



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Course Introduction – Part 1

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

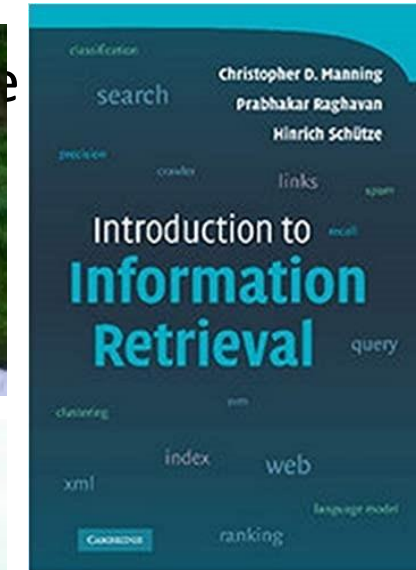
Course Introduction –Part 1

Bhaskarjyoti Das

Department of Computer Science Engineering

Text Book Followed in This Course

1. Slides for this course are adapted from the book slides available at the [Stanford Web site of this book](#)
2. **Christopher Manning** is Thomas M. Siebel Professor in Machine Learning; Professor of Linguistics and of Computer Science ; Director , Stanford Artificial Intelligence Laboratory (SAIL)
3. **Prabhakar Raghavan** is currently head of search at Google, previously head of Yahoo! Research and a consulting professor at Stanford University
4. **Hinrich Schütze** is Chair of Computational Linguistics; Director, Center for Information and Language Processing at LMU, Germany



1. Pre-requisites

- The first four units require **UE18CS251 – Design and Analysis of Algorithm**
- The fifth unit requires **UE18CS303 – Machine Intelligence**

2. Tools and languages used

- For IR applications and Machine learning functionality – Python 3.x, Tensorflow, Scikit Learn and Natural Language Processing libraries in Python
- Apache solr/nutch for search related implementations

- **Search e-mail** for a particular topic
- **Legal information** retrieval by a lawyer
- A customer service agent searches **corporate knowledge bases**
- **Searching the web** for latest information related to Covid-19

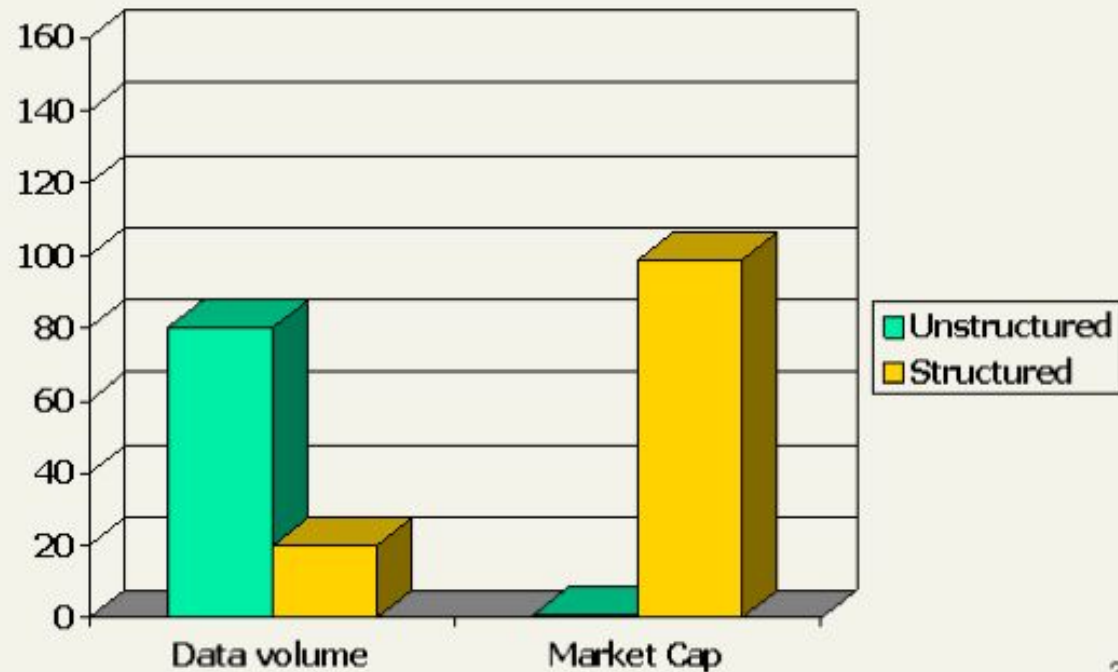
Common Themes in IR Tasks

1. **Information** is from **text documents**
2. Documents are indexed and stored as corpus.
3. The user has an information need.
4. User translates the Information need into query and submits.
5. Relevant documents are retrieved.
6. Retrieval should be efficient
7. The retrieved documents are ranked for relevance before presenting to the user.

