



NM 100

1923 - 2014



**Dr. MAHALINGAM**  
COLLEGE OF ENGINEERING AND TECHNOLOGY

Udumalai Road, Pollachi, Coimbatore District 642003

Estd. 1998 | AICTE Approved | Affiliated to Anna University

An Autonomous Institution



**NAAC A++ GRADE**  
Cycle 2 (2018-2023)  
The Highest Grade

# 19CSEN2003 - Information Retrieval Techniques

## (UNIT 1 - Session 1)

**UNIT 1 : Text Retrieval Systems**

**TOPIC : Boolean retrieval**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

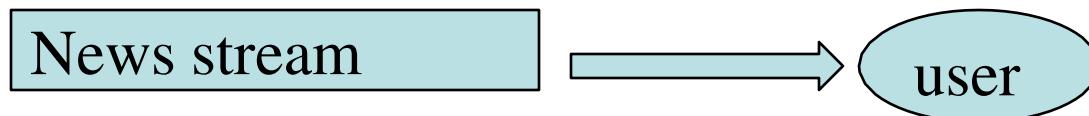
- Construct term – Incident matrix and processing Boolean query for the given document.

# CONTENTS

- Retrieval Tasks
- Term-Document incidence
- Inverted Index
- Preprocessing steps

# Retrieval Tasks

- **Ad hoc retrieval:** Fixed document corpus, varied queries.
- **Filtering:** Fixed query, continuous document stream.
  - User Profile: A model of relative static preferences.
  - Binary decision of relevant/not-relevant.



- **Routing:** Same as filtering but continuously supply ranked lists rather than binary filtering.

# Retrieval Models

- A retrieval model specifies the details of:
  - 1) Document representation
  - 2) Query representation
  - 3) Retrieval function: how to find relevant results
  - Determines a notion of relevance.
- Classical models
  - Boolean models (set theoretic)
    - Extended Boolean
  - Vector space models (statistical/algebraic)
    - Generalized VS
    - Latent Semantic Indexing
  - Probabilistic models

# Boolean Model

- A document is represented as a **set** of keywords.
- Queries are Boolean expressions of keywords, connected by AND, OR, and NOT, including the use of brackets to indicate scope.
  - Rio & Brazil | Hilo & Hawaii
  - hotel & !Hilton
- Output: Document is relevant or not. No partial matches or ranking.
  - Can be extended to include ranking.
- Popular retrieval model in old time:
  - Easy to understand. Clean formalism.
  - But still too complex for web users

# Query example

- Which plays of Shakespeare contain the words *Brutus AND Caesar* but *NOT Calpurnia*?
- Could **grep** all of Shakespeare's plays for *Brutus* and *Caesar*, then strip out lines containing *Calpurnia*?
  - Slow (for large corpora)
  - *NOT Calpurnia* is non-trivial
  - Other operations (e.g., find the phrase *Romans and countrymen*) not feasible

# Term-document incidence

1 if play contains word, 0 otherwise

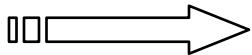
	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

- Incident vectors: 0/1 vector for each term.
- Query answer with bitwise operations (AND, negation, OR):
  - Which plays of Shakespeare contain the words *Brutus* AND *Caesar* but NOT *Calpurnia*?
  - $110100 \text{ AND } 110111 \text{ AND } 101111 = 100100$ .

# Inverted index

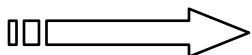
- For each term  $T$ , must store a list of all documents that contain  $T$ .

Brutus



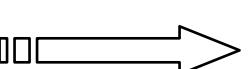
2	4	8	16	32	64	128	
---	---	---	----	----	----	-----	--

Calpurnia



1	2	3	5	8	13	21	34
---	---	---	---	---	----	----	----

Caesar

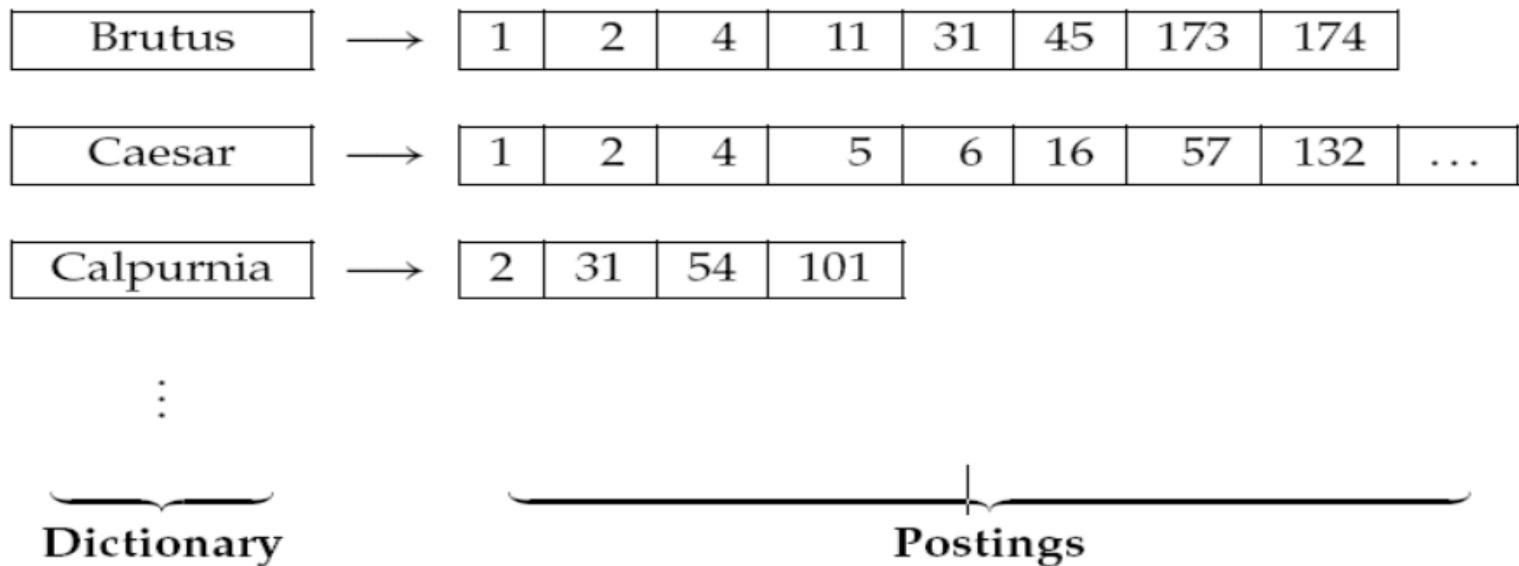


13	16						
----	----	--	--	--	--	--	--

What happens if the word Caesar is added to document 14?

# Inverted index

- **Linked lists generally preferred to arrays**
  - Dynamic space allocation
  - Insertion of terms into documents easy
  - Space overhead of pointers



# Possible Document Preprocessing Steps

- Strip unwanted characters/markup (e.g. HTML tags, punctuation, numbers, etc.).
- Tokenize the text, turning each document into a list of tokens
- Possible linguistic processing (used in some applications, but dangerous for general web search)
  - Stemming (cards ->card)
  - Remove common stopwords (e.g. a, the, it, etc.).
  - Used sometime, but dangerous
- Build inverted index
  - keyword → list of docs containing it.
  - Common phrases may be detected first using a domain specific dictionary.

# Inverted index construction

## Inverted Index Construction

Documents to be indexed.



Friends, Romans, countrymen.

Token stream.

**Tokenizer**

Friends

Romans

Countrymen

Modified tokens.

Linguistic modules

friend

roman

countryman

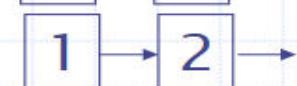
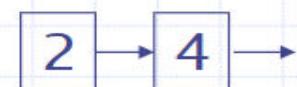
Inverted index.

**Indexer**

*friend*

*roman*

*countryman*



# Discussions

- **Index construction**
  - Stemming?
  - Which terms in a doc do we index?
    - All words or only “important” ones?
    - Stopword list: terms that are so common
      - they MAY BE ignored for indexing.
      - e.g., ***the, a, an, of, to*** ...
      - language-specific.
      - May have to be included for general web search

- **How do we process a query?**

- Stop word removal
  - Where is UCSB?
- Stemming?

<b>Dataset</b>	<b>Small</b>	<b>Big</b>
Offline	Stemming	Less or no stemming
Online	Stemming Stopword removal	Less or no stemming Stopword removal

# Summary

- Retrieval Tasks
- Term-Document incidence
- Inverted Index
- Preprocessing steps

# References

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

*Thank you !!!*



# **19CSEN2003 - Information Retrieval Techniques**

## **(UNIT 1 - Session 2)**

**UNIT 1 : Text Retrieval Systems**

**TOPIC :Term Vocabulary and Posting Lists**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

- Discuss about the extended Boolean model versus ranked retrieval.

# RECAP OF PREVIOUS SESSION

- Query Processing
- Merge
- Vector-space Model
- Cosine Similarity Measure

# CONTENTS

- Building an Inverted index
- Processing Boolean Queries
- Ranked Retrieval

# **Boolean Retrieval**

# **Term Vocabulary and Posting Lists**

# **Web Search Basics**

# What is Information Retrieval?

“Information Retrieval is finding material (**documents**) of an unstructured nature (**text**) that satisfies an information need from within large collections (**in computers**)”

- What was previously held to few professions is now widespread
  - Librarians, paralegals, etc → Everyday digital web surfers



- Information Retrieval Systems have three distinguishable scales:
  - **Web Search**
    - Billions of documents stored in millions of computers
  - **Personal Information Retrieval**
    - Most OS have integrated IR. Same with Email programs.
  - **Enterprise, Institutional, and Domain-Specific Search**
    - Internal documents, database of patents, research articles, etc...

# A First Take at Building an Inverted Index

## 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:

- ❖ Collect the documents to be indexed
- ❖ Tokenize the text, turning each document into a list of tokens
- ❖ Do linguistic preprocessing, producing a list of normalized tokens (indexing terms)
- ❖ Index the documents that each term occurs in by creating an inverted index consisting of a dictionary and postings.

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

# Processing Boolean Queries

Boolean queries are queries that use AND, OR and NOT.  
Primary commercial retrieval tool for 3 decades.  
Precise and you will get the results you want.

Consider the query: BRUTUS AND CALPURNIA

The steps will be:

1. Locate BRUTUS in the dictionary
2. Retrieve its posting list from the postings file
3. Locate CALPURNIA in the dictionary
4. Retrieve its posting list from the Postings file
5. Intersect the two posting lists.

# The Extended Boolean Model vs Ranked Retrieval

Boolean Retrieval Model will:

- Represent each document unweighted
- Represent the query unweighted
- Retrieve an unordered set of documents containing the query words.

Ranked Retrieval Model will:

- Represent each document weighted based on document frequency
- Represent the query unweighted
- Retrieve a ranking documents containing the query words.

# Summary

- Building an Inverted index
- Processing Boolean Queries
- Ranked Retrieval

# References

- Christopher D. Manning and Prabhakar Raghavan, "Introduction to Information Retrieval", Cambridge University Press, 2008.

*Thank you !!!*



# **19CSEN2003 - Information Retrieval Techniques**

## **(UNIT 1 - Session 3)**

**UNIT 1 : Text Retrieval Systems**

**TOPIC : Document Delineation**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

- Explain Document delineation and character sequence decoding & Determining the vocabulary of terms.

# RECAP OF PREVIOUS SESSION

- Building and Inverted index
- Processing Boolean Queries
- Ranked Retrieval

# CONTENTS

- Character Sequence Decoding
- Vocabulary of Terms

# Document Delineation and Character Sequence Decoding

Digital documents are typically bytes in a file or on a web server

Step 1 - Determine the correct encoding (Unicode, ASCII, MME, etc)

Step 2 - Decode the byte sequence to a linear character sequence

- Characters may have to be decoded out of some binary representation like Microsoft Word DOC files or zip files and additional decoding may need to be done on character entities

- ❖ Textual part of the document may need to be extracted out of other material that will not be processed
- ❖ **Step 3 - Determine the document unit for indexing**
  - ❖ Decoding documents with attachments - regard each as a separate document or as one single document
  - ❖ Decoding a single document, such as a PowerPoint, and split it into separate files
  - ❖ Granularity - determine whether to index each chapter, paragraph or sentence as separate files to improve relevance of search queries

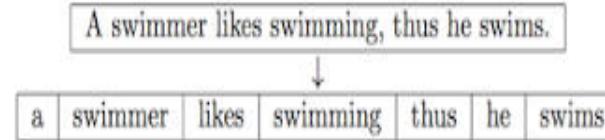
# Determining the Vocabulary of Terms

Tokenization - task of chopping a character sequence into little pieces (tokens)

- Input: Friends, Romans, Countrymen,
- Output: Friends | Romans | Countrymen

Type - class of all tokens containing same character sequence  
 Term - Type included in the system's dictionary

Table 2: Example of tokenization.



Computer technology has introduced new tokenizers that should tokenize into a single token such as email addresses and web URLs. Other new updates include

- Hyphenation (i.e. co-education, Hewlett-Packard)
- Compound words in foreign languages (i.e. Lebensversicherungsgesellschaftsangestellter)

# Summary

- Character Sequence Decoding
- Vocabulary of Terms

# References

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

*Thank you !!!*



# **19CSEN2003 - Information Retrieval Techniques**

## **(UNIT 1 - Session 4)**

**UNIT 1 : Text Retrieval Systems**

**TOPIC : Faster Postings List Intersection via Skip Pointers**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

- Examine Faster postings list intersection via skip pointers.

# RECAP OF PREVIOUS SESSION

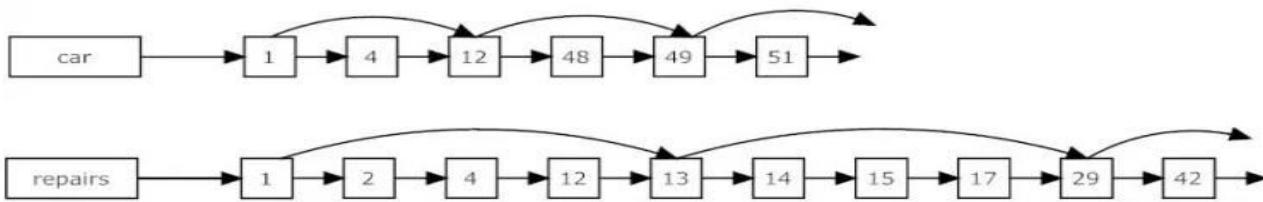
- Character Sequence Decoding
- Vocabulary of Terms

# CONTENTS

- Positional Postings and Phrase Queries
- Web Search Basics
- The Search User Experience

# Faster Postings List Intersection via Skip Pointers

- If we have a size X posting lists and size Y postings lists and the intersection takes  $X+Y$  operations. By using the skip pointers can make the intersection in less than  $X+Y$  operations.
- For example, First we start normal intersection and matched 12 and in car list, next is 48, in repairs list the next one is 13 and we have a skip pointer on 13 then we move to next skip pointer which is 29 and it's still smaller than 48.
- Thus we skip 14, 15, 17 those 3 items which means we shorter the intersection operations.
- But if the next skip pointer is 49 instead of 29 then we have to back to 13 and do the normal intersection operation. For now the operations may larger than  $X+Y$  since we have one step to check the skip pointer.



# Positional Postings and Phrase Queries

- **Phrase Queries**

Want to be able to answer queries such as “University of North Carolina at Charlotte”. Then the sentence “I went to university at Charlotte” is not match.

- **Biword indexes**

One approach to handing phrases is to consider every pair of consecutive terms in a document as a phrae. For example, the text “A B C” would generate the biword: “AB” and “BC”. When we have longer queries then we get more biword which were treated as vocabulary term.

- **Positional Postings**

Since the Biword indexes is still not a standard solution. Positional Postings put each distinct term into posting list then record their frequency and the position in the document.

For example we have “A B” and then we are going to seek the position of A, and then going to find the term B which has 1 position higher than A. Thus we can find “A B”.

# Web Search Basics

The client (browser) sends an http request to a web server - specifying the URL

- Domain - <https://mail.google.com/mail/u/0/#inbox> - that specifies the root of a hierarchy of web pages
- Path in the hierarchy -  
<https://mail.google.com/mail/u/0/#inbox> - that contains the information to be returned by the server

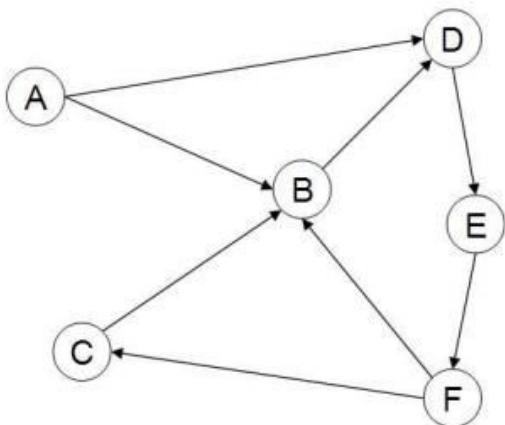
## Making web information “discoverable”

- Full text index search engines - presented the user with a keyword search interface supported by inverted indexes and ranking mechanisms (Altavista, Excite and Infoseek)
- Taxonomies - trees whose nodes are labelled with entities that are expected to occur in a web search query
  - populated with web pages in categories organized by topic or type (Yahoo, Bing, Google)

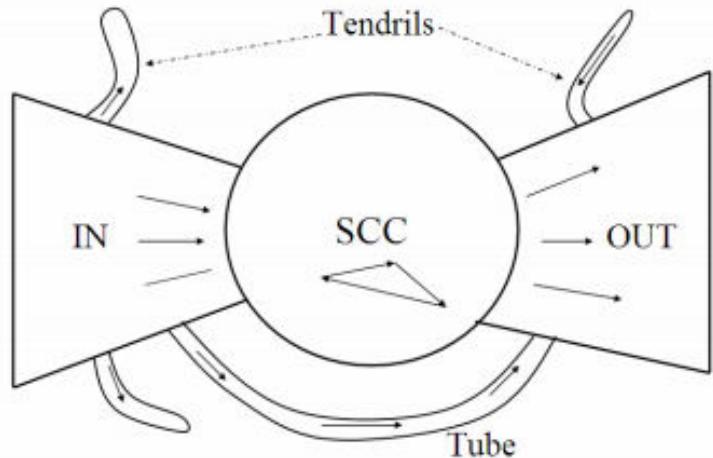
# Web Characteristics



Dynamic Web App



Static Web App Graph



“Bowtie Graph”

# Advertising as the Economic Model

Advertising pricing models:

- Cost Per Mil (CPM): The cost to the company to display banner 1000 times
- Cost per Click (CPC): Cost based on number of clicks ad receives.

Search Engine Marketing: How search engines implement algorithm ranking and how to allocate marketing campaign budgets to different keywords.

- Understand common search engine queries
- Recognize “Click Spam”
- Pure vs impure search engines



# The Search User Experience



User web queries are grouped into three categories

- 1.Informational - seeking information on a broad topic
  - Ex: Infuenza, World War 2
- 2.Navigational - seeing a website or entity that the user has in mind
  - Ex: Lufthansa airlines
- 3.Transaction - a prelude to the user performing a transaction on the Web
  - Ex: purchasing a product, making a reservation, downloading a file

# Summary

- Positional Postings and Phrase Queries
- Web Search Basics
- The Search User Experience

# References

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

*Thank you !!!*



# **19CSEN2003 - Information Retrieval Techniques**

## **(UNIT 1 - Session 5)**

**UNIT 1 : Text Retrieval Systems**

**TOPIC : Dictionaries and Tolerant retrieval**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

- Building postings data structures suitable for handling phrase and proximity queries.

