buffer

Term ID	Term Doc ID
4 bytes	4 bytes

1 term = 12 bytes

RCV1 has 400,000 terms so 48 million bytes

TO SOLVE CHALLANGE

1. BSBI
2. SPIMI

→ sorting with fewer disk seeds

Blocked sort Based Indexing (BSBI)

↳ Take buffer into MM
↳ 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.

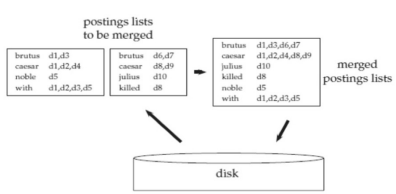
Blocked Sort-Based Indexing

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

killer instance

▶ **Figure 4.2** Blocked sort-based indexing. The algorithm stores inverted blocks in files $f_1, \ldots, f_n$ 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

- BSBI Complexity
  - How expensive is BSBI?
  - Its time complexity is Q(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

↳ do not 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

placement new()

all objects will be made here

Single-Pass In-Memory Indexing

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 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     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

linked list

buffer holding list

is buffer filled

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

PROS

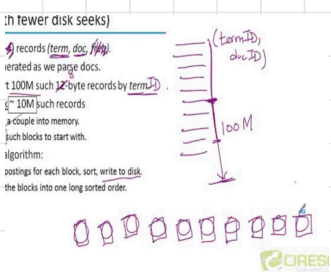
↳ no need to sort

↳ will be complete inverted index

BSBI

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

termID - docID



VS

SMIMI

↳ adds a Postings directly to its Postings list

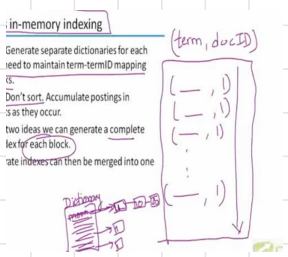
↳ no sorting

↳ faster → as no sorting

↳ saves memory

↳ each Posting 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.



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

pool.

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

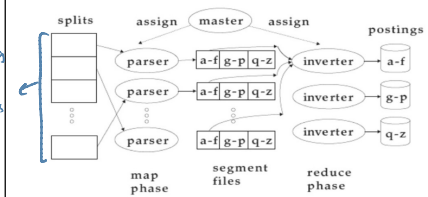
Distributed Indexing

- 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
    - Parser writes pairs into *j* partitions
  - ~~Indexer~~ Inverters
    - An inverter collects all (term,doc) pairs (= postings) for one term-partition.
    - Sorts and writes to postings lists

PRO
- no disk failure

generates a stream of normalized (term, docID) pairs

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	→ list(k, v)
reduce: (k,list(v))	→ output

Instantiation of the schema for index construction

map: web collection	→ list(termID, docID)
reduce: ((termID$_1$, list(docID)), (termID$_2$, list(docID)), ...)	→ (postings_list$_1$, postings_list$_2$, ...)

Example for index construction

map: d_2 : C died. d_1 : C came, C c'ed.	→ ((C, d_2), (died,d_2), (C,d_1), (came,d_1), (C,d_1), (c'ed,d_1))
reduce: ((C,d_2,d_1,d_1), (died,(d_2)), (came,(d_1)), (c'ed,(d_1)))	→ ((C,d_1:2,d_2:1)), (died,(d_2:1)), (came,(d_1:1)), (c'ed,(d_1:1)))

► **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 c'ed.

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

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

Dynamic Index

insertions

↳ maintains big main index

↳ new docs go into small auxiliary index

newly appearing indexes

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

↳ take results from both

↳ then combine them

when becomes too large merge with main index

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

- Google Dance
  - Approach is same as in dynamic Indexing
  - Search engines do incremental (auxiliary posting list) plus some time full indexing in back ground.