How IR is Different From Text Retrieval From RDBMS

Aspect	IR System	RDBMS
Corpus	Unstructured or Semi-structured	Structured
Query	Imprecise, natural language based, keyword based	Precise i.e. SQL for example
Retrieval	Approximate match, similarity based	Exact match
Ranked	Ranked by relevance	Ranked by “order-by clause” ?

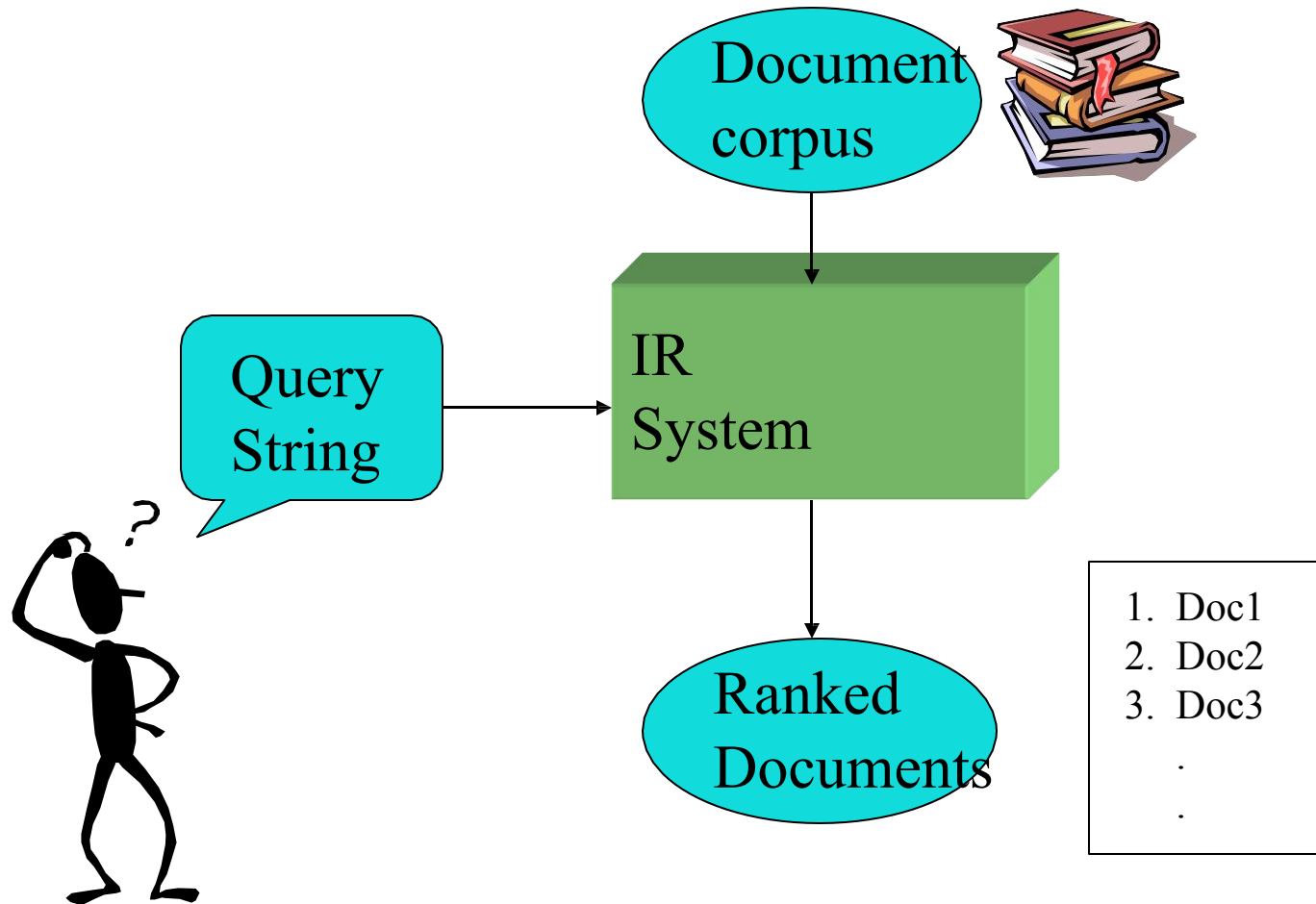
Unstructured (text) vs. structured (database) data in 1996



Unstructured (text) vs. structured (database) data in 2006



In 2020, market cap of companies dealing with IR and unstructured data has grown much more..