# RECAP OF PREVIOUS SESSION

- Positional Postings and Phrase Queries
- Web Search Basics
- The Search User Experience

# CONTENTS

- Dictionary data structures for inverted indexes
- Tree: binary tree

- Dictionary data structures
- “Tolerant” retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex

# Dictionary data structures for inverted indexes

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

BRUTUS → 

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

CAESAR → 

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

CALPURNIA → 

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

:

  
**dictionary**

  
**postings**

# A naïve dictionary

- An array of struct:

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

char[20]    int                              Postings \*

20 bytes    4/8 bytes                      4/8 bytes

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

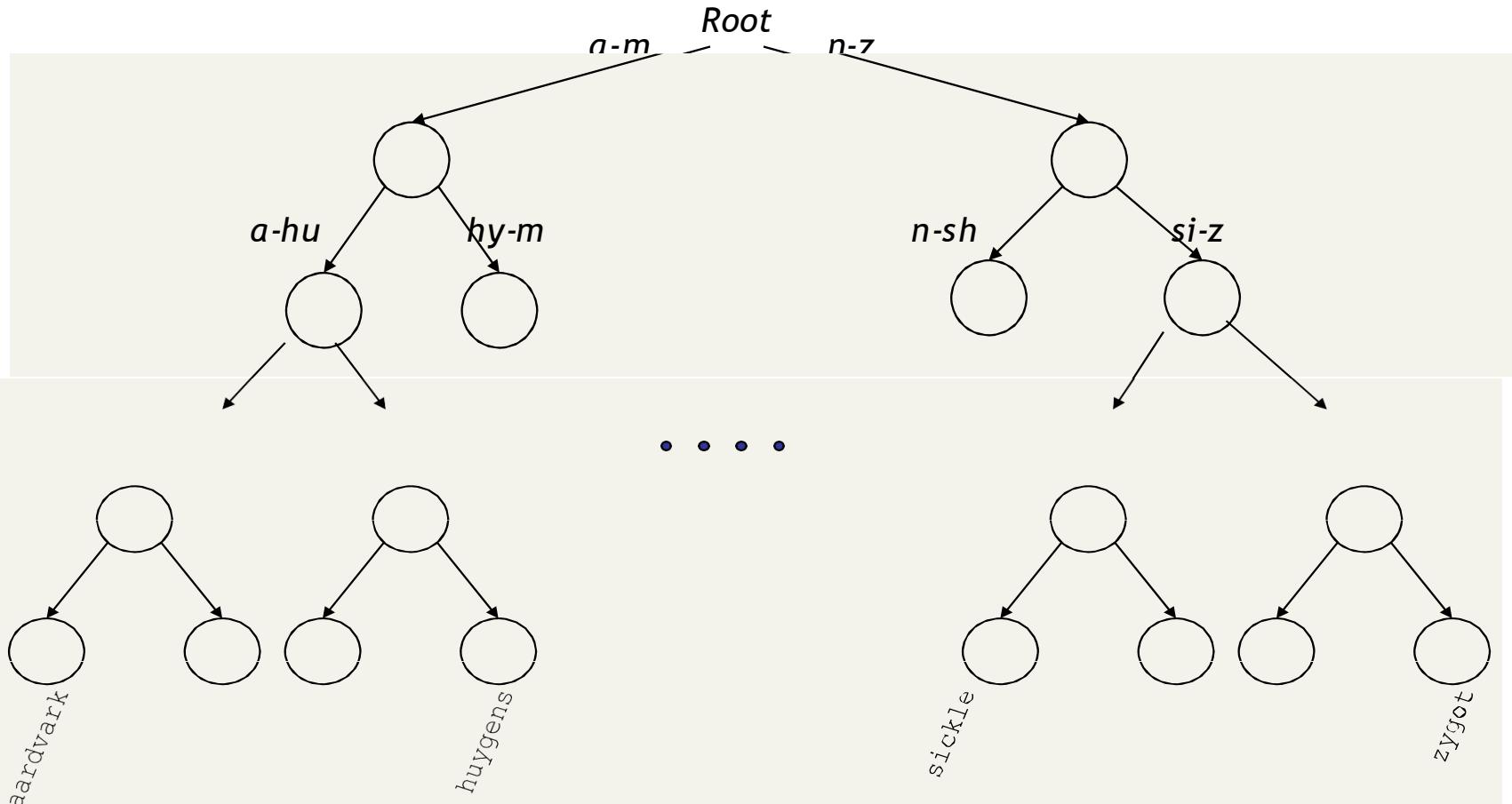
# Dictionary data structures

- Two main choices:
  - Hash table
  - Tree
- Some IR systems use hashes, some trees

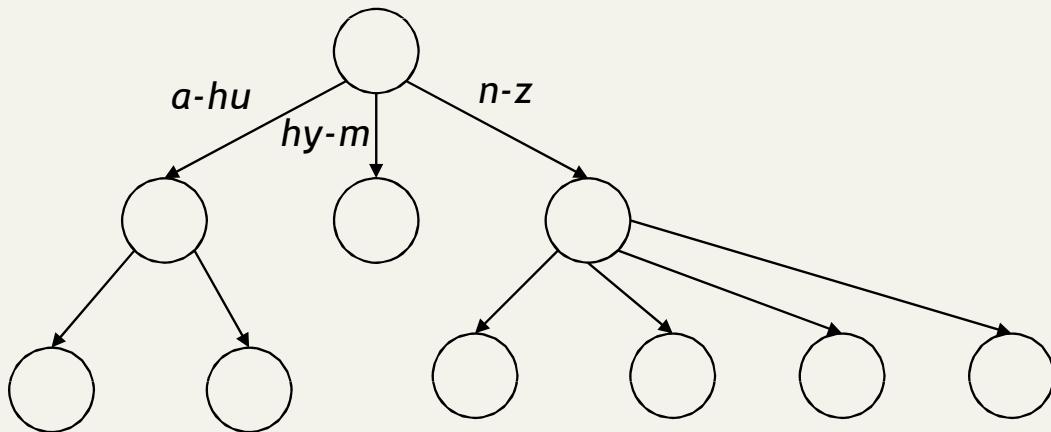
# Hashes

- Each vocabulary term is hashed to an integer
  - (We assume you've seen hashtables before)
- Pros:
  - Lookup is faster than for a tree:  $O(1)$
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps going, need to occasionally do the expensive operation of rehashing *everything*

# Tree: binary tree



# Tree: B-tree



- Definition: Every internal node has a number of children in the interval  $[a,b]$  where  $a, b$  are appropriate natural numbers, e.g.,  $[2,4]$ .

# Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings ... but we standardly have one
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower:  $O(\log M)$  [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

# Summary

- Dictionary data structures for inverted indexes
- Tree: binary tree

# References

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

*Thank you !!!*



# **19CSEN2003 - Information Retrieval Techniques**

## **(UNIT 1 - Session 6)**

**UNIT 1 : Text Retrieval Systems**

**TOPIC : Inverted index.**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

- Develop data structures that help the search for terms in the vocabulary in an inverted index.

# RECAP OF PREVIOUS SESSION

- Dictionary data structures for inverted indexes
- Tree: binary tree

# CONTENTS

- Inverted Index
- Types of inverted indexes
- Steps to Build an Inverted Index

# Inverted Index

- An **Inverted Index** is a data structure used in information retrieval systems to efficiently retrieve documents or web pages containing a specific term or set of terms.
- In an inverted index, the index is organized by terms (words), and each term points to a list of documents or web pages that contain that term.

# Inverted Index

- An inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a document or a set of documents.
- In simple words, it is a hashmap-like data structure that directs you from a word to a document or a web page.

# Example

**Example: Consider the following documents.**

Document 1: The quick brown fox jumped over the lazy dog.

Document 2: The lazy dog slept in the sun.

- To create an **inverted index** for these documents, we first tokenize the documents into terms, as follows.  
Document 1: The, quick, brown, fox, jumped, over, the, lazy, dog.  
Document 2: The, lazy, dog, slept, in, the, sun.
- Next, we create an index of the terms, where each term points to a list of documents that contain that term, as follows.

# Example

- The -> Document 1, Document 2
- Quick -> Document 1
- Brown -> Document 1
- Fox -> Document 1
- Jumped -> Document 1
- Over -> Document 1
- Lazy -> Document 1, Document 2
- Dog -> Document 1, Document 2
- Slept -> Document 2
- In -> Document 2
- Sun -> Document 2

# Types of inverted indexes

There are two types of inverted indexes:

- **Record-Level Inverted Index:** Record Level Inverted Index contains a list of references to documents for each word.
- **Word-Level Inverted Index:** Word Level Inverted Index additionally contains the positions of each word within a document. The latter form offers more functionality but needs more processing power and space to be created.

# Steps to Build an Inverted Index

- **Fetch the Document:** Removing of Stop Words: Stop words are the most occurring and useless words in documents like “I”, “the”, “we”, “is”, and “an”.
- **Stemming of Root Word:** Whenever I want to search for “cat”, I want to see a document that has information about it. But the word present in the document is called “cats” or “catty” instead of “cat”. To relate both words, I’ll chop some part of every word I read so that I could get the “root word”. There are standard tools for performing this like “Porter’s Stemmer”.
- **Record Document IDs:** If the word is already present add a reference of the document to index else creates a new entry. Add additional information like the frequency of the word, location of the word, etc.

## Example:

- Words                  Document
- ant                      doc1
- demo                    doc2
- world                   doc1, doc2

# Summary

- Inverted Index
- Types of inverted indexes
- Steps to Build an Inverted Index

# References

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

*Thank you !!!*



# **19CSEN2003 - Information Retrieval Techniques**

## **(UNIT 1 - Session 7)**

**UNIT 1 : Text Retrieval Systems**

**TOPIC : Wildcard Queries**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

- Outline on Wildcard queries.

# RECAP OF PREVIOUS SESSION

- Inverted Index
- Types of inverted indexes
- Steps to Build an Inverted Index

# CONTENTS

- Wild-card queries

# Wild-card queries

# Wild-card queries \*

- ***mon\****: find all docs containing any word beginning “mon”.
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon ≤ w < moo***
- ***\*mon***: find words ending in “mon”: harder
  - Maintain an additional B-tree for terms *backwards*.  
Can retrieve all words in range: ***nom ≤ w < non***.

*Exercise: from this, how can we enumerate all terms meeting the wild-card query ***pro\*cent*** ?*

# Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query:

**se\*ate AND fil\*er**

This may result in the execution of many Boolean AND queries.

# B-trees handle \*'s at the end of a query term

- How can we handle \*'s in the middle of query term?
  - ***co\*tion***
- We could look up ***co\**** AND ***\*tion*** in a B-tree and intersect the two term sets
  - Expensive
- The solution: transform wild-card queries so that the \*'s occur at the end
- This gives rise to the **Permuterm** Index.

# Permuterm index

- For term ***hello***, index under:
  - ***hello\$*, *ello\$h*, *llo\$he*, *lo\$hel*, *o\$hell***  
**where \$ is a special symbol.**
- 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\**

# Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before.
- *Permuterm problem:  $\approx$  quadruples lexicon size*

*Empirical observation for English.*

# Bigram ( $k$ -gram) indexes

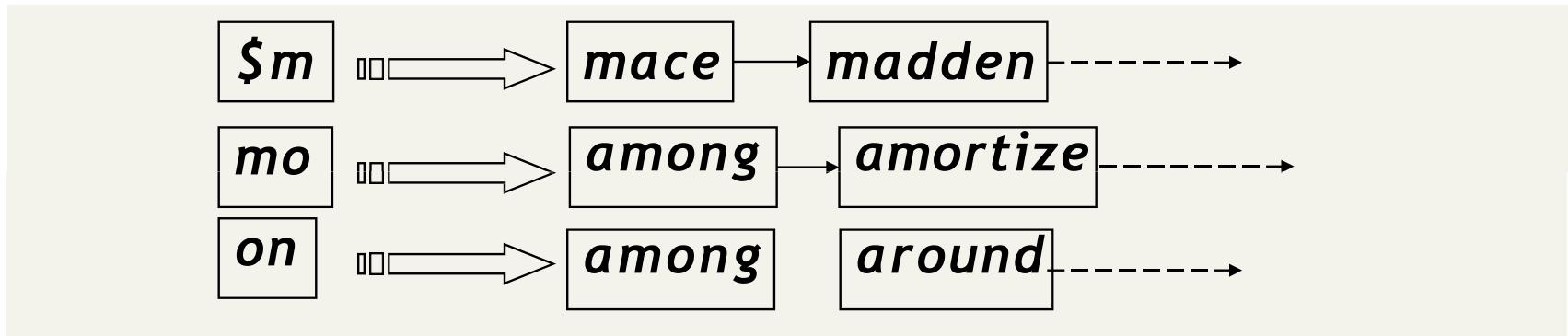
- Enumerate all  $k$ -grams (sequence of  $k$  chars) occurring in any term
- e.g., from text “***April is the cruelest month***” we get the 2-grams (*bigrams*)

\$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
- Maintain a second inverted index from bigrams to dictionary terms that match each bigram.

# Bigram index example

- The  $k$ -gram index finds *terms* based on a query consisting of  $k$ -grams



# Processing *n*-gram 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).

# Processing wild-card queries

- 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.

- Does Google allow wildcard queries?

# Summary

- Wild-card queries
- Processing n-gram wild-cards

# References

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

*Thank you !!!*



# **19CSEN2003 - Information Retrieval Techniques**

## **(UNIT 1 - Session 8)**

**UNIT 1 : Text Retrieval Systems**

**TOPIC : Spelling correction**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

- Discuss techniques for correcting spelling errors in queries.

# RECAP OF PREVIOUS SESSION

- Dictionary data structures for inverted indexes
- Tree: binary tree
- Wild-card queries

# CONTENTS

- Spell Correction
- Document Correction
- Isolated word correction

# Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve “right” answers
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew form Heathrow to Narita.***

# 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
- But often we don't change the documents but aim to fix the query-document mapping

# Query mis-spellings

- Our principal focus here
  - E.g., the query ***Alanis Morisett***
- We can either
  - Retrieve documents indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - *Did you mean ... ?*

# 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
  - The lexicon of the indexed corpus
    - . . . ,
    - All names, acronyms etc.
    - (Including the mis-spellings)

# Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- What's “closest”?
- We'll study several alternatives
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - $n$ -gram overlap

# Edit distance

- Given two strings  $S_1$ , and  $S_2$ , the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from **dof** to **dog** is 1
  - From **cat** to **act** is 2 (Just 1 with transpose.)
  - from **cat** to **dog** is 3.
- Generally found by dynamic programming.
- See <http://www.merriampark.com/ld.htm> for a nice example plus an applet.

# Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors, e.g.  $m$  more likely to be mis-typed as  $n$  than as  $q$
  - Therefore, replacing  $m$  by  $n$  is a smaller edit distance than by  $q$
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

# Using edit distances

- Given query, first enumerate all character sequences within a preset (weighted) edit distance (e.g., 2)
- Intersect this set with list of “correct” words
- Show terms you found 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
- The alternatives disempower the user, but save a round of interaction with the user

# Edit distance to all dictionary terms?

- Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
  - Alternative?
- How do we cut the set of candidate dictionary terms?
- One possibility is to use  $n$ -gram overlap for this
- This can also be used by itself for spelling correction.

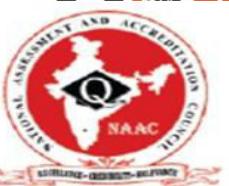
# Summary

- Spell Correction
- Document Correction
- Isolated word correction

# References

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

*Thank you !!!*



# **19CSEN2003 - Information Retrieval Techniques**

## **(UNIT 1 - Session 9)**

**UNIT 1 : Text Retrieval Systems**

**TOPIC : Soundex**

**M.GOMATHI**  
**Assistant Professor/CSE**

# OUTCOMES

## Course Outcome:

- Implement the text retrieval systems based on Boolean retrieval model by determining the vocabulary of terms

## Session Outcome:

- Outline on method for seeking vocabulary terms that are phonetically close to the query term[s].

# RECAP OF PREVIOUS SESSION

- n-gram overlap
- Jaccard coefficient

# CONTENTS

- Soundex algorithm

# Soundex

- 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 – typical algorithm

- Turn every token to be indexed into a 4-character reduced form
- Do the same with query terms
- Build and search an index on the reduced forms
  - (when the query calls for a soundex match)

# Soundex – typical algorithm

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):  
'A', 'E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:
  - B, F, P, V → 1
  - C, G, J, K, Q, S, X, Z → 2
  - D, T → 3
  - L → 4
  - M, N → 5
  - R → 6

# Soundex continued

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
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>.

E.g., **Herman** becomes H655.

*Will hermann generate the same code?*

# Soundex

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, ...)
- How useful is soundex?
- Not very – for information retrieval
- Okay for “high recall” tasks (e.g., Interpol), though biased to names of certain nationalities
- Zobel and Dart (1996) show that other algorithms for phonetic matching perform much better in the context of IR

# What queries can we process?

- We have
  - Positional inverted index with skip pointers
  - Wild-card index
  - Spell-correction
  - Soundex
- Queries such as

**(SPELL(*moriset*) /3 *toron*\**to*) OR  
SOUNDEX(*chaikofski*)**

# Exercise

- Draw yourself a diagram showing the various indexes in a search engine incorporating all the functionality we have talked about
- Identify some of the key design choices in the index pipeline:
  - Does stemming happen before the Soundex index?
  - What about  $n$ -grams?
- Given a query, how would you parse and dispatch sub-queries to the various indexes?

# Resources

- Efficient spell retrieval:
  - K. Kukich. Techniques for automatically correcting words in text. ACM Computing Surveys 24(4), Dec 1992.
  - J. Zobel and P. Dart. Finding approximate matches in large lexicons. Software - practice and experience 25(3), March 1995. <http://citeseer.ist.psu.edu/zobel95finding.html>
  - Mikael Tillenius: Efficient Generation and Ranking of Spelling Error Corrections. Master's thesis at Sweden's Royal Institute of Technology. <http://citeseer.ist.psu.edu/179155.html>
- **Nice, easy reading on spell correction:**
  - Peter Norvig: How to write a spelling corrector  
<http://norvig.com/spell-correct.html>

# Summary

- Soundex algorithm

# References

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

*Thank you !!!*



NM 100

1923 - 2014

Ma 100

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT II - Session 1)**

**UNIT II : Index construction and Compression**

**TOPIC : Hardware basics**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Deploy an algorithm for indexing using suitable index construction and compression methods for a given database

## **Session Outcome :**

- Discuss about basics of computer hardware relevant for indexing.

# CURRENT SESSION

## Hardware basics

- Design decisions based on the characteristics of hardware
- Reviewing hardware basics

# HARDWARE BASICS

- Access to data in memory is much faster than access 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

- 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.

## RCV1: Our collection for this lecture

- Shakespeare's collected works definitely aren't large enough for demonstrating many of the points in this course.
- The collection we'll use isn't really large enough either, but it's publicly available and is at least a more plausible example.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
- This is one year of Reuters newswire (part of 1995 and 1996)

# A Reuters RCV1 document



You are here: Home > News > Science > Article

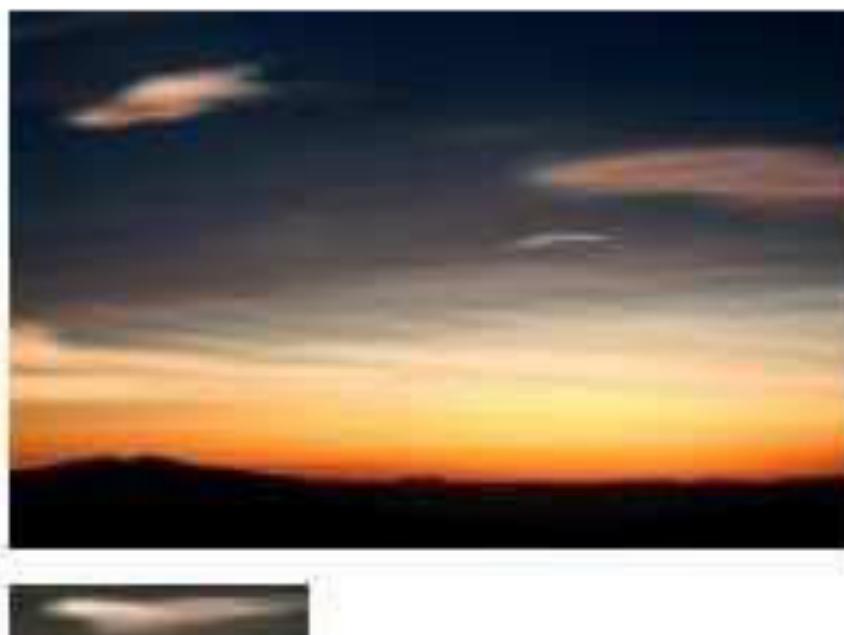
Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enough

## Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

[Email This Article](#) | [Print This Article](#) | [Reprints](#)

[+] Text [-]



SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

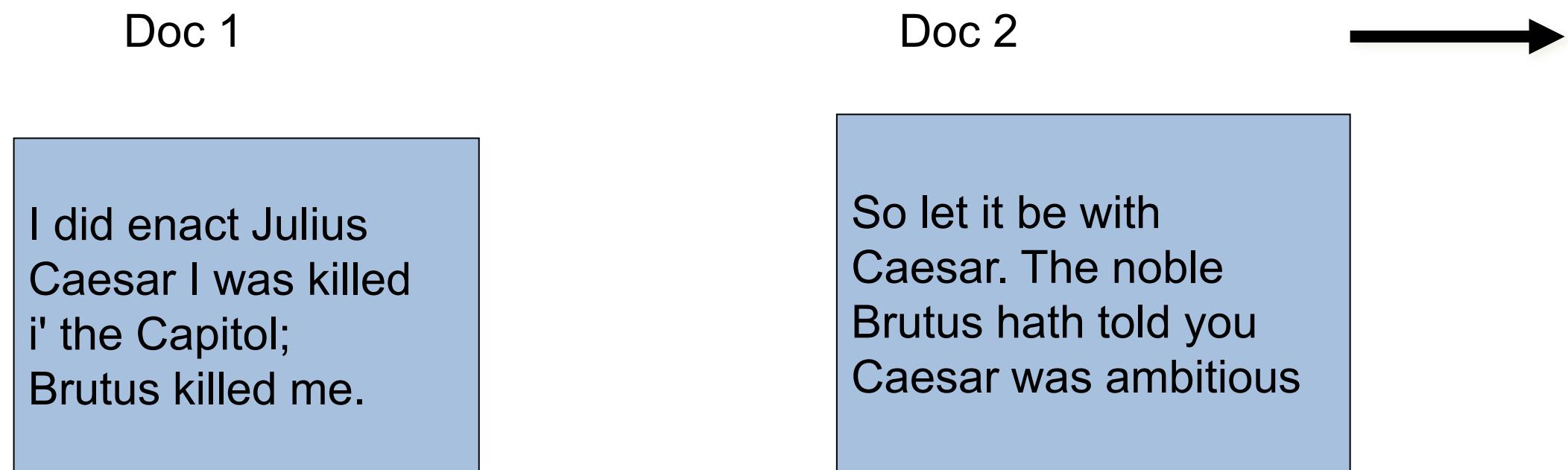
Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

# Reuters RCV1 statistics

- | symbol   | statistic   | value       |
|--|---|-------------|
| N  | documents   | 800,000     |
| L  | avg. # tokens per doc                             | 200         |
| M  | terms (= word types)                              | 400,000     |
|  | avg. # bytes per token<br>(incl. spaces/punct.)   | 6           |
|  | avg. # bytes per token<br>(without spaces/punct.) | 4.5         |
|  | avg. # bytes per term                             | 7.5         |
|  | non-positional postings                           | 100,000,000 |
| 4.5 bytes per word token vs. 7.5 bytes per word type: why? |   |             |

# Recall IIR 1 index construction

- Documents are parsed to extract words and these are saved with the Document ID.