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 unit 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 tasks 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

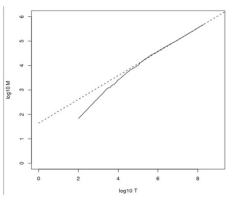
→ estimate dictionary size

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

$M = kT^b$

(vocabulary size, as term of tokens)

- 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 larger dictionary
• dictionary is continous

Zipf's law

most freq word for k constant → 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* .
- $cf_i \propto 1/i = a/i$ where *a* is a normalizing constant
- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

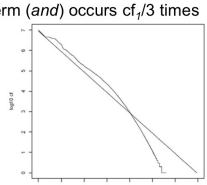
↳ there will be a rs bnw freq

↳ most freq → rank 1

↳ 2nd most freq → rank 2

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 $\log i$



tells something about Postings

Zipf's law: rank x frequency ~ 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	
1	the	61,847	der	7,377,879	que	32,894	non	25,757	de	4,770
2	of	29,391	die	7,036,092	de	32,116	di	22,868	en	2,709
3	and	26,817	und	4,813,169	no	29,897	che	22,738	het/'t	2,469
4	a	21,626	in	3,768,565	a	22,313	è	18,624	van	2,259
5	in	18,214	den	2,717,150	la	21,127	e	17,600	ik	1,999
6	to	16,284	von	2,250,642	el	18,112	la	16,404	te	1,935
7	it	10,875	zu	1,992,268	es	16,620	il	14,765	dat	1,875
8	is	9,982	das	1,983,589	y	15,743	un	14,460	die	1,807
9	to	9,343	mit	1,878,243	en	15,303	a	13,915	in	1,639
10	was	9,236	sich	1,680,106	lo	14,010	per	10,501	een	1,637

rank x frequency ≈

Posting size estimate → ZIPS LAW
Vocab size estimate → HEAPS LAW
→ both use inverted index

↳ $M = kT^b \rightarrow 0.4$

i. Is there any relation between Heaps Law and Zipfs Law? Explain

Zipfs law leads to heaps law. The zipfs law concerning the frequencies on individual wrods(token) in a collection. It state that, if t1 is the most common term in the collection, t2 is the next most common, and so on, then the collection frequency cfi of the ith most common term is proportional to 1/i. Mathematically, cfi 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 = kTb 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 as an array
def int int char[20000000]
TermID DOCID Posn
4b 4b 4b

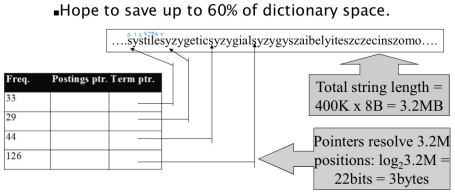
Dictionary Compression –RVC 1

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 20 bytes for each term
- 400K terms x 28 ⇒ 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→3

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 ⇒ 7.6 MB (against 11.2MB for fixed width)

Now avg. 11 bytes/term, not 20.

TO COMPRESS BETTER
as array
as string
as BIDCLs in string
as fronted coding 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.

Posting Compression

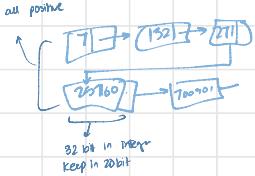
- Variable length Encoding
  - Aim:
    - For ***arachnocentric***, we will use ~20 bits/gap entry.
    - For ***the***, we will use ~1 bit/gap entry.
  - If the average gap for a term is *G*, we want to use ~$\log_2 G$ bits/gap entry.
  - Key challenge: encode every integer (gap) with about as few bits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

↳ integers into bits representation

↳ reduce docID into bits → To reduce posting list

↳ use 20 bits per DOC ID

all positive



bits: 4 × 4 × 8 latih = 32

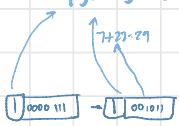
now: 20 120 × 8 latih = 20

integers into bit representation

7, 29, 89, 172, 1024 → DELTA ENCODING

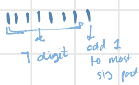
7, 22, 59, 128, 748 (29-7, 89-29, 172-89, 1024-172)

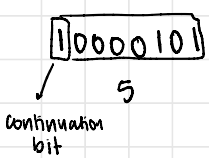
→ can be held in single byte

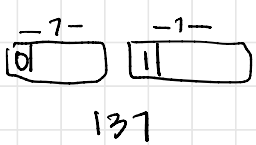


Gamma code:- 9 → 00001001 → 0 code

12







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 Retrival

Agenda

4a

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

Agenda

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

4b

Boolean Retrival Model cons

↳ too few or too many results
AND OR

↳ Query formation is hard

Score Ranking

Ranked Retrival

↳ returns an ordering over the (top) documents in the collection for a query

↳ free text queries: users query is just words in human language → no operators or expressions

TWO CHOICES