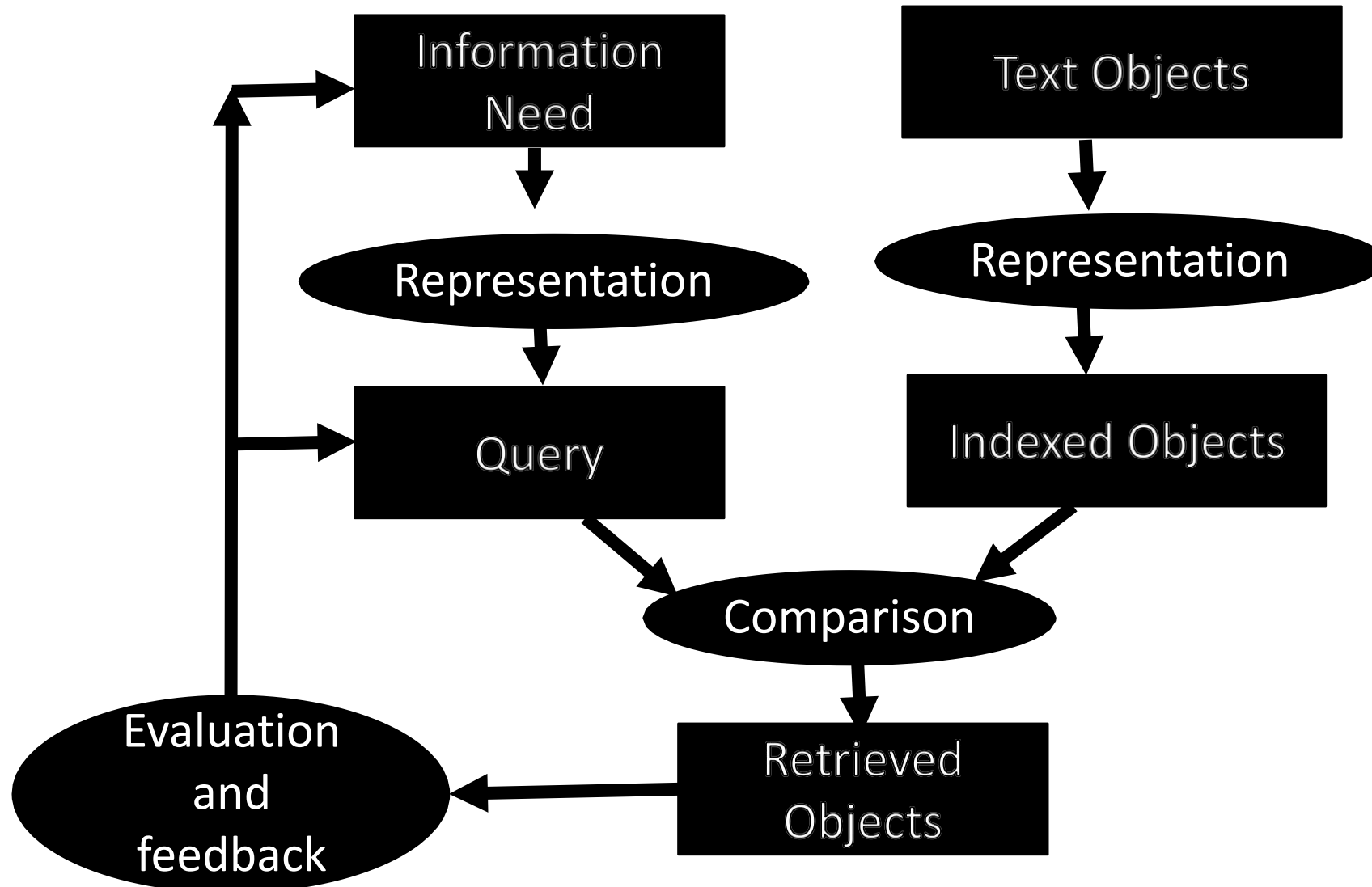
Definition of Information Retrieval

Information retrieval (IR) is **finding** material (**usually documents**) of an **unstructured** nature (usually text) that satisfies an **information need** from within **large collections** (usually stored on computers).

Key Challenge in IR : Relevance Itself is Subjective

- **Relevance is a subjective judgment**
- **Query may not express** the “information need” properly
- **How would you measure relevance** if the “relevance criteria” is subjective ?