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

# Key step

- After all documents have been parsed, the inverted file is sorted by terms.

We focus on this sort step.  
We have 100M items to sort.

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

# Scaling index construction

- In-memory index construction does not scale
  - Can't stuff entire collection into memory, sort, then write back
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about .
  - ..
- Memory, disk, speed, etc.

# Sort-based index construction

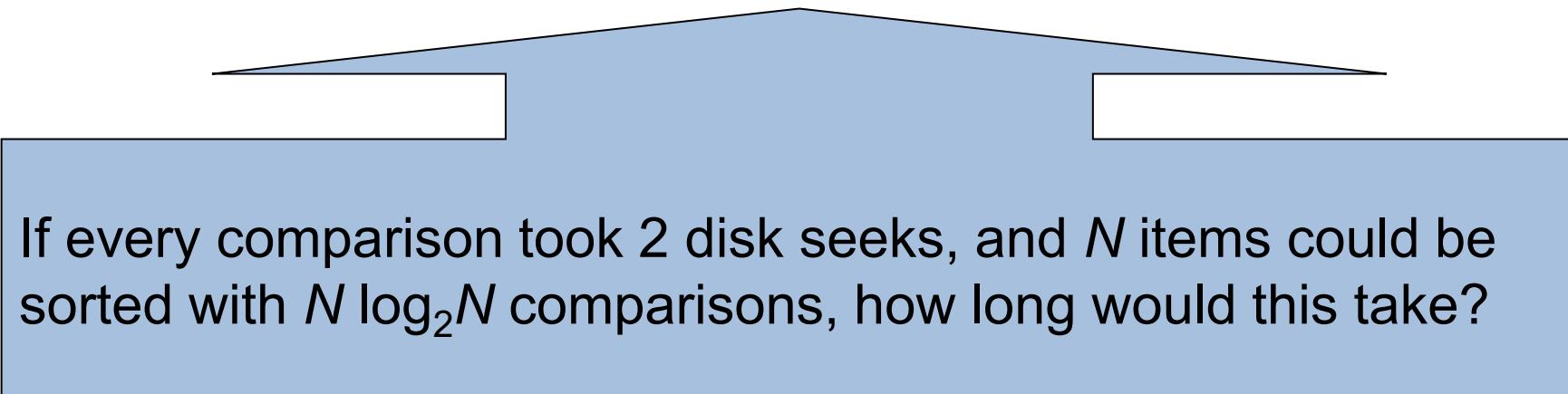
- As we build the index, we parse docs one at a time.
  - While building the index, we cannot easily exploit compression tricks (**you can, but much more complex**)
- The final postings for any term are incomplete until the end.
- At 12 bytes per non-positional postings entry (*term, doc, freq*), demands a lot of space for large collections.
- $T = 100,000,000$  in the case of RCV1
  - So ... we can do this in memory in 2009, but typical collections are much larger. E.g., the *New York Times* provides an index of >150 years of newswire
- Thus: We need to store intermediate results on disk.

# Sort using disk as “memory”?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: Sorting  $T = 100,000,000$  records on disk is too slow – too many disk seeks.
- We need an *external* sorting algorithm.

# BOTTLENECK

- Parse and build postings entries one doc at a time
- Now sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow – must sort  $T=100M$  records



# SUMMARY

## Hardware basics

- Design decisions based on the characteristics of hardware
- Reviewing hardware basics
- Scaling index construction
- Sort-based index construction

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

# **THANK YOU !!!**



NM 100

1923 - 2014

Memorial

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT II - Session 2)**

**UNIT II : Index construction and Compression**

**TOPIC : Blocked sort-based indexing**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Deploy an algorithm for indexing using suitable index construction and compression methods for a given database

## **Session Outcome :**

- Implement blocked sort-based indexing algorithm.

# RECAP OF PREVIOUS SESSION

- Hardware basics
- Scaling index construction
- Sort-based index construction

# CURRENT SESSION

- Blocked sort-based Indexing
  - Sorting with fewer disk seeks
  - BSBI Index Construction
  - How to merge the sorted runs?

# Blocked sort-based Indexing

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

**12-byte (4+4+4) records (term, doc, freq).**

**These are generated as we parse docs.**

**Must now sort 100M such 12-byte records by term.**

**Define a Block ~ 10M such records**

- Can easily fit a couple into memory.
- Will have 10 such blocks to start with.

**Basic idea of algorithm:**

- Accumulate postings for each block, sort, write to disk.
- Then merge the blocks into one long sorted order.

# Blocked sort-based Indexing

The basic steps in constructing a non-positional index are:

- 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.

## Blocked sort-based Indexing

To make index construction more efficient, we represent terms as termIDs instead of strings termID, where each termID is a unique serial number.

We can build the mapping from terms to termIDs on the fly while we are processing the collection; or, in a two-pass approach,

- we compile the vocabulary in the first pass and
- construct the inverted index in the second pass.

# BSBI

- segments the collection into parts of equal size,
- sorts the  $\langle \text{termID}, \text{docID} \rangle$  pairs of each part in memory,
- stores intermediate sorted results on disk, and merges all intermediate results into the final index.

## postings to be merged

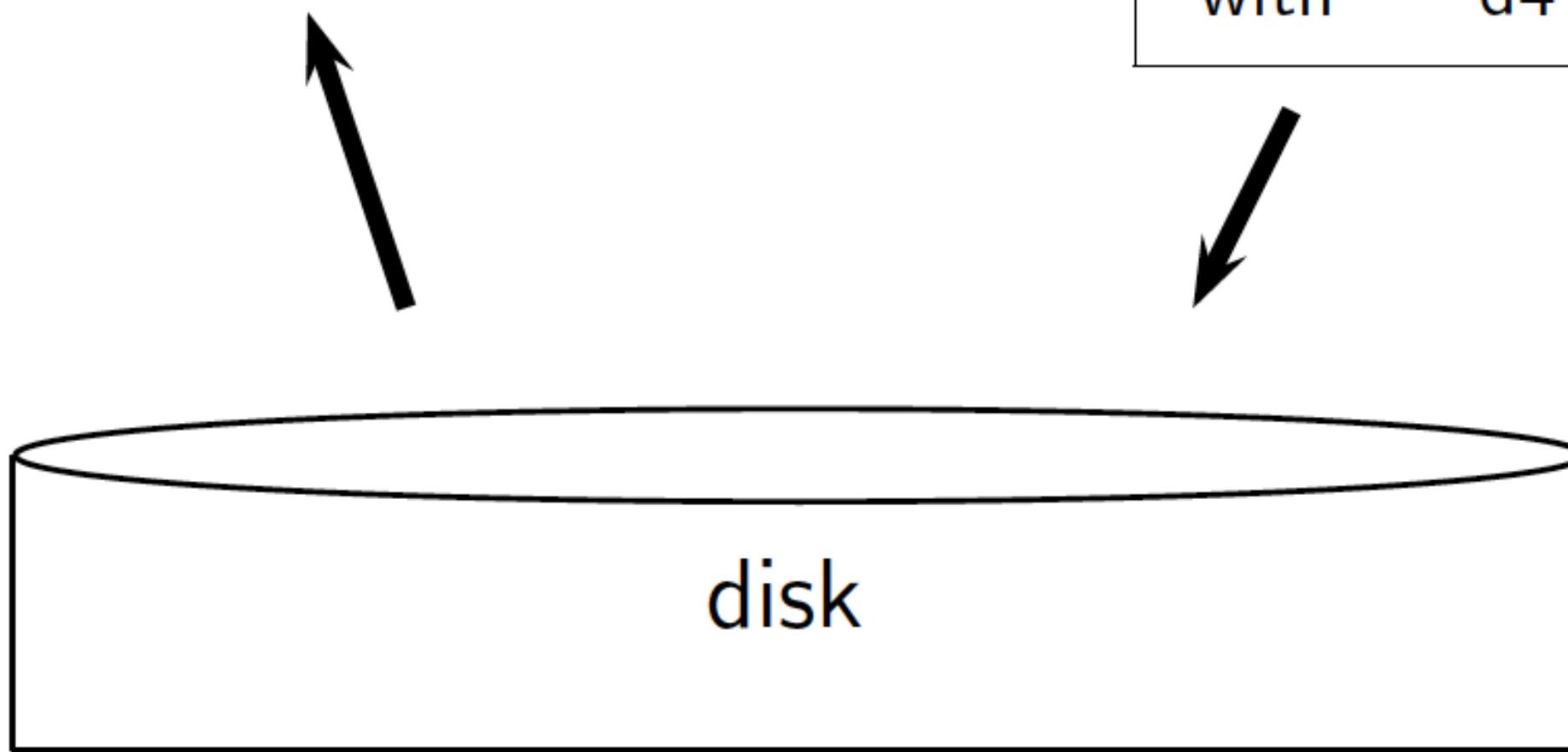
brutus	d3
caesar	d4
noble	d3
with	d4

brutus	d2
caesar	d1
julius	d1
killed	d2



brutus	d2
brutus	d3
caesar	d1
caesar	d4
julius	d1
killed	d2
noble	d3
with	d4

merged  
postings



## Sorting 10 blocks of 10M records

First, read each block and sort within:

- Quicksort takes  $2N \ln N$  expected steps
- In our case  $2 \times (10M \ln 10M)$  steps

Exercise: estimate total time to read each block from disk and quicksort it.

10 times this estimate – gives us 10 sorted runs of 10M records each.

Done straightforwardly, need 2 copies of data on disk

- But can optimize this

## BSBINDEXCONSTRUCTION()

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

## How to merge the sorted runs?

- But it is more efficient to do a multi-way merge, where you are reading from all blocks simultaneously
- Providing you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you're not killed by disk seeks

## Remaining problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with term,docID postings instead of termID,docID postings . . .
- Intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)

# SUMMARY

- Blocked sort-based Indexing
  - Sorting with fewer disk seeks
  - BSBI Index Construction
  - How to merge the sorted runs?

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

# **THANK YOU !!!**



NM 100

1923 - 2014

Memorial

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT II - Session 3)**

**UNIT II : Index construction and Compression**

**TOPIC : Single pass in-memory indexing**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Deploy an algorithm for indexing using suitable index construction and compression methods for a given database

## **Session Outcome :**

- Implement Single pass in-memory indexing algorithm.

# RECAP OF PREVIOUS SESSION

## Blocked sort-based Indexing

- Sorting with fewer disk seeks
- BSBI Index Construction
- How to merge the sorted runs?

# CURRENT SESSION

- Single-pass in-memory indexing
- Inversion of a block in single-pass in-memory indexing
- Advantages

# SPIMI: 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.

# Single-pass in-memory indexing

- BSBI has excellent scaling properties, but it needs a data structure for mapping terms to termIDs.
- For very large collections, this data structure does not fit into memory.
- A more scalable alternative is single-pass inmemory indexing (SPIMI),
  - SPIMI uses terms instead of termIDs,
  - writes each block's dictionary to disk, and then starts a new dictionary for the next block.
  - SPIMI can index collections of any size as long as there is enough disk space available.

# Single-pass in-memory indexing

- SPIMI-INVERT is called repeatedly on the token stream until the entire collection has been processed.
- Tokens are processed one by one (line 4) during each successive call of SPIMI-INVERT.
- When a term occurs for the first time, it is added to the dictionary, and a new postings list is created (line 6).
- The call in line 7 returns this postings list for subsequent occurrences of the term.

# Inversion of a block in single-pass in-memory indexing

SPIMI-INVERT(*token\_stream*)

```
1  output_file = NEWFILE()
2  dictionary = NEWHASH()
3  while (free memory available)
4  do token  $\leftarrow$  next(token_stream)
5    if term(token)  $\notin$  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  $\leftarrow$  SORTTERMS(dictionary)
12       WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13   return output_file
```

## Inversion of a block in single-pass in-memory indexing

- A difference between BSBI and SPIMI is that SPIMI adds a posting directly to its postings list(line 10).
- Instead of first collecting all <termID, docID> pairs are then sorting them (as we did in BSBI), each postings list is dynamic (i.e., its size is adjusted as it grows) and it is immediately available to collect postings.

## Inversion of a block in single-pass in-memory indexing

- We allocate space for a short postings list initially and double the space each time it is full (lines 8-9).
- When memory has been exhausted, we write the index of the block to disk (line 12).
- We have to sort the terms (line 11) before doing this because we want to write postings lists in lexicographic order to facilitate the final merging step.

## Inversion of a block in single-pass in-memory indexing

- Each call of SPIMI-INVERT writes a block to disk, just as in BSBI. The last step of SPIMI is then to merge the blocks into the final inverted index.
- In addition to constructing a new dictionary structure for each block and eliminating the expensive sorting step, SPIMI has a third important component: compression.
- Both the postings and the dictionary terms can be stored compactly on disk if we employ compression.

# ADVANTAGES

This has two advantages:

- It is faster (no sorting required), and
- it saves memory (we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored.

# SPIMI: Compression

Compression makes SPIMI even more efficient.

- Compression of terms
- Compression of postings

# SUMMARY

- Single-pass in-memory indexing
- Inversion of a block in single-pass in-memory indexing
- Advantages
  -

## REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

**THANK YOU !!!**



NM 100

1923 - 2014

Memorial

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT II - Session 4)**

**UNIT II : Index construction and Compression**

**TOPIC : Distributed indexing**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Deploy an algorithm for indexing using suitable index construction and compression methods for a given database

## **Session Outcome :**

- Implement Distributed indexing algorithm.

# RECAP OF PREVIOUS SESSION

- Single-pass in-memory indexing
- Inversion of a block in single-pass in-memory indexing

# CURRENT SESSION

- Distributed indexing
  - Parallel tasks
  - Parsers
  - Inverters
  - MapReduce

# Distributed indexing

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

# Web search engine data centers

- Web search data centers (Google, Bing, Baidu) mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: Google ~1 million servers, 3 million processors/cores (Gartner 2007)

# MASSIVE DATA CENTERS

- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system?

Answer: 63%

- Exercise: Calculate the number of servers failing per minute for an installation of 1 million servers.

# DISTRIBUTED INDEXING

- Maintain a master machine directing the indexing job – considered “safe”.
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.

# PARALLEL TASKS

- We will use two sets of parallel tasks
  - Parsers
  - Inverters
- Break the input document collection into splits
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)

# PARSERS

- 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
- Each partition is for a range of terms' first letters
  - (e.g., **a-f**, **g-p**, **q-z**) – here  $j = 3$ .
- Now to complete the index inversion

# INVERTERS

- An inverter collects all (term,doc) pairs (= postings) for one term-partition.
- Sorts and writes to postings lists

# MAPREDUCE

- Index construction was just one phase.
- Another phase: transforming a term-partitioned index into a document-partitioned index.
  - Term-partitioned: one machine handles a subrange of terms
  - Document-partitioned: one machine handles a subrange of documents
- As we'll discuss in the web part of the course, most search engines use a document-partitioned index ... better load balancing, etc.

# MAPREDUCE

- The index construction algorithm we just described is an instance of MapReduce.
- MapReduce (Dean and Ghemawat 2004) is a robust and conceptually simple framework for distributed computing ...
- Without having to write code for the distribution part.
- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.

# Schema for index construction in MapReduce

- **Schema of map and reduce functions**
- map: input → list(k, v)    reduce: (k,list(v)) → output
- **Instantiation of the schema for index construction**
- map: collection → list(termID, docID)
- reduce: (<termID1, list(docID)>, <termID2, list(docID)>, ...) → (postings list1, postings list2, ...)

# EXAMPLE FOR INDEX CONSTRUCTION

- Map:
- $d_1 : C \text{ came}, C \text{ c'ed}.$
- $d_2 : C \text{ died.} \rightarrow$
- $\langle C, d_1 \rangle, \langle \text{came}, d_1 \rangle, \langle C, d_1 \rangle, \langle \text{c'ed}, d_1 \rangle, \langle C, d_2 \rangle, \langle \text{died}, d_2 \rangle$
- Reduce:
- $(\langle C, (d_1, d_2, d_1) \rangle, \langle \text{died}, (d_2) \rangle, \langle \text{came}, (d_1) \rangle, \langle \text{c'ed}, (d_1) \rangle) \rightarrow$   
 $(\langle C, (d_1:2, d_2:1) \rangle, \langle \text{died}, (d_2:1) \rangle, \langle \text{came}, (d_1:1) \rangle, \langle \text{c'ed}, (d_1:1) \rangle)$

# SUMMARY

- Distributed indexing
  - Parallel tasks
  - Parsers
  - Inverters
  - MapReduce

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

# **THANK YOU !!!**



NM 100

1923 - 2014

Memorial

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT II - Session 5)**

**UNIT II : Index construction and Compression**

**TOPIC : Dynamic indexing**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Deploy an algorithm for indexing using suitable index construction and compression methods for a given database

## **Session Outcome :**

- Implement dynamic indexing algorithm.

# RECAP OF PREVIOUS SESSION

- Distributed indexing
  - Parallel tasks
  - Parsers
  - Inverters
  - MapReduce

# CURRENT SESSION

- Dynamic indexing
  - Issues with main and auxiliary indexes
  - Logarithmic merge
  - Dynamic indexing at search engines

# DYNAMIC INDEXING

- Up to now, we have assumed that collections are static.
- They rarely are:
  - Documents come in over time and need to be inserted.
  - Documents are deleted and modified.
- This means that the dictionary and postings lists have to be modified:
  - Postings updates for terms already in dictionary
  - New terms added to dictionary

# SIMPLEST APPROACH

- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index

# ISSUES WITH MAIN AND AUXILIARY INDEXES

- Problem of frequent merges – you touch stuff a lot
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient for OS.
- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists, collect postings lists of length 1 in one file etc.)

# LOGARITHMIC MERGE

- Maintain a series of indexes, each twice as large as the previous one
  - At any time, some of these powers of 2 are instantiated
- Keep smallest ( $Z_0$ ) in memory
- Larger ones ( $I_0, I_1, \dots$ ) on disk
- If  $Z_0$  gets too big ( $> n$ ), write to disk as  $I_0$
- or merge with  $I_0$  (if  $I_0$  already exists) as  $Z_1$
- Either write merge  $Z_1$  to disk as  $I_1$  (if no  $I_1$ )
- Or merge with  $I_1$  to form  $Z_2$

# LOGARITHMIC MERGE

LMERGEADDTOKEN(*indexes*,  $Z_0$ , *token*)

```
1   $Z_0 \leftarrow \text{MERGE}(Z_0, \{\text{token}\})$ 
2  if  $|Z_0| = n$ 
3    then for  $i \leftarrow 0$  to  $\infty$ 
4      do if  $I_i \in \text{indexes}$ 
5        then  $Z_{i+1} \leftarrow \text{MERGE}(I_i, Z_i)$ 
6          ( $Z_{i+1}$  is a temporary index on disk.)
7           $\text{indexes} \leftarrow \text{indexes} - \{I_i\}$ 
8        else  $I_i \leftarrow Z_i$  ( $Z_i$  becomes the permanent index  $I_i$ .)
9           $\text{indexes} \leftarrow \text{indexes} \cup \{I_i\}$ 
10         BREAK
11      $Z_0 \leftarrow \emptyset$ 
```

LOGARITHMICMERGE()

```
1   $Z_0 \leftarrow \emptyset$  ( $Z_0$  is the in-memory index.)
2   $\text{indexes} \leftarrow \emptyset$ 
3  while true
4  do LMERGEADDTOKEN(indexes,  $Z_0$ , GETNEXTTOKEN())
```

# LOGARITHMIC MERGE

- Auxiliary and main index: index construction time is  $O(T^2)$  as each posting is touched in each merge.
- Logarithmic merge: Each posting is merged  $O(\log T)$  times, so complexity is  $O(T \log T)$
- So logarithmic merge is much more efficient for index construction
- But query processing now requires the merging of  $O(\log T)$  indexes
  - Whereas it is  $O(1)$  if you just have a main and auxiliary index

# FURTHER ISSUES WITH MULTIPLE INDEXES

- Collection-wide statistics are hard to maintain
- E.g., when we spoke of spell-correction: which of several corrected alternatives do we present to the user?
  - We said, pick the one with the most hits
- How do we maintain the top ones with multiple indexes and invalidation bit vectors?
  - One possibility: ignore everything but the main index for such ordering
- Will see more such statistics used in results ranking

# DYNAMIC INDEXING AT SEARCH ENGINES

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
    - Sarah Palin, ...
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is deleted

Get Search News Recaps!

Email:

Daily  Monthly

**Subscribe**

 [Feeds and more info](#)



**Google**  
Land

**YAHOO!**  
Land

**Microsoft**  
Land

Columns  
Land

Marketing  
Land

Searching  
Land

Ask, AOL &  
More Lands

Newsletters  
& Feeds 

Confe  
& Web

« [Local Store And Inventory Data Poised To Transform "Online Shopping"](#) | [Main](#) | [SEO Company, Fathom Online, Acquired By Geary Interactive](#) »

Mar 31, 2008 at 8:45am Eastern by Barry Schwartz

## **Google Dance Is Back? Plus Google's First Live Chat Recap & Hyperactive Yahoo Slurp**

Is the Google Dance back? Well, not really, but I [am noticing](#) Google Dance-like behavior from Google based on reading some of the feedback at a [WebmasterWorld](#) thread.

The Google Dance refers to how years ago, a change to Google's ranking algorithm often began showing up slowly across data centers as they reflected different results, a sign of coming changes. These days Google's data centers are typically always showing small changes and differences, but the differences between [this data center](#) and [this one](#) seem to be more like the extremes of the past Google Dances.

So either Google is preparing for a massive update or just messing around with our heads. As of now, these results have not yet moved over to the main Google.com results.

Search:

**netklix**

Click here for  
\$40 Free  
Advertising

**ONWARD**  
**Q**search ▾

the leading  
provider of search  
marketing jobs

**seomoz**  
PREMIUM MEMBERSHIP

# OTHER SORTS OF INDEXES

- Positional indexes
  - Same sort of sorting problem ... just larger
- Building character n-gram indexes:
  - As text is parsed, enumerate *n*-grams.
  - For each *n*-gram, need pointers to all dictionary terms containing it – the “postings”.
  - Note that the same “postings entry” will arise repeatedly in parsing the docs – need efficient hashing to keep track of this.
    - E.g., that the trigram you occurs in the term **deciduous** will be discovered on each text occurrence of **deciduous**
    - Only need to process each term once

# SUMMARY

- Dynamic indexing
  - Issues with main and auxiliary indexes
  - Logarithmic merge
  - Dynamic indexing at search engines

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

# **THANK YOU !!!**



NM 100  
1923 - 2014  
M A H A L I N G A M

# **19CSEN2003-Information Retrieval Techniques** **(UNIT II - Session 6)**

**UNIT II : Index construction and Compression**

**TOPIC : Dictionary compression**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Deploy an algorithm for indexing using suitable index construction and compression methods for a given database

## **Session Outcome :**

- Implement Dictionary compression algorithm.

# RECAP OF PREVIOUS SESSION

- Compression
- Recall Reuters

# CURRENT SESSION

- Dictionary storage
- Compressing the term list
- Dictionary search

# WHY COMPRESS THE DICTIONARY?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important

# DICTIONARY STORAGE – NAÏVE VERSION

- Array of fixed-width entries
  - ~400,000 terms; 28 bytes/term = 11.2 MB.

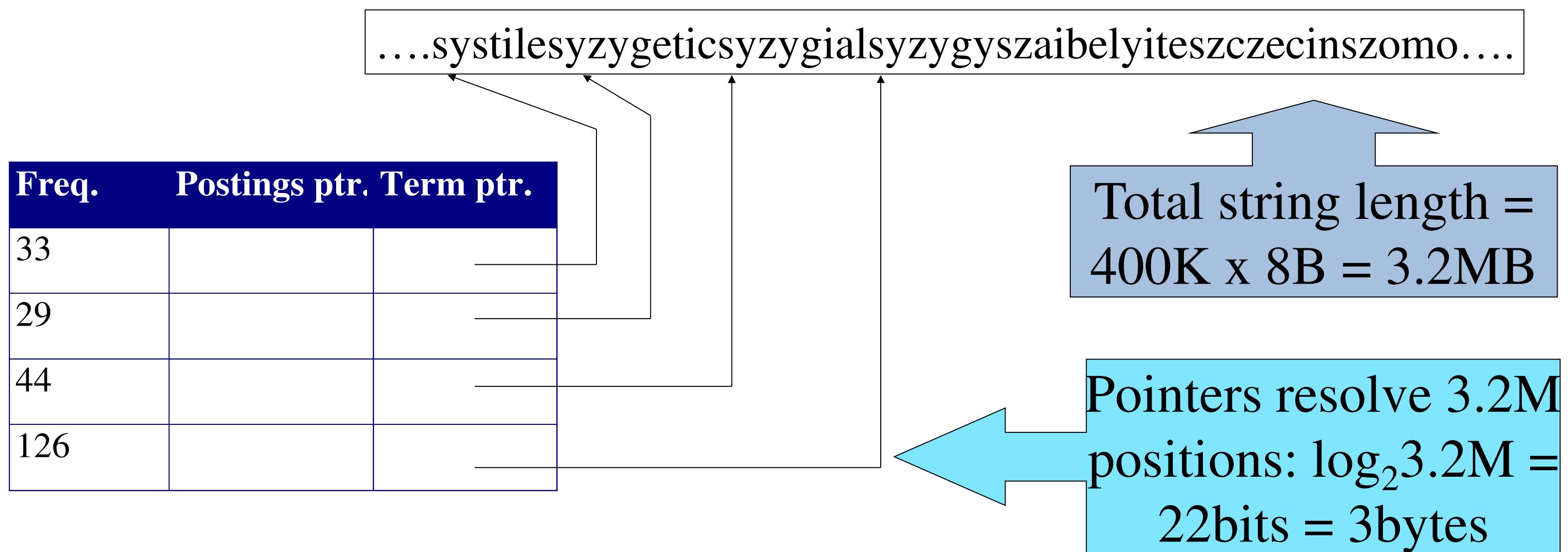
Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
.....	.....	
zulu	221	

# FIXED-WIDTH TERMS ARE WASTEFUL

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
  - And we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons*.
- Written English averages ~4.5 characters/word.
  - Exercise: Why is/isn't this the number to use for estimating the dictionary size?
- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

# Compressing the term list: Dictionary-as-a-String

- 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

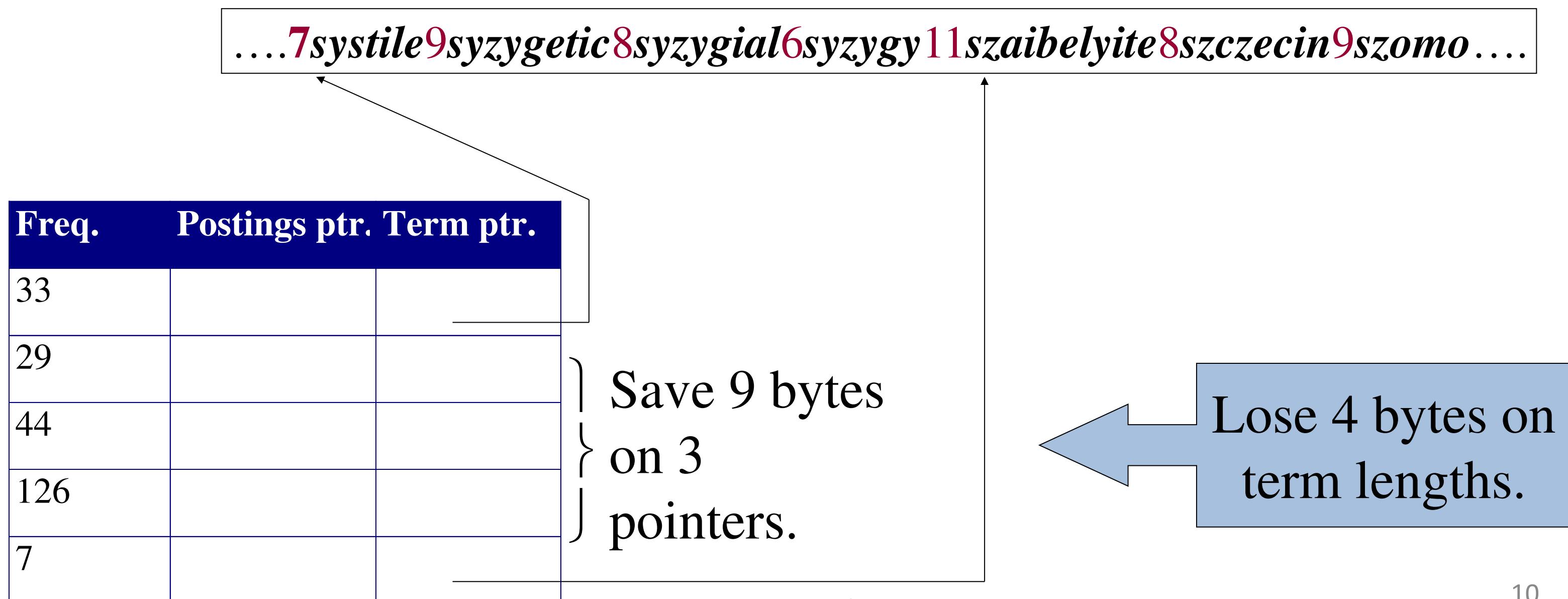


## SPACE FOR DICTIONARY AS A STRING

- 4 bytes per term for Freq.
  - 4 bytes per term for pointer to Postings.
  - 3 bytes per term pointer
  - Avg. 8 bytes per term in term string
  - 400K terms x 19  $\Rightarrow$  7.6 MB (against 11.2MB for fixed width)
- } Now avg. 11 bytes/term, not 20.

# BLOCKING

- Store pointers to every  $k$ th term string.
  - Example below:  $k=4$ .
- Need to store term lengths (1 extra byte)



## BLOCKING NET GAINS

- Example for block size  $k = 4$
  - Where we used 3 bytes/pointer without blocking
    - $3 \times 4 = 12$  bytes,
- now we use  $3 + 4 = 7$  bytes.

Question: Why not go with larger  $k$ ?

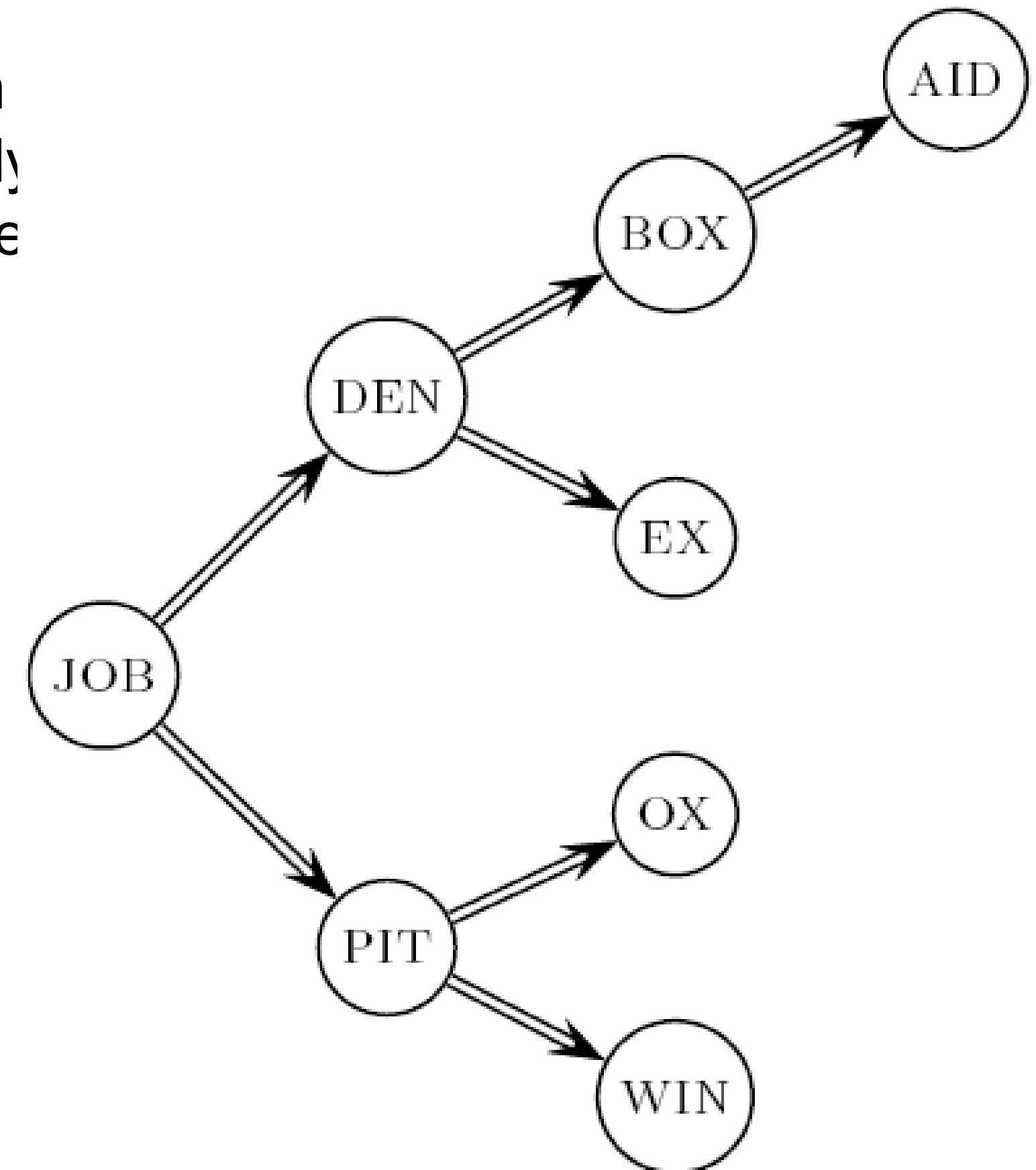
Saved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

We can save more with larger  $k$ .

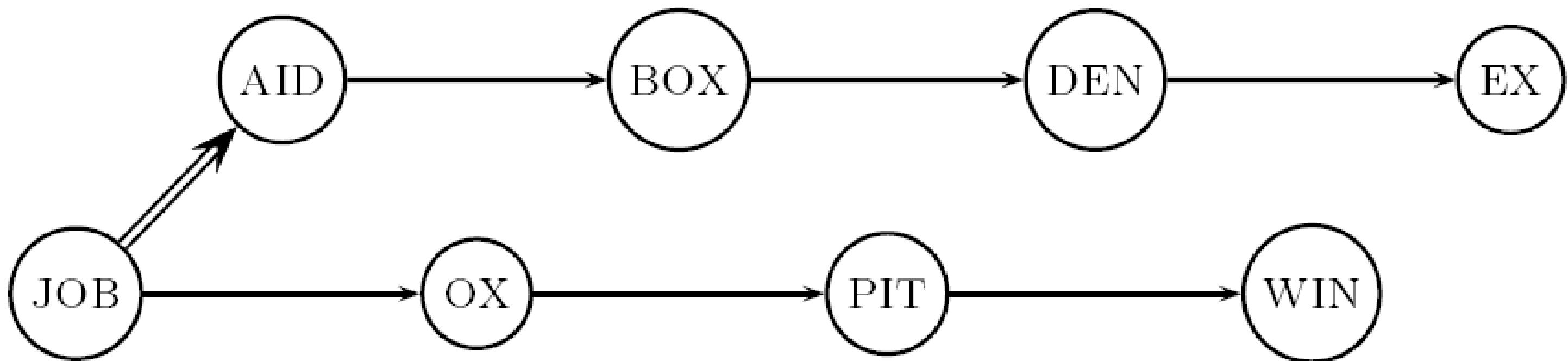
# DICTIONARY SEARCH WITHOUT BLOCKING

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons =  
$$(1+2\cdot2+4\cdot3+4)/8 \sim 2.6$$

Exercise: what if the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?



# DICTIONARY SEARCH WITH BLOCKING



Binary search down to 4-term block;  
Then linear search through terms in block.

Blocks of 4 (binary tree), avg. =  $(1+2\cdot2+2\cdot3+2\cdot4+5)/8 = 3$   
compares

# EXERCISES

- Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of  $k = 4, 8$  and  $16$ .
- Estimate the impact on search performance (and slowdown compared to  $k=1$ ) with blocking, for block sizes of  $k = 4, 8$  and  $16$ .

# FRONT CODING

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only
  - (for last  $k-1$  in a block of  $k$ )

**8automata8automate9automatic10automation**

→ **8automat\***  $a1\diamond e2\diamond ic3\diamond ion$

Encodes prefix **automat**

Extra length  
beyond **automat.**

# RCV1 dictionary compression summary

• <b>Technique</b>	<b>Size in MB</b>
• Fixed width	11.2
• Dictionary-as-String with pointers to every term	7.6
• + blocking, $k = 4$	7.1
• + blocking + front coding	5.9

# SUMMARY

- Dictionary storage
- Compressing the term list
- Dictionary search

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

# **THANK YOU !!!**



NM 100

1923 - 2014

Memorial

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT II - Session 7)**

**UNIT II : Index construction and Compression**

**TOPIC : Postings file compression**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Deploy an algorithm for indexing using suitable index construction and compression methods for a given database

## **Session Outcome :**

- Implement Postings file compression.

# RECAP OF PREVIOUS SESSION

- Dictionary storage
- Compressing the term list
- Dictionary search

# CURRENT SESSION

- Postings compression
- Gap encoding of postings file entries
- Three postings entries

# POSTINGS COMPRESSION

- The postings file is much larger than the dictionary, factor of at least 10, often over 100 times larger
- 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.