Information Retrieval Process – a Refined View



1. **Vector Space model of both** documents and query
2. **Storing the corpus**
 - a. **Operations on text** to form **index words** (terms).
 - b. Index words stored in **dictionary**
 - c. Use **an inverted index** to go from word to document and **retrieve matching documents**.
 - d. **Index compression** so that storing takes less space.
3. **Ranking** based on scoring mechanisms

1. **Evaluation schemes** for ranked retrievals
 - **Precision** – relevant retrieved/total retrieved
 - **Recall** – relevant retrieved / relevant exist
 - **Alternative methods** of evaluation
2. **If the relevance score** is not acceptable, refine the query
 - **Query Expansion** using thesaurus
 - **Query Transformation** using Relevance feedback
3. An introduction to **alternative IR models (XML based, Probabilistic)**



THANK YOU

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Boolean Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

A Simple IR Example

Bhaskarjyoti Das

Department of Computer Science Engineering

An Example IR problem

1. **Document corpus** :
collected works (37 plays \$hakespeare's
2. **Information need** : Which plays of Shakespeare contain the words **Brutus and Caesar** but **not Calpurnia** ?



Term Document Incidence matrix

- **Naïve approach (linear scan)** : One could grep all of Shakespeare's plays for **BRUTUS** and **CAESAR**, then strip out lines containing **CALPURNIA**
- **Why is “grep” not the solution?**
 - **Slow** (for large collections) though it may work for Shakespeare's works
 - grep is **line-oriented**, IR is document-oriented
 - “NOT CALPURNIA” is **non-trivial**
 - Other queries (find the word ROMANS near COUNTRYMAN) **not feasible**

Better Approach

1. **Pre-process** the corpus in advance
2. **Organize the information about the occurrence of words** in a way that **speeds up** query processing
3. For each document, **record** which terms appear in it

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Term Document Incidence Matrix

Bhaskarjyoti Das

Department of Computer Science Engineering

Term Document Incidence Matrix

Anthony and Cleopatra Julius Caesar The Tempest Hamlet Othello Macbeth ...

ANTHONY	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
...						

Entry is 1 if term occurs. Entry is 0 if term does not occur. So we have a **0/1 vector for each term. We call this an Incidence Vector.**

Example: **CALPURNIA** occurs in *Julius Caesar*. Does not occur in *The tempest*.

- To answer the query **BRUTUS AND CAESAR AND NOT CALPURNIA**
 - **Take the vectors** for BRUTUS, CAESAR AND NOT CALPURNIA
 - **COMPLEMENT** the vector of CALPURNIA
 - Do a **(bitwise) AND** on the three vectors
 - 110100 (BRUTUS) AND 110111 (CAESAR) AND 101111 (NOT CALPURNIA) = 100100

Brutus and Caesar and not Calpurnia

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth ...
--	-----------------------------	------------------	----------------	--------	---------	----------------

ANTHONY	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
...						
result:	1	0	0	1	0	0

- **Antony and Cleopatra, Act III, Scene ii**

Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus, When
Antony found Julius **Caesar** dead,
He cried almost to roaring; and he wept When at
Philippi he found **Brutus** slain.



- **Hamlet, Act III, Scene ii**

Lord Polonius: I did enact Julius **Caesar** I was killed i' the
Capitol; **Brutus** killed me.

Boolean Retrieval

- **Queries are Boolean expressions**, e.g., CAESAR AND BRUTUS
- The search engine **returns all documents that exactly satisfy the Boolean expression.**
- The Boolean model is arguably the **simplest model** to base an information retrieval system on.
- **Does Google use the Boolean model?** The search engine doesn't process the "and" and "or" as binary operator but processes them just as search tokens

Document corpus takes lot of disk space!

- Consider $N = 10^6$ documents (1 million)
- Each document is of 10^3 tokens (conservative estimate)
- Total corpus has 10^9 tokens (note the word **token i.e. a bunch of successive characters**)
- On average **6 bytes per token** : Size of corpus is about $6 \times 10^9 = 6$ GB
- A corpus of 1 million documents is common. Any web corpus will be a lot bigger !

Incidence Matrix of the Corpus Requires Lot of Memory

- Assume there are **500,000 distinct terms** in the collection
- *Matrix* = $500,000 \times 10^6 = \text{0.5 trillion 0s and 1s.}$
- That is around **500 GB** assuming 1 byte/cell. Can you load this into your computer's working memory ?

Incidence Matrix is Very Sparse