# POSTINGS: TWO CONFLICTING FORCES

- A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using  $\log_2 1M \approx 20$  bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting  $\approx 2\text{MB}$  is too expensive.
  - Prefer 0/1 bitmap vector in this case ( $\approx 100\text{K}$ )

# GAP ENCODING OF POSTINGS FILE ENTRIES

- We store the list of docs containing a term in increasing order of docID.
  - ***computer***: 33,47,154,159,202 ...
- Consequence: it suffices to store gaps.
  - 33,14,107,5,43 ...
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.
  - Especially for common words

# THREE POSTINGS ENTRIES

	encoding	postings list						
THE	docIDs	...	283042	283043	283044	283045	...	
	gaps			1	1	1		...
COMPUTER	docIDs	...	283047	283154	283159	283202	...	
	gaps			107	5	43		...
ARACHNOCENTRIC	docIDs	252000	500100					
	gaps	252000	248100					

# 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  $\sim \log_2 G$  bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a ***variable length encoding***
- Variable length codes achieve this by using short codes for small numbers

# UNARY CODE

- Represent  $n$  as  $n$  1s with a final 0.
  - Unary code for 3 is 1110.
  - Unary code for 40 is

- Unary code for 80 is:

- This doesn't look promising, but....
    - Optimal if  $P(n) = 2^{-n}$
    - We can use it as part of our solution

# GAMMA CODES

- We can compress better with bit-level codes
  - The Gamma code is the best known of these.
- Represent a gap  $G$  as a pair *length* and *offset*
- *offset* is  $G$  in binary, with the leading bit cut off
  - For example  $13 \rightarrow 1101 \rightarrow 101$
- *length* is the length of offset
  - For  $13$  (offset  $101$ ), this is  $3$ .
- We encode *length* with *unary code*:  $1110$ .
- Gamma code of  $13$  is the concatenation of *length* and *offset*:  
 $1110101$

# REMINDER: BITWISE OPERATIONS

- For compression, you need to use bitwise operators

The screenshot shows a dark-themed website header with navigation links: CS107, Schedule, Assignments, Labs, Gradebook, Resources, and Getting Help. Below the header, the main content area has a title "Computer Organization & Systems" in a large serif font. A sub-header "Week 2" is followed by a section titled "Lecture 3 (Mon 4/8): Bits and Bitwise Operators". This section includes a link to "Lecture 3 Slides" (B&O Ch 2.1) and information about inputs ("In: assign0") and outputs ("Out: assign1"). A descriptive text states: "We'll dive further into bits and bytes, and how to manipulate them using bitwise operators."

- Python (and most everything else):
  - & bitwise and; | bitwise or; ^ bitwise xor; ~ ones complement
  - << left shift bits, >> right shift; LACKS >>> zero fill right shift
  - Recipes:
    - Extract 7 bits: `a & 0x7f00 >> 8`; if take high-order bit add: `& 0x7f`
    - Combine 3 5-bit numbers: `a | (b << 5) | (c << 10)`
    - Lookup tables rather than decoding can be faster, yet still small

# GAMMA CODE PROPERTIES

- $G$  is encoded using  $2 \lfloor \log G \rfloor + 1$  bits
  - Length of offset is  $\lfloor \log G \rfloor$  bits
  - Length of length is  $\lfloor \log G \rfloor + 1$  bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible,  $\log_2 G$
- Gamma code is uniquely prefix-decodable, like VB
- Gamma code can be used for any distribution
  - Optimal for  $P(n) \approx 1/(2n^2)$
- Gamma code is parameter-free

# GAMMA SELDOM USED IN PRACTICE

- Machines have word boundaries – 8, 16, 32, 64 bits
  - Operations that cross word boundaries are slower
- Compressing and manipulating at the granularity of bits can be too slow
- All modern practice is to use byte or word aligned codes
  - Variable byte encoding is a faster, conceptually simpler compression scheme, with decent compression

# VARIABLE BYTE (VB) CODES

- For a gap value  $G$ , we want to use close to the fewest bytes needed to hold  $\log_2 G$  bits
- Begin with one byte to store  $G$  and dedicate 1 bit in it to be a continuation bit  $c$
- If  $G \leq 127$ , binary-encode it in the 7 available bits and set  $c = 1$
- Else encode  $G$ 's lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ( $c = 1$ ) – and for the other bytes  $c = 0$ .

# OTHER VARIABLE UNIT CODES

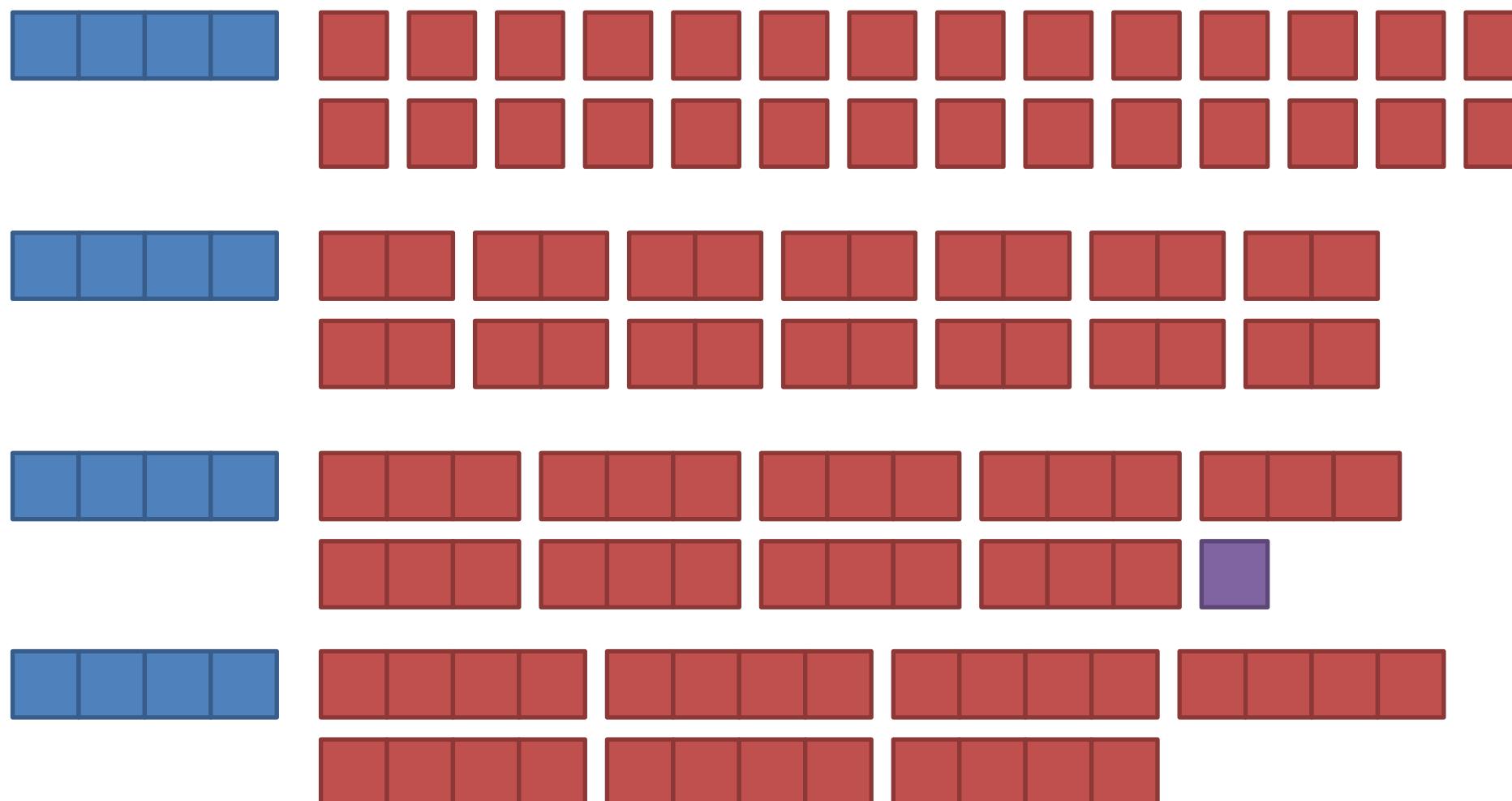
- Variable byte codes are used by many real systems
  - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches
- Byte alignment wastes space if you have many small gaps – as gap encoding often makes
- More modern work mainly uses the ideas:
  - Be word aligned (32 or 64 bits; even faster)
  - Encode several gaps at the same time
  - Often assume a maximum gap size, perhaps with an escape

# GROUP VARIABLE INTEGER CODE

- Used by Google around turn of millennium....
  - Jeff Dean, keynote at WSDM 2009 and presentations at CS276
- Encodes 4 integers in blocks of size 5–17 bytes
- First byte: four 2-bit binary length fields
  - $L_1 \mid L_2 \mid L_3 \mid L_4$  ,  $L_j \in \{1,2,3,4\}$
- Then,  $L_1+L_2+L_3+L_4$  bytes (between 4–16) hold 4 numbers
  - Each number can use 8/16/24/32 bits. Max gap length ~4 billion
- It was suggested that this was about twice as fast as VB encoding
  - Decoding gaps is much simpler – no bit masking
  - First byte can be decoded with lookup table or switch

# SIMPLE-9 [ANH & MOFFAT, 2004]

- A word-aligned, multiple number encoding scheme
- How can we store several numbers in 32 bits with a format selector?



# Simple9 Encoding Scheme [Anh & Moffat, 2004]

- Encoding block: 4 bytes (32 bits)
- Most significant nibble (4 bits) describe the layout of the 28 other bits as follows:
  - 0: a single 28-bit number
  - 1: two 14-bit numbers
  - 2: three 9-bit numbers (and one spare bit)
  - 3: four 7-bit numbers
  - 4: five 5-bit numbers (and three spare bits)
  - 5: seven 4-bit numbers
  - 6: nine 3-bit numbers (and one spare bit)
  - 7: fourteen two-bit numbers
  - 8: twenty-eight one-bit numbers
- Simple16 is a variant with 5 additional (uneven) configurations
- Efficiently decoded with hand-coded decoder, using bit masks
- Extended Simple Family – idea applies to 64-bit words, etc.

Layout (4 bits)	n numbers of b bits each $n * b \leq 28$
--------------------	---

# SUMMARY

- Postings compression
- Gap encoding of postings file entries
- Three postings entries

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

**THANK YOU !!!**



NM 100  
1923 - 2014  
*M a n i z*

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT II - Session 8 & 9)**

**UNIT II : Index construction and Compression**

**TOPIC : Term frequency and weighting**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Deploy an algorithm for indexing using suitable index construction and compression methods for a given database

## **Session Outcome :**

- Inverse document frequency and TF-IDF weighting

# RECAP OF PREVIOUS SESSION

- Postings compression
- Gap encoding of postings file entries
- Three postings entries

# CURRENT SESSION

- Term frequency
- Log-frequency weighting
- idf weight
- tf-idf weighting

# TERM FREQUENCY TF

- The term frequency  $tf_{t,d}$  of term  $t$  in document  $d$  is defined as the number of times that  $t$  occurs in  $d$ .
- We want to use tf when computing query-document match scores. But how?
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
  - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

NB: frequency = count in IR

# LOG-FREQUENCY WEIGHTING

- The log frequency weight of term  $t$  in  $d$  is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \text{tf}_{t,d}, & \text{if } \text{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 1.3, 10 \rightarrow 2, 1000 \rightarrow 4$ , etc.
- Score for a document-query pair: sum over terms  $t$  in both  $q$  and  $d$ :
- $\text{score} = \sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})$
- The score is 0 if none of the query terms is present in the document.

# DOCUMENT FREQUENCY

- Rare terms are more informative than frequent terms
  - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric*
- → We want a high weight for rare terms like *arachnocentric*.

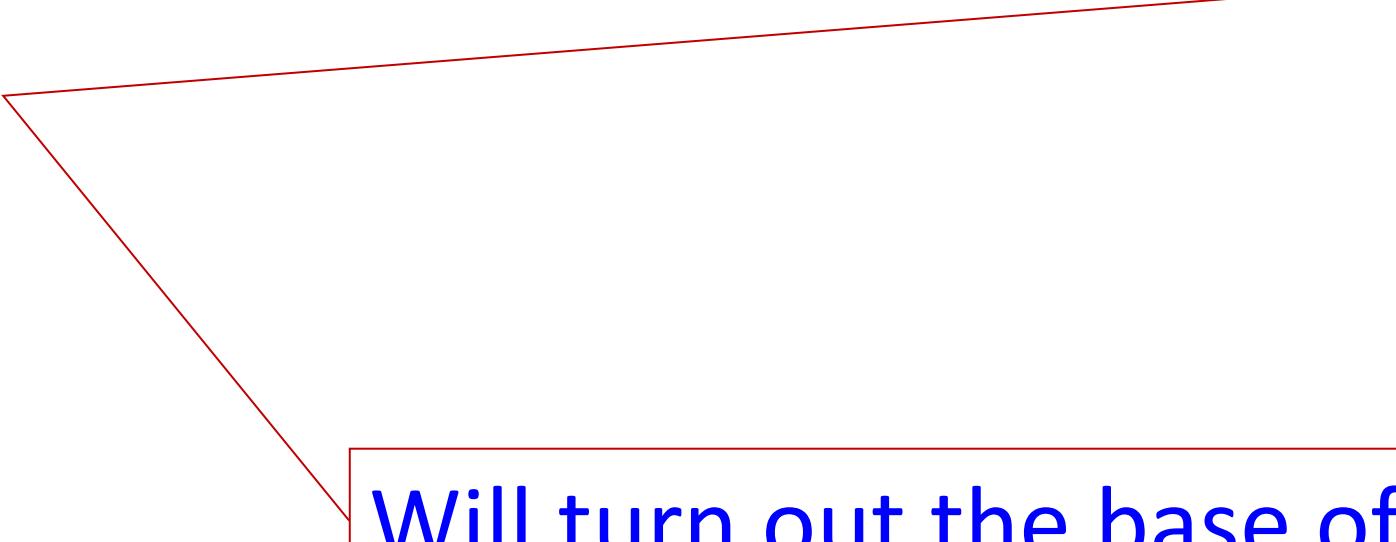
# DOCUMENT FREQUENCY, CONTINUED

- Frequent terms are less informative than rare terms
- Consider a query term that is frequent in the collection (e.g., *high*, *increase*, *line*)
- A document containing such a term is more likely to be relevant than a document that doesn't
- But it's not a sure indicator of relevance.
  - → For frequent terms, we want high positive weights for words like *high*, *increase*, and *line*
- But lower weights than for rare terms.
- We will use document frequency (df) to capture this.

## IDF WEIGHT

- $\text{df}_t$  is the document frequency of  $t$ : the number of documents that contain  $t$ 
  - $\text{df}_t$  is an inverse measure of the informativeness of  $t$
  - $\text{df}_t \leq N$
- We define the idf (inverse document frequency) of  $t$  by
  - We use  $\log(N/\text{df}_t)$  instead of  $N/\text{df}_t$  to “dampen” the effect of idf.

$$\text{idf}_t = \log_{10}(N/\text{df}_t)$$



Will turn out the base of the log is immaterial.

# EFFECT OF IDF ON RANKING

- Does idf have an effect on ranking for one-term queries, like
  - iPhone
- idf has no effect on ranking one term queries
  - idf affects the ranking of documents for queries with at least two terms
  - For the query **capricious person**, idf weighting makes occurrences of **capricious** count for much more in the final document ranking than occurrences of **person**.

# COLLECTION VS. DOCUMENT FREQUENCY

- The collection frequency of  $t$  is the number of occurrences of  $t$  in the collection, counting multiple occurrences.

# TF-IDF WEIGHTING

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$w_{t,d} = \log(1 + \text{tf}_{t,d}) \times \log_{10}(N / \text{df}_t)$$

- Best known weighting scheme in information retrieval
  - Note: the “-” in tf-idf is a hyphen, not a minus sign!
  - Alternative names: tf.idf, tf x idf
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

## SCORE FOR A DOCUMENT GIVEN A QUERY

$$\text{Score}(q, d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

There are many variants

How “tf” is computed (with/without logs)

Whether the terms in the query are also weighted

...

# BINARY → COUNT → WEIGHT MATRIX

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

Each document is now represented by a real-valued vector of tf-idf weights  $\in \mathbb{R}^{|V|}$

# SUMMARY

- Term frequency
- Log-frequency weighting
- idf weight
- tf-idf weighting

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

# **THANK YOU !!!**



NM 100

1923 - 2014

Ma 100

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT III - Session 1)**

**UNIT III : Vector Space Models and Evaluation**

**TOPIC : Vector Space Model**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Evaluate the vector space model for any given document using suitable evaluation techniques

## **Session Outcome :**

- Develop the basic ideas underlying vector space scoring

# CURRENT SESSION

## **Vector Space Model**

- Queries
- measures
- Ranking documents
- VSM variations

# OVERVIEW

- The Vector Space Model (VSM) is a way of representing documents through the words that they contain
- It is a standard technique in Information Retrieval
- The VSM allows decisions to be made about which documents are similar to each other and to keyword queries

## How it works: Overview

- Each document is broken down into a word frequency table
- The tables are called vectors and can be stored as arrays
- A vocabulary is built from all the words in all documents in the system
- Each document is represented as a vector based against the vocabulary

# Example

- **Document A**
  - “A dog and a cat.”

a	dog	and	cat
2	1	1	1

## Document B

- “A frog.”

a	frog
1	1

## Example, continued

- The vocabulary contains all words used
  - a, dog, and, cat, frog
- The vocabulary needs to be sorted
  - a, and, cat, dog, frog

## Example, continued

- **Document A:** “A dog and a cat.”
  - **Vector:** (2,1,1,1,0)

a	and	cat	dog	frog
2	1	1	1	0

- **Document B:** “A frog.”
  - **Vector:** (1,0,0,0,1)

a	and	cat	dog	frog
1	0	0	0	1

# QUERIES

- Queries can be represented as vectors in the same way as documents:
  - Dog = (0,0,0,1,0)
  - Frog = ( )
  - Dog and frog = ( )

# SIMILARITY MEASURES

- There are many different ways to measure how similar two documents are, or how similar a document is to a query
- The cosine measure is a very common similarity measure
- Using a similarity measure, a set of documents can be compared to a query and the most similar document returned

# THE COSINE MEASURE

- For two vectors  $d$  and  $d'$  the cosine similarity between  $d$  and  $d'$  is given by:

$$\frac{d \times d'}{|d||d'|}$$

- Here  $d \times d'$  is the vector product of  $d$  and  $d'$ , calculated by multiplying corresponding frequencies together
- The cosine measure calculates the angle between the vectors in a high-dimensional virtual space

## Example

- Let  $d = (2,1,1,1,0)$  and  $d' = (0,0,0,1,0)$ 
  - $d \times d' = 2 \times 0 + 1 \times 0 + 1 \times 0 + 1 \times 1 + 0 \times 0 = 1$
  - $|d| = \sqrt{2^2+1^2+1^2+1^2+0^2} = \sqrt{7} = 2.646$
  - $|d'| = \sqrt{0^2+0^2+0^2+1^2+0^2} = \sqrt{1} = 1$
  - Similarity =  $1/(1 \times 2.646) = 0.378$

# RANKING DOCUMENTS

- A user enters a query
- The query is compared to all documents using a similarity measure
- The user is shown the documents in decreasing order of similarity to the query term

# VSM VARIATIONS

- Stopword lists
  - Commonly occurring words are unlikely to give useful information and may be removed from the vocabulary to speed processing
  - Stopword lists contain frequent words to be excluded
  - Stopword lists need to be used carefully
    - E.g. “to be or not to be”

# SUMMARY

- **Vector Space Model**
  - Queries
  - measures
  - Ranking documents
  - VSM variations

## REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

**THANK YOU !!!**



NM 100

1923 - 2014

Memorial

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT III - Session 2)**

**UNIT III : Vector Space Models and Evaluation**

**TOPIC : TF-IDF functions**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Evaluate the vector space model for any given document using suitable evaluation techniques
- **Session Outcome :**
- Develop several variants of term-weighting for the vector space model.

# RECAP OF PREVIOUS SESSION

- **Vector Space Model**
  - Queries
  - measures
  - Ranking documents
  - VSM variations

# CURRENT SESSION

- What is TF-IDF?
- How TF-IDF is calculated
- Applications of TF-IDF

TF-IDF is one of the mechanisms used by search engines, in order to address the relevance of the user information being retrieved based on certain assumptions in general.

### Tf Idf Representation

Tf - term frequency ← bag of words  
Idf - inverse document frequency ← weighting by how often word occurs in corpus

Quiz: Would you weight common words higher, or rare words?

common

rare

## What is TF-IDF?

- TF-IDF, short for Term Frequency - Inverse Document Frequency, is a text mining technique
  - gives a numeric statistic as to how important a word is to a document in a collection or corpus.
- Technique used to categorize documents according to certain words and their importance to the document.

## Example

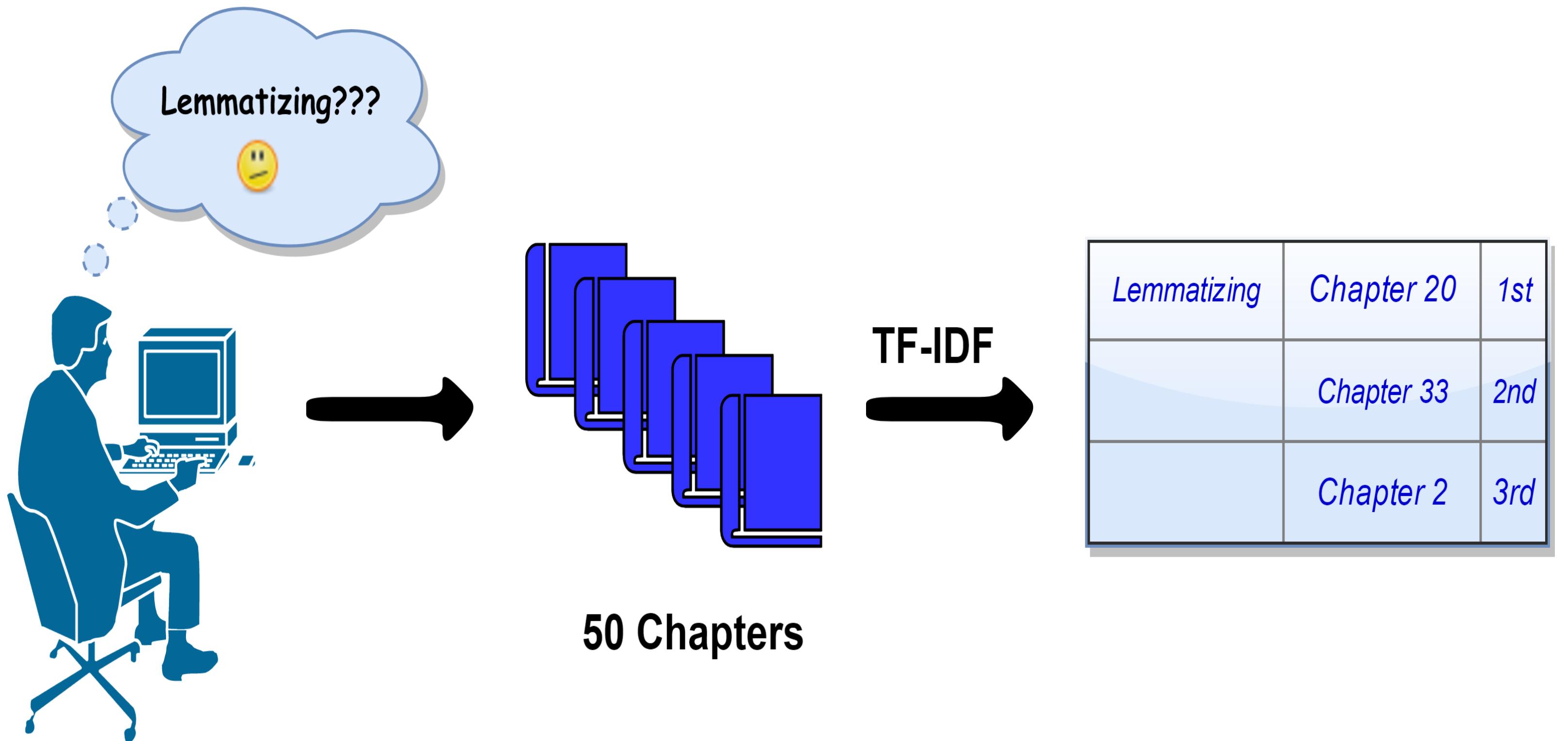
- Similar to Bag of Words (BoW), where BoW is an algorithm that counts how many times a word appears in a document.
- If a particular term appears in a document, many times, then there is a possibility of that term being an important word in that document.
- This is the basic concept of BoW, where the word count allows us to compare and rank documents based on their similarities for applications like search, document classification and text mining.
- Each of these are modeled into a vector space so as to easily categorize terms and their documents. But the general BoW technique, does not omit the common words that appear in documents and it is also being modeled in the vector space.



- But when it comes to TF-IDF, this is also considered as a measure to categorize documents based on the terms that appear in it. But unlike BoW, this does provide a weight for each term, rather than just the count.
- The TF-IDF value measures the relevance, not frequency.

## Example:

- Assume you are trying to finish an Assignment from your information retrieval class. The last question says something about "Lemmatizing"
- So you wish to look this up in your text book, which has around 50 chapters.
- How can you find which chapter has the correct information?
- With TF-IDF you can easily identify which word is important in which document.



50 Chapters

# Why TF-IDF?

- TF-IDF allows us to score the importance of words in a document, based on how frequently they appear on multiple documents.
- If the word appears frequently in a document - assign a high score to that word (term frequency - TF)
- If the word appears in a lot of documents - assign a low score to that word. (inverse document frequency - IDF)

## How TF-IDF is calculated

- TF-IDF is the product of two main statistics, term frequency and the inverse document frequency. Different information retrieval systems use various calculation mechanisms, but here we present the most general mathematical formulas. TF-IDF is calculated to all the terms in a document.

$$\text{tf-idf}_{(t,d)} = \text{tf}_{(t,d)} \times \text{idf}_{(t)}$$

t = term

d = document

# Calculating Term Frequency

$$tf(t, d) = \frac{\text{Frequency of term } t, \text{ in document } d}{\text{Total number of terms in document } d}$$

# Calculating Inverse Document Frequency

$$\text{idf}(t) = \log \frac{\text{Total number of documents}}{\text{Number of documents with term } t \text{ in it}}$$

## Example:

- onsider a document d, where the word "car" appears 6 times in it. That document contains 600 words.
- $\text{tf} ("car" , d) = 6/600 = 0.01$
- Now, we have a collection of 10,000 document, and the word "cat" appears in 300 of these documents.
- $\text{idf} ("car" ) = \log ( 10000/ 300) = 1.523$
- $\text{tf-idf} ("car" , d) = 0.01 \times 1.523 = 0.01523$

# Document Ranking for a Given Query

- Finding the rank of documents for a query, we need to calculate the score of the document for a given query.

$$\text{Score}(q, d) = \sum_{t \in q \cap d} tfidf(t, d)$$

q = query

t = term

d = document

# Applications of TF-IDF

## 1. Automatic Website Tagging

- TF-IDF becomes very useful when you have a very large document set, and you wish to develop an information retrieval system on them, or simply categorize the documents.
- TF-IDF will be picking the best and unique tags (words) from each of the documents and assigning them a score within each document.



## 2. Web Search Engines

- Company based information retrieval systems, web search engines, and website search bars, use different variations of TF-IDF weighting so as to achieve best quality results with less trade-offs on the other quality factors like time and relevance.



### 3.Digital Libraries

TF-IDF grows daily where Wikipedia says that, 83% of Text based recommender systems in the domain of digital library use TF-IDF. A digital library is an information hub that contains electronic information such as texts, books, images, graphs etc.



# SUMMARY

- What is TF-IDF?
- How TF-IDF is calculated
- Applications of TF-IDF

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

# **THANK YOU !!!**



NM 100  
1923 - 2014  
M A H A L I N G A M

# **19CSEN2003-Information Retrieval Techniques** **(UNIT III - Session 3)**

**UNIT III : Vector Space Models and Evaluation**  
**TOPIC : Scoring & Ranking**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Evaluate the vector space model for any given document using suitable evaluation techniques

## **Session Outcome :**

- Develop efficient scoring and ranking vector space model.

# RECAP OF PREVIOUS SESSION

- What is TF-IDF?
- How TF-IDF is calculated
- Applications of TF-IDF

# CURRENT SESSION

## Scoring & Ranking

- Represent each document as a weighted tf-idf vector
- Rank documents with respect to the query by score

# RANKED RETRIEVAL MODELS

- A set of documents satisfying a query expression, in ranked retrieval, the system returns an ordering over the (top) documents in the collection for a query
- Free text queries: Rather than a query language of operators and expressions, the user's query is just one or more words in a human language
- Ranked list of results: No more feast or famine

## Scoring as the basis of ranked retrieval

- We wish to return in order the documents most likely to be useful to the searcher
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in  $[0, 1]$  – to each document
- This score measures how well document and query “match”.

# QUERY-DOCUMENT MATCHING SCORES

- We need a way of assigning a score to a query/document pair
- If the query term does not occur in the document: score should be 0
- The more frequent the query term in the document, the higher the score (should be)

# JACCARD COEFFICIENT

- $\text{jaccard}(A,B) = |A \cap B| / |A \cup B|$
- $\text{jaccard}(A,A) = 1$
- $\text{jaccard}(A,B) = 0$  if  $A \cap B = 0$
- Always assigns a number between 0 and 1.

Attributes occurring in both samples	4
Attributes occurring in sample $X$ only	1
Attributes occurring in sample $Y$ only	2

$$=4/(4+1+2)$$

Jaccard's Similarity Index → 0.57

# ISSUES WITH JACCARD FOR SCORING

- Privileges shorter documents
  - We need a more sophisticated way of normalizing

for length  $|A \cap B| / \sqrt{|A \cup B|}$

- It doesn't consider *term frequency*
  - how many times a term occurs in a document
  - Does not account for term *informativeness*
    - *How important is the term in the document*

# ACCOUNTING FOR TERM FREQUENCY

	<b>Antony and Cleopatra</b>	<b>Julius Caesar</b>	<b>The Tempest</b>	<b>Hamlet</b>	<b>Othello</b>	<b>Macbeth</b>
<b>Antony</b>	1	1	0	0	0	1
<b>Brutus</b>	1	1	0	1	0	0
<b>Caesar</b>	1	1	0	1	1	1
<b>Calpurnia</b>	0	1	0	0	0	0
<b>Cleopatra</b>	1	0	0	0	0	0
<b>mercy</b>	1	0	1	1	1	1
<b>worser</b>	1	0	1	1	1	0

Each document is represented by a binary vector  $\in \{0,1\}^M$

# TERM FREQUENCY TF

- The term frequency  $tf_{t,d}$  of term  $t$  in document  $d$  is defined as the number of times that  $t$  occurs in  $d$ .
- We want to use tf when computing query- document match scores. But how?
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
  - But not 10 times more relevant.

# LOG-FREQUENCY WEIGHTING

- The log frequency weight of term  $t$  in  $d$  is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \text{tf}_{t,d}, & \text{if } \text{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 1.3, 10 \rightarrow 2, 1000 \rightarrow 4$ , etc.
- Score for a document-query pair: sum over terms  $t$  in both  $q$  and  $d$ :
- $\text{score} = \sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})$
- The score is 0 if none of the query terms is present in the document.

# DOCUMENT FREQUENCY

- Rare terms are more informative than frequent terms
  - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric*
- → We want a high weight for rare terms like *arachnocentric*.

# DOCUMENT FREQUENCY, CONTINUED

- Frequent terms are less informative than rare terms
- Consider a query term that is frequent in the collection (e.g., *high*, *increase*, *line*)
- A document containing such a term is more likely to be relevant than a document that doesn't
- But it's not a sure indicator of relevance.
  - How/when will it break?

# IDF WEIGHT

- $df_t$  is the document frequency of  $t$ : the number of documents that contain  $t$ 
  - $df_t$  is an inverse measure of the informativeness of  $t$
  - $df_t \square N$
- We define the idf (inverse document frequency) of  $t$  by
$$idf_t \square \log_{10} (N/df_t)$$
  - We use  $\log (N/df_t)$  instead of  $N/df_t$  to “dampen” the effect of idf.

## idf example, suppose $N = 1$ million

term	$\text{df}_t$	$\text{idf}_t$
calpurnia	1	6
animal	100	4
sunday	1,000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

$$\text{idf}_t = \log_{10} (N/\text{df}_t)$$

There is one idf value for each term  $t$  in a collection.

# TF.IDF WEIGHTING

- The tf.idf weight of a term is the product of its tf weight and its idf weight.

$$w_{t,d} = \log(1 + tf_{t,d}) \cdot \log_{10}(N / df_t)$$

- Best known weighting scheme in information retrieval
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

# EFFECT OF IDF ON RANKING

- Does idf have an effect on ranking for one- term queries, like
  - iPhone
- idf has no effect on ranking one term queries
  - idf affects the ranking of documents for queries with at least two terms
  - For the query **capricious person**, idf weighting makes occurrences of **capricious** count for much more in the final document ranking than occurrences of **person**.

# SCORE FOR A DOCUMENT GIVEN A QUERY

$$\text{Score}(q,d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

- There are many variants
  - How “tf” is computed (with/without logs)
  - Whether the terms in the query are also weighted
  - ...

# Weighting may differ in queries vs documents

- Many search engines allow for different weightings for queries vs. documents
- SMART Notation: denotes the combination in use in an engine, with the notation *ddd.ddd*, using the acronyms from the previous table
  - A very standard weighting scheme is: Inc.ltc
    - Document: logarithmic tf (l as first character), no idf and cosine normalization
    - Query: logarithmic tf (l in leftmost column), idf (t in second column), no normalization ...

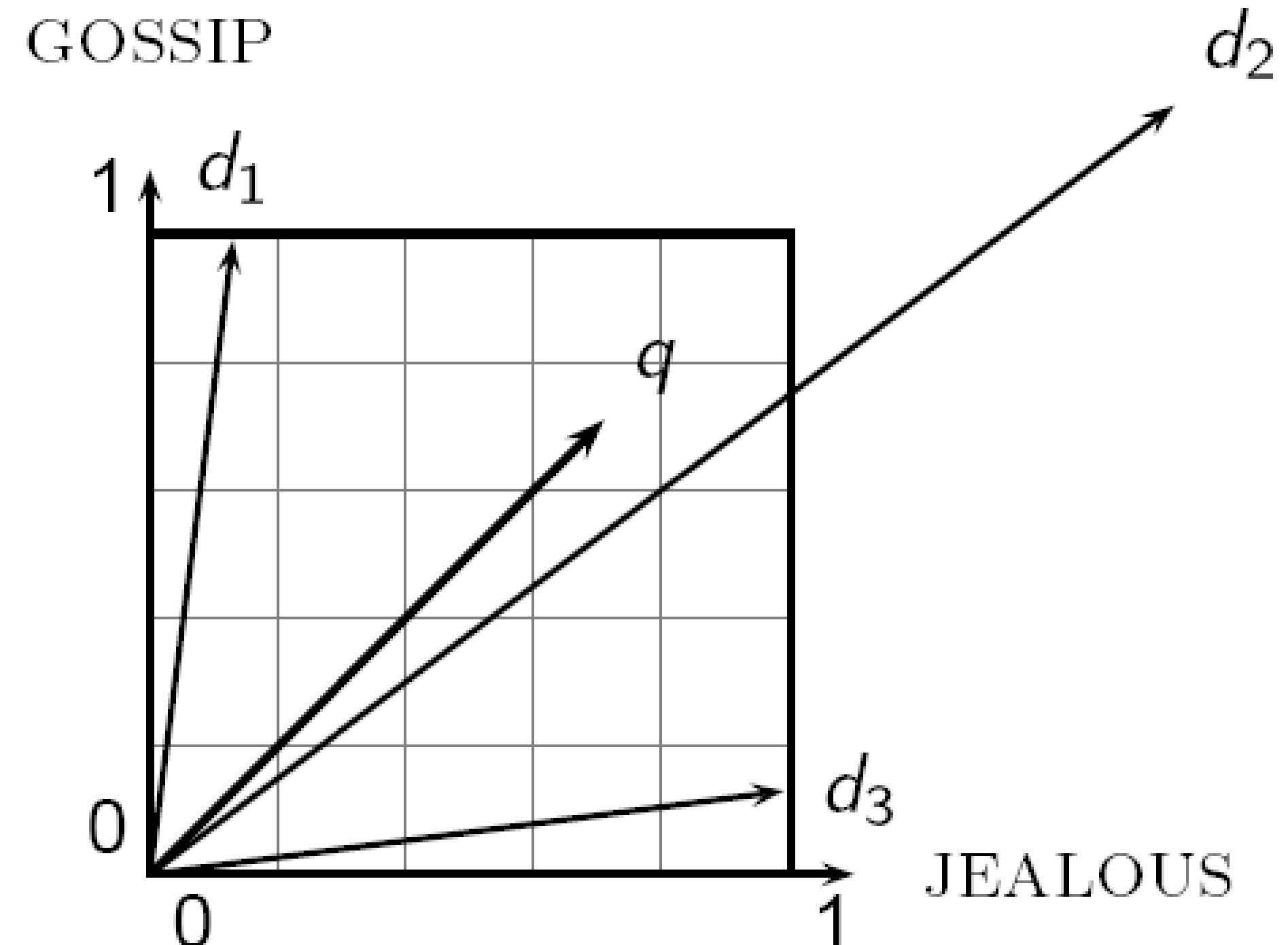
## Binary → count → weight matrix

	<b>Antony and Cleopatra</b>	<b>Julius Caesar</b>	<b>The Tempest</b>	<b>Hamlet</b>	<b>Othello</b>	<b>Macbeth</b>
<b>Antony</b>	<b>5.25</b>	<b>3.18</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0.35</b>
<b>Brutus</b>	<b>1.21</b>	<b>6.1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>
<b>Caesar</b>	<b>8.59</b>	<b>2.54</b>	<b>0</b>	<b>1.51</b>	<b>0.25</b>	<b>0</b>
<b>Calpurnia</b>	<b>0</b>	<b>1.54</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>Cleopatra</b>	<b>2.85</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>mercy</b>	<b>1.51</b>	<b>0</b>	<b>1.9</b>	<b>0.12</b>	<b>5.25</b>	<b>0.88</b>
<b>worser</b>	<b>1.37</b>	<b>0</b>	<b>0.11</b>	<b>4.15</b>	<b>0.25</b>	<b>1.95</b>

Each document is now represented by a real-valued vector of tf-idf weights  $\in \mathbb{R}^{|V|}$

# EUCLIDEAN DISTANCE IS A BAD IDEA

- The Euclidean distance between  $q$
- and  $d_2$  is large even though the
- distribution of terms in the query  $q$  and the distribution of
- terms in the document  $d_2$  are
- very similar.



# COSINE(QUERY, DOCUMENT)

Dot product      Unit vectors

$$\cos(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{\|\vec{q}\| \|\vec{d}\|} = \frac{\vec{q} \bullet \frac{\vec{d}}{\|\vec{d}\|}}{\|\vec{q}\|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

$q_i$  is the tf-idf weight of term  $i$  in the query  
 $d_i$  is the tf-idf weight of term  $i$  in the document

$\cos(\vec{q}, \vec{d})$  is the cosine similarity of  $\vec{q}$  and  $\vec{d}$ ... or,  
 equivalently, the cosine of the angle between  $\vec{q}$  and  $\vec{d}$ .

# LENGTH NORMALIZATION

- A vector can be (length-) normalized by dividing each of its components by its length – for this we use the  $L_2$  norm:

$$\|x\|_2 = \sqrt{\sum_i x_i^2}$$

- Dividing a vector by its  $L_2$  norm makes it a unit (length) vector (on surface of unit hypersphere)
- Effect on the two documents  $d$  and  $d'$  ( $d$  appended to itself) from earlier slide: they have identical vectors after length-normalization.
  - Long and short documents now have comparable weights

# COSINE FOR LENGTH-NORMALIZED VECTORS

- For length-normalized vectors, cosine similarity is simply the dot product (or scalar product):

$$\cos(q, d) = q \bullet d = \sum_{i=1}^{|V|} q_i d_i$$

for  $q, d$  length-normalized.

# SUMMARY

- Represent each document as a weighted tf-idf vector
- Rank documents with respect to the query by score

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

**THANK YOU !!!**



NM 100

1923 - 2014

Memorial

# **19CSEN2003-Information Retrieval Techniques**

## **(UNIT III - Session 4)**

**UNIT III : Vector Space Models and Evaluation**

**TOPIC : Measuring Effectiveness of test collections**

**Ms.M.Gomathi**  
**Assistant Professor/CSE**

# OUTCOMES

## **Course Outcome :**

- Evaluate the vector space model for any given document using suitable evaluation techniques

## **Session Outcome :**

- Explain standard test collections used for measuring the effectiveness.

# RECAP OF PREVIOUS SESSION

- Represent each document as a weighted tf-idf vector
- Rank documents with respect to the query by score

# CURRENT SESSION

## Evaluation in information retrieval

- Relevant vs. Retrieved Documents
- Precision vs. Recall

# INFORMATION RETRIEVAL SYSTEM EVALUATION

- What to Evaluate?
- How much of the information need is satisfied.
- How much was learned about a topic.
- Incidental learning:
  - How much was learned about the collection.
  - How much was learned about other topics.
- How easy the system is to use.
- **Usually based on what documents we retrieve**

# WHAT TO EVALUATE?

What can be measured that reflects users' ability to use a system?  
(Cleverdon 66)

- Coverage of Information
- Form of Presentation
- Effort required/Ease of Use
- Time and Space Efficiency
- Effectiveness

Effectiveness!

- Recall
  - proportion of relevant material actually retrieved
- Precision
  - proportion of retrieved material actually relevant

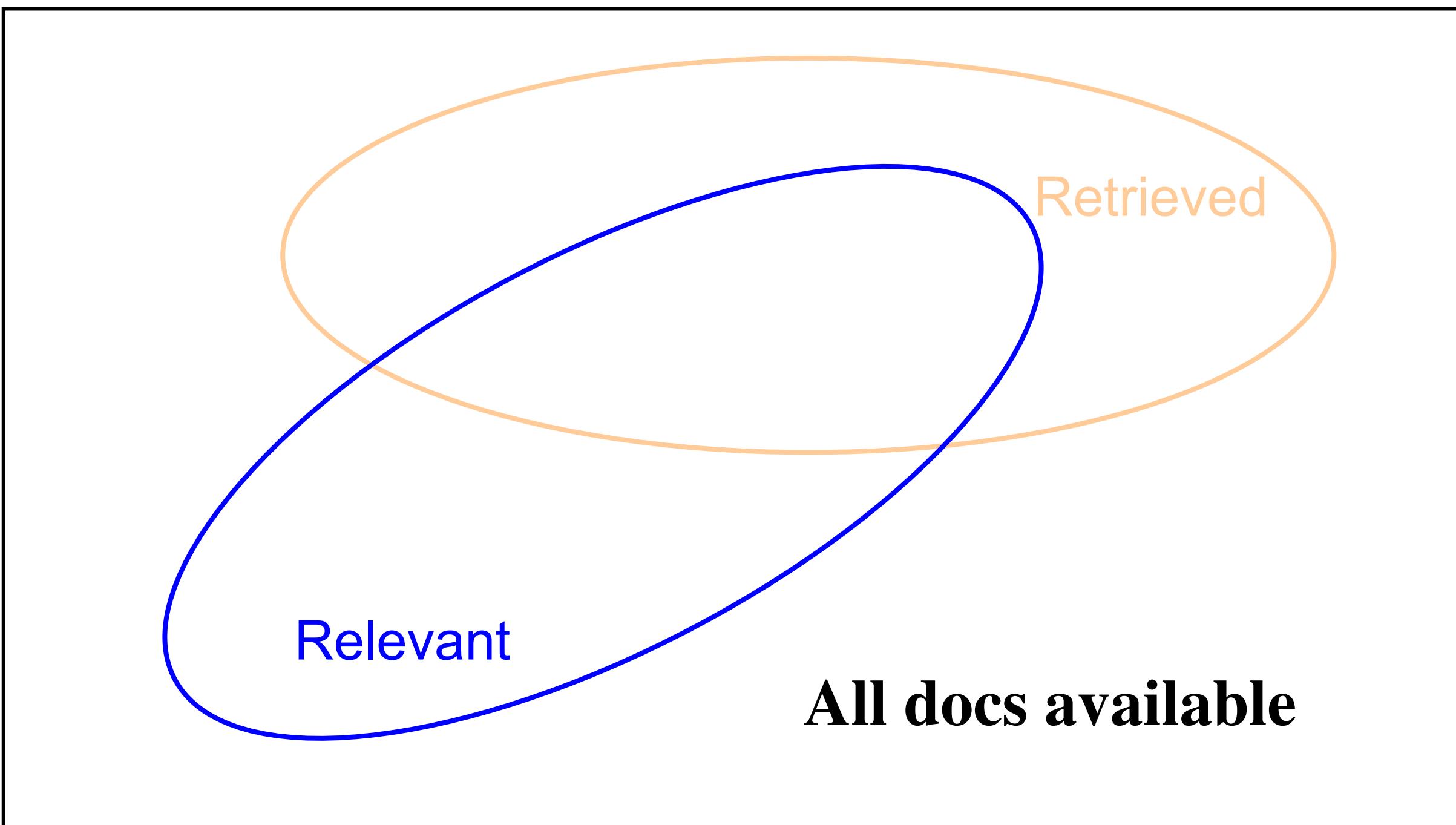
# HOW DO WE MEASURE RELEVANCE?

- Measures of relevance:
  - Binary measure
    - 1 relevant
    - 0 not relevant
  - N-ary measure
    - 3 very relevant
    - 2 relevant
    - 1 barely relevant
    - 0 not relevant
  - Negative values?
- N=? consistency vs. expressiveness tradeoff

# GIVEN: we have a relevance ranking of documents

- Have some known relevance evaluation
  - Query independent – based on information need
  - Experts (or you)
- Apply binary measure of relevance
  - 1 - relevant
  - 0 - not relevant
- Put in a query
  - Evaluate relevance of what is returned
- What comes back?
  - Example: lion

# RELEVANT VS. RETRIEVED DOCUMENTS



# CONTINGENCY TABLE OF RELEVANT AND RETRIEVED DOCUMENTS

		<u>relevant</u>	
		Rel	NotRel
<u>retrieved</u>	Ret	Ret <sub>Rel</sub>	Ret <sub>NotRel</sub>
	NotRet	NotRet <sub>Rel</sub>	NotRet <sub>NotRel</sub>

$$\text{Ret} = \text{Ret}_{\text{Rel}} + \text{Ret}_{\text{NotRel}}$$

$$\text{NotRet} = \text{NotRet}_{\text{Rel}} + \text{NotRet}_{\text{NotRel}}$$

$$\text{Relevant} = \text{Ret}_{\text{Rel}} + \text{NotRet}_{\text{Rel}}$$

$$\text{Not Relevant} = \text{Ret}_{\text{NotRel}} + \text{NotRet}_{\text{NotRel}}$$

$$\text{Total \# of documents available } N = \text{Ret}_{\text{Rel}} + \text{NotRet}_{\text{Rel}} + \text{Ret}_{\text{NotRel}} + \text{NotRet}_{\text{NotRel}}$$

- Precision:  $P = \text{Ret}_{\text{Rel}} / \text{Retrieved}$

$$P = [0,1]$$

- Recall:  $R = \text{Ret}_{\text{Rel}} / \text{Relevant}$

$$R = [0,1]$$

		Actual condition	
		Present	Absent
Test result	Positive	Condition Present + Positive result = True Positive	Condition absent + Positive result = False Positive <b>Type I error</b>
	Negative	Condition present + Negative result = False (invalid) Negative <b>Type II error</b>	Condition absent + Negative result = True (accurate) Negative

Example, using infectious disease test results:

		Actual condition	
		Infected	Not infected
Test result	Test shows "infected"	True Positive	False Positive (i.e. infection reported but not present) <b>Type I error</b>
	Test shows "not infected"	False Negative (i.e. infection not detected) <b>Type II error</b>	True Negative

Example, testing for guilty/not-guilty:

		Actual condition	
		Guilty	Not guilty
Test result	Verdict of "guilty"	True Positive	False Positive (i.e. guilt reported unfairly) <b>Type I error</b>
	Verdict of "not guilty"	False Negative (i.e. guilt not detected) <b>Type II error</b>	True Negative

Example, testing for innocent/not innocent – sense is reversed from previous example:

		Actual condition	
		Innocent	Not innocent
Test result	Judged "innocent"	True Positive	False Positive (i.e. guilty but not caught) <b>Type I error</b>
	Judged "not innocent"	False Negative (i.e. innocent but condemned) <b>Type II error</b>	True Negative

# EXAMPLE

- Documents available:  
D1,D2,D3,D4,D5,D6,D7,D  
8,D9,D10
- Relevant: D1, D4, D5, D8,  
D10
- Query to search engine  
retrieves: D2, D4, D5, D6,  
D8, D9

	relevant	not relevant
retrieved		
not retrieved		

# EXAMPLE

- Documents available:  
D1,D2,D3,D4,D5,D6,D  
7,D8,D9,D10
- Relevant: D1, D4, D5,  
D8, D10
- Query to search engine  
retrieves: D2, D4, D5,  
D6, D8, D9

	relevant	not relevant
retrieved	D4,D5,D 8	D2,D6,D9
not retrieved	D1,D10	D3,D7

# Precision and Recall – Contingency Table

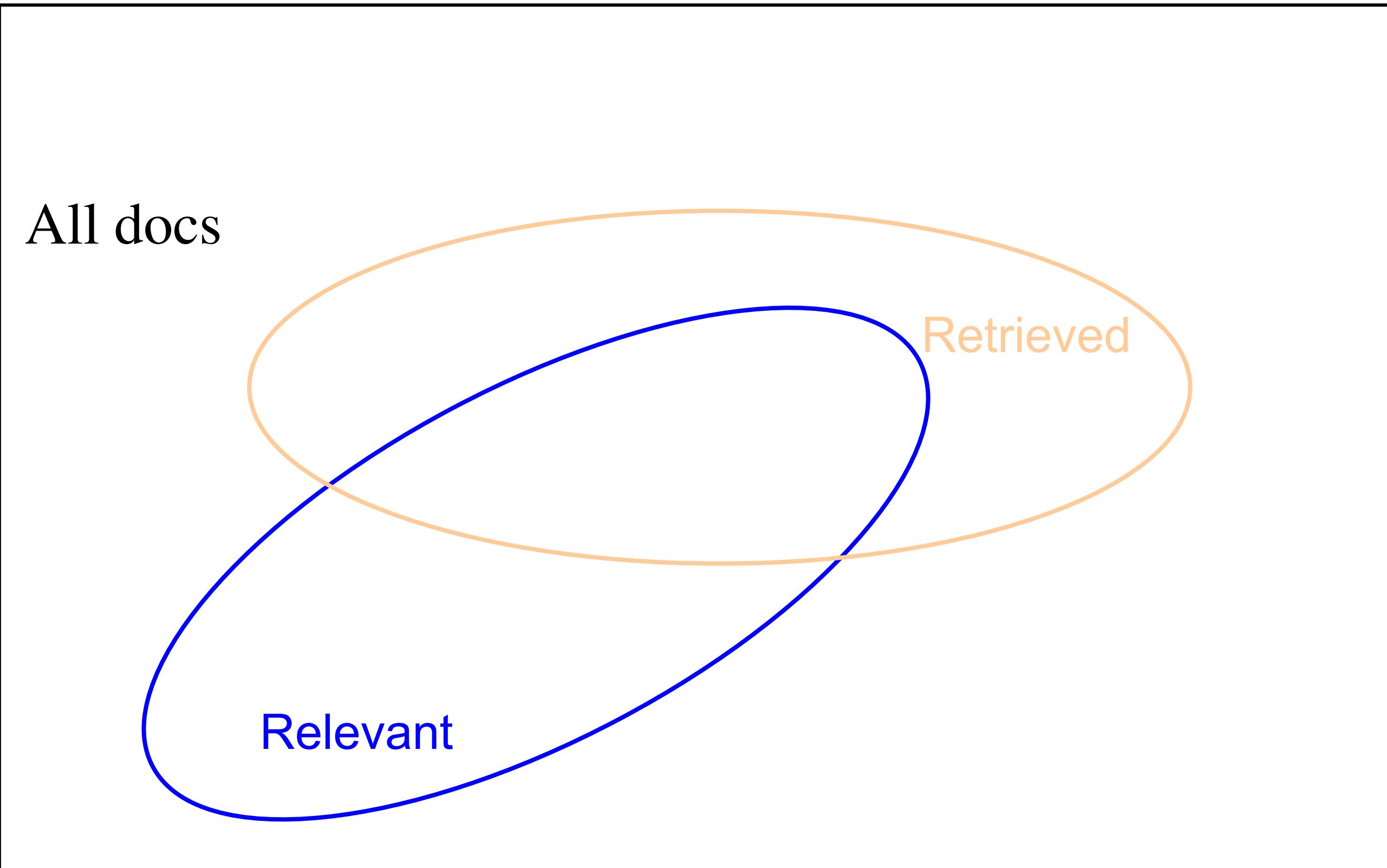
		Retrieved	Not retrieved	
		w=3	x=2	Relevant = w+x= 5
Relevant	Retrieved	w=3	x=2	
	Not retrieved	y=3	z=2	Not Relevant = y+z = 5
Retrieved = w+y = 6		Not Retrieved = x+z = 4		

$$\text{Total documents } N = w+x+y+z = 10$$

- Precision:  $P = w / w+y = 3/6 = .5$

- Recall:  $R = w / w+x = 3/5 = .6$

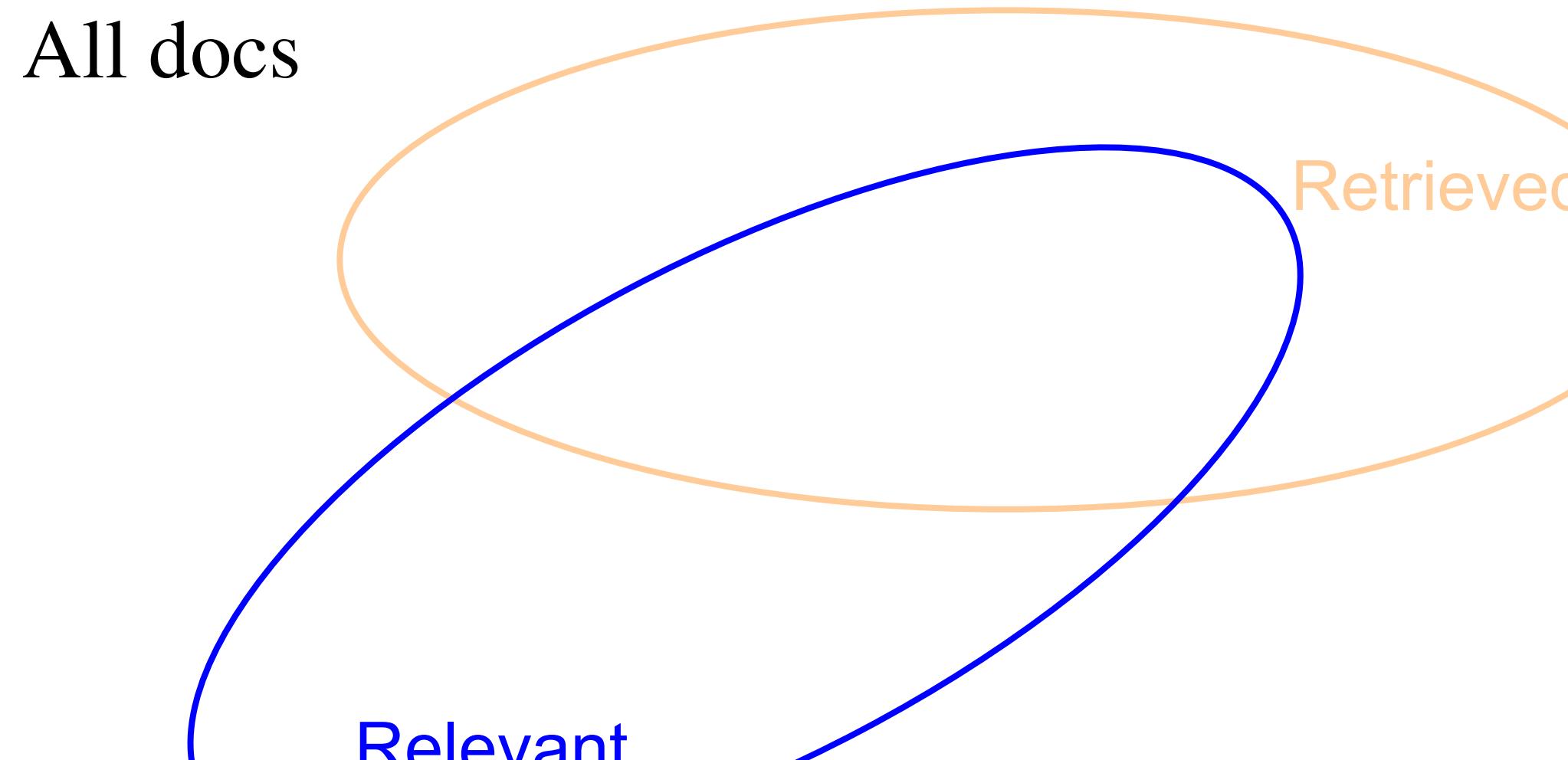
# Relevant vs. Retrieved



# PRECISION VS. RECALL

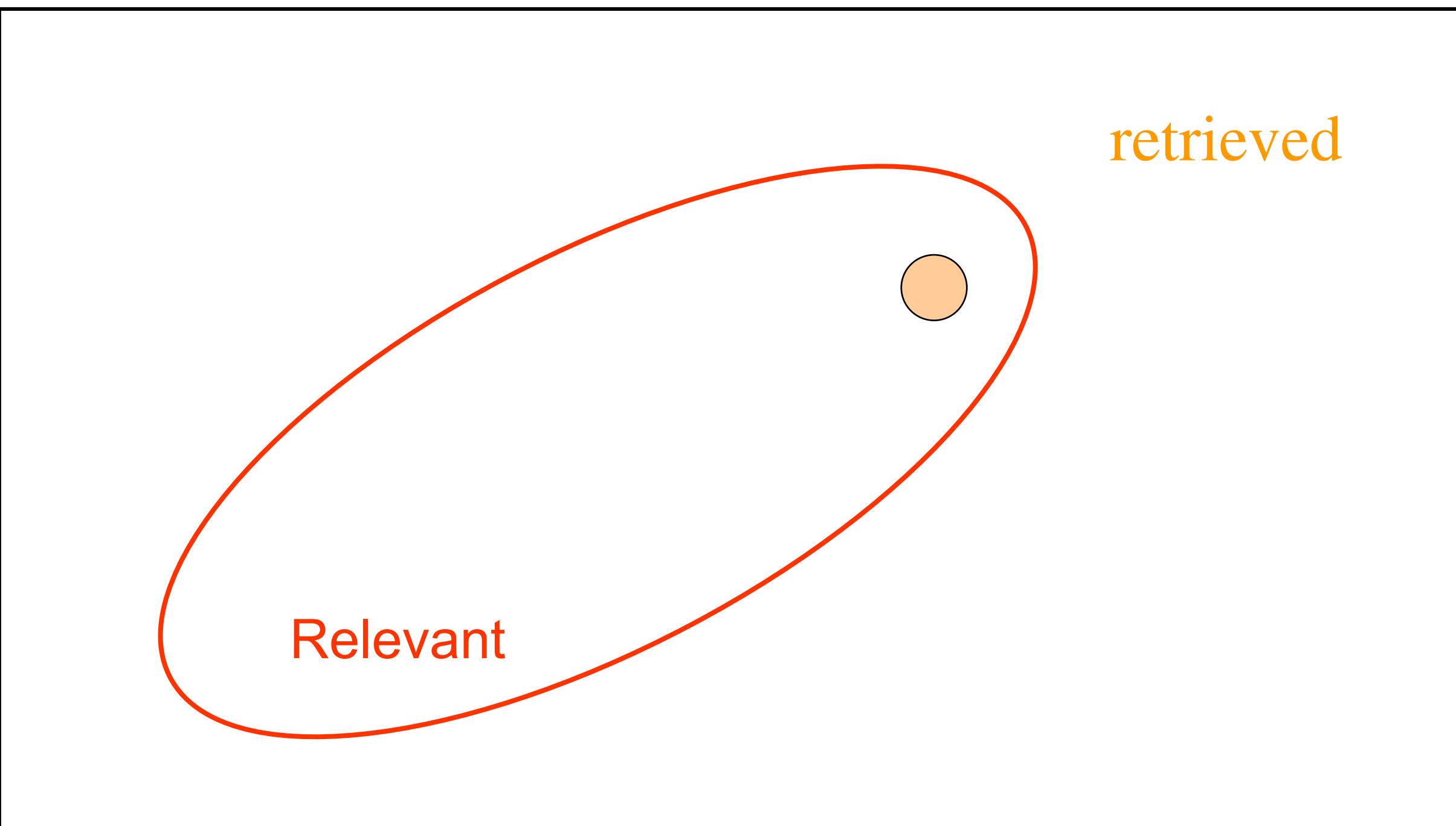
$$\text{Precision} = \frac{|\text{RelRetrieved}|}{|\text{Retrieved}|}$$

$$\text{Recall} = \frac{|\text{RelRetrieved}|}{|\text{Rel in Collection}|}$$



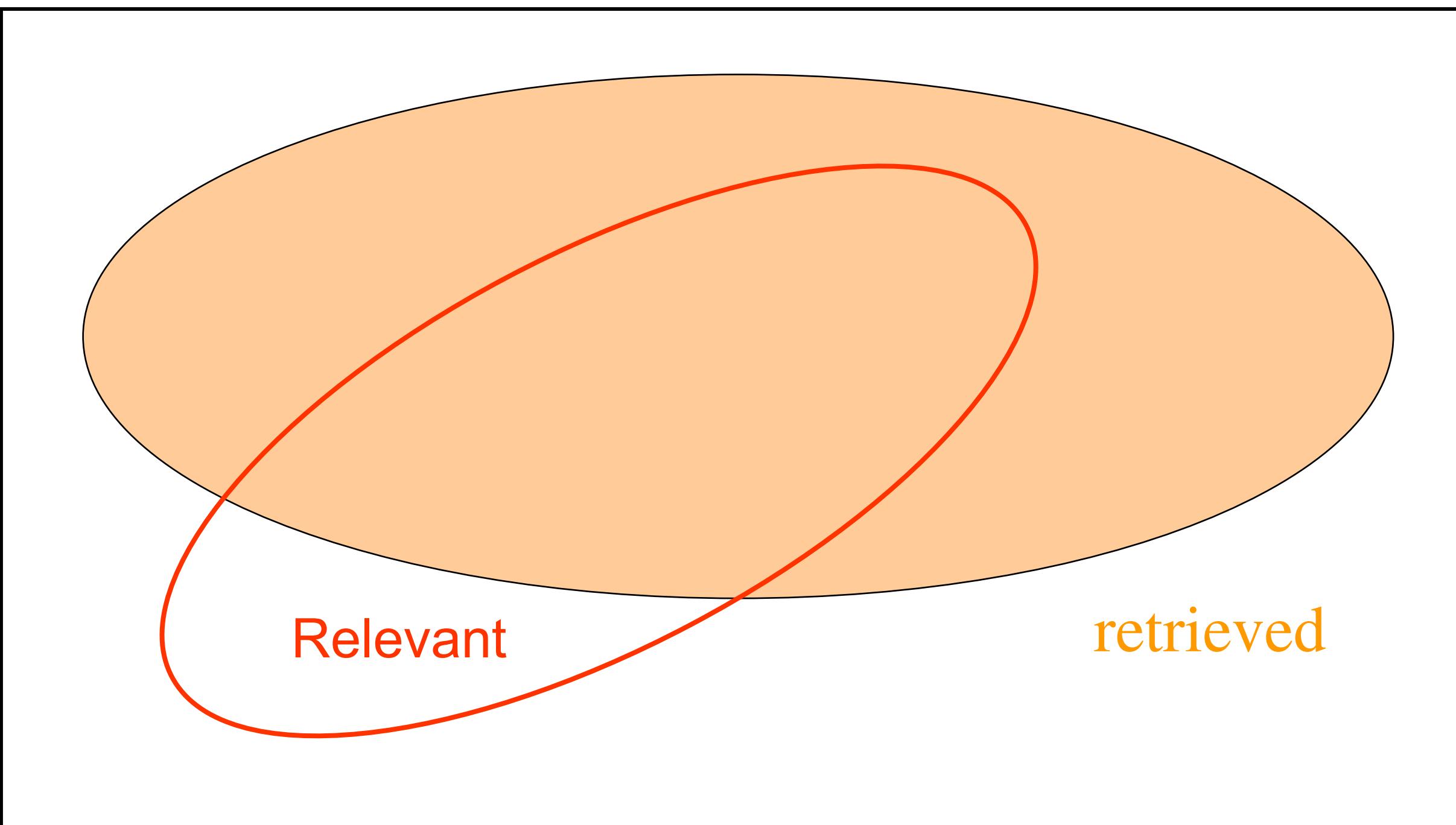
# RETRIEVED VS. RELEVANT DOCUMENTS

- Very high precision, very low recall



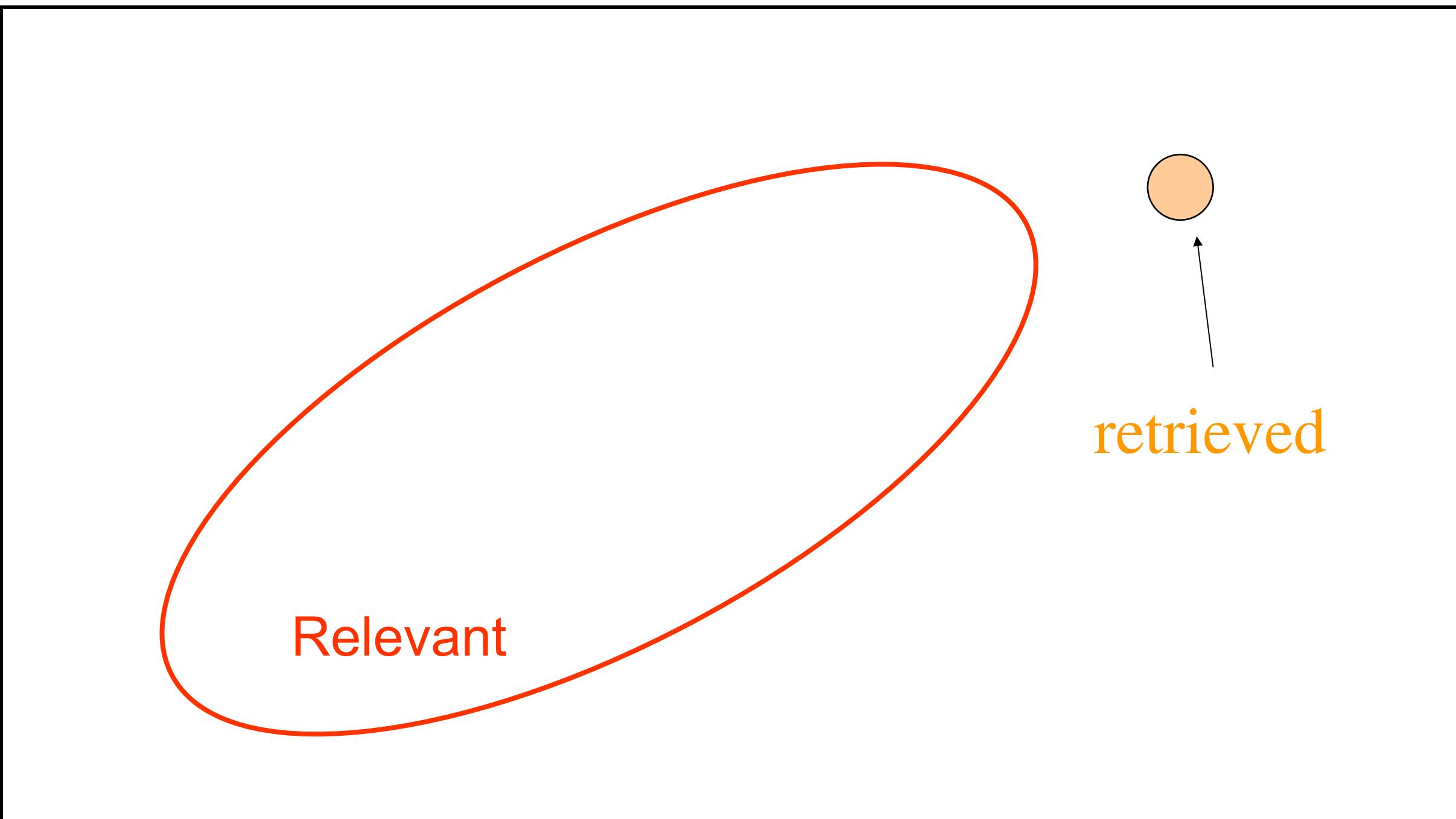
# RETRIEVED VS. RELEVANT DOCUMENTS

- High recall, but low precision



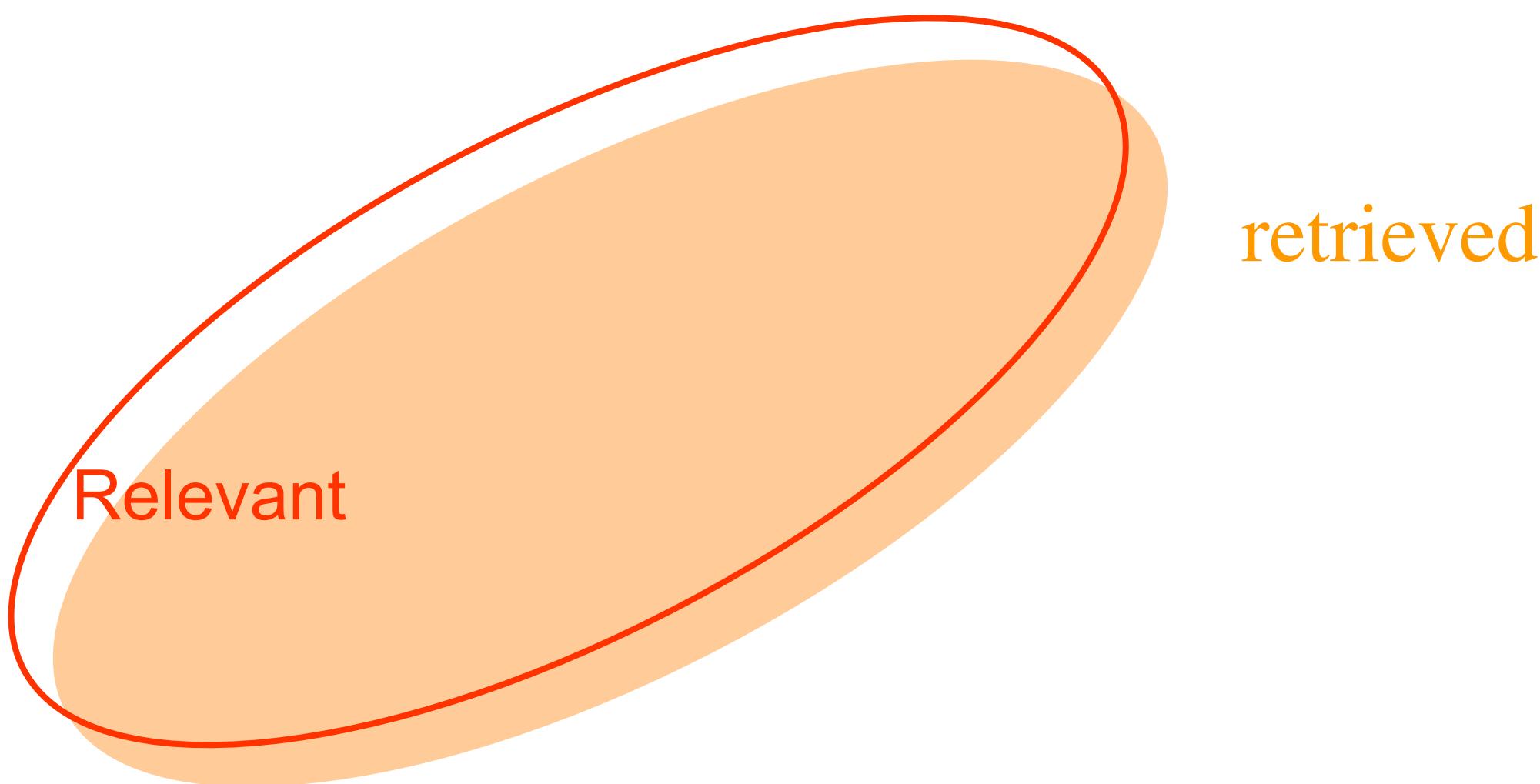
# RETRIEVED VS. RELEVANT DOCUMENTS

- Very low precision, very low recall (0 for both)



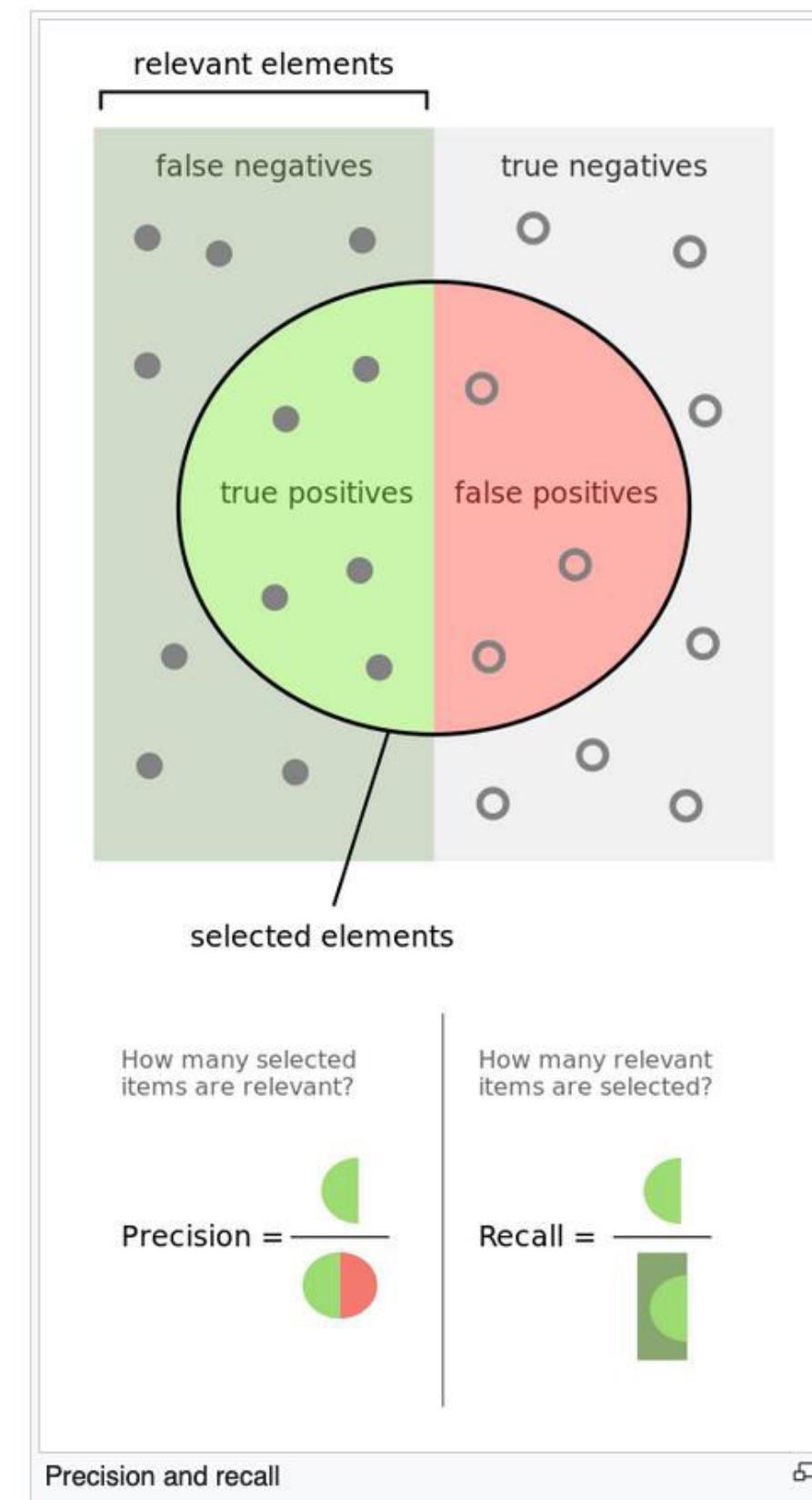
# RETRIEVED VS. RELEVANT DOCUMENTS

- High precision, high recall (at last!)



# Precision vs Recall

TP, FP, TN, FN



# WHY PRECISION AND RECALL?

Get as much of what we want while at the same time getting as little junk as possible.

Recall is the percentage of relevant documents returned compared to everything that is available!

Precision is the percentage of relevant documents compared to what is returned!

What different situations of recall and precision can we have?

# EXPERIMENTAL RESULTS

- Much of IR is experimental!
- Formal methods are lacking
  - Role of artificial intelligence hopes to change this
- Derive much insight from these results

Rec- recall      NRel - # relevant

Prec - precision

- **Retrieve one document at a time without replacement and in order.**

- Given: only 25 documents of which 5 are relevant (D1, D2, D4, D15, D25)

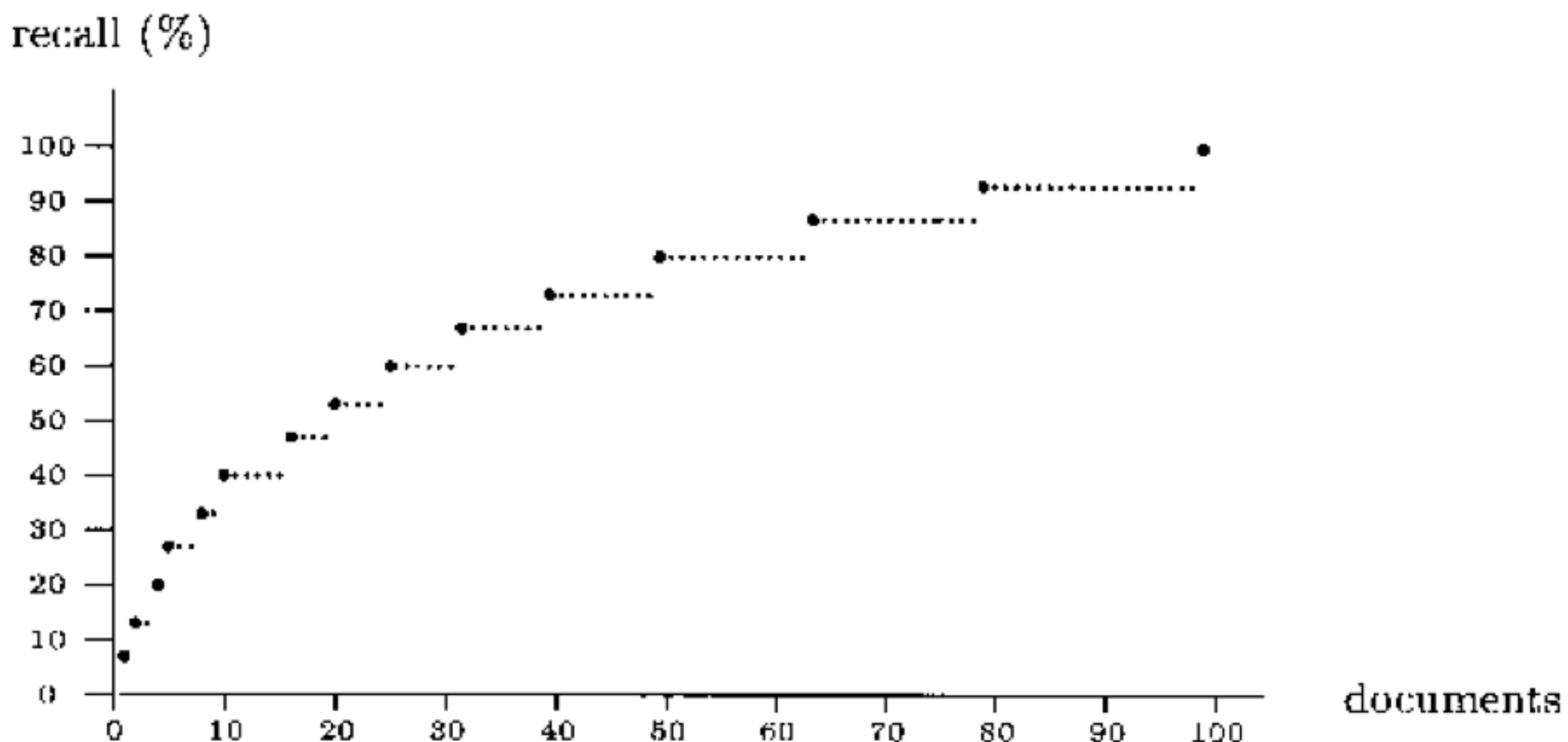
- Calculate precision and recall after each document retrieved

- Retrieve D1
- Have D1
- Retrieve D2
- Have D1, D2
- Retrieve D3
- Now have D1, D2, D3

	Ref?	NRel	Rec	Prec
1	1	1	0.20	1.00
2	1	2	0.40	1.00
3	0	2	0.40	0.67
4	1	3	0.60	0.75
5	0	3	0.60	0.60
6	0	3	0.60	0.50
7	0	3	0.60	0.43
8	0	3	0.60	0.38
9	0	3	0.60	0.33
10	0	3	0.60	0.30
11	0	3	0.60	0.27
12	0	3	0.60	0.25
13	0	3	0.60	0.23
14	0	3	0.60	0.21
15	1	4	0.80	0.27
16	0	4	0.80	0.25
17	0	4	0.80	0.24
18	0	4	0.80	0.22
19	0	4	0.80	0.21
20	0	4	0.80	0.20
21	0	4	0.80	0.19
22	0	4	0.80	0.18
23	0	4	0.80	0.17
24	0	4	0.80	0.17
25	1	5	1.00	0.20

# RECALL PLOT

Recall when more and more documents are retrieved.  
Why this shape?



# SUMMARY

- Relevant vs. Retrieved Documents
- Precision vs. Recall

# REFERENCES

- Christopher D. Manning and Prabhakar Raghavan, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

**THANK YOU !!!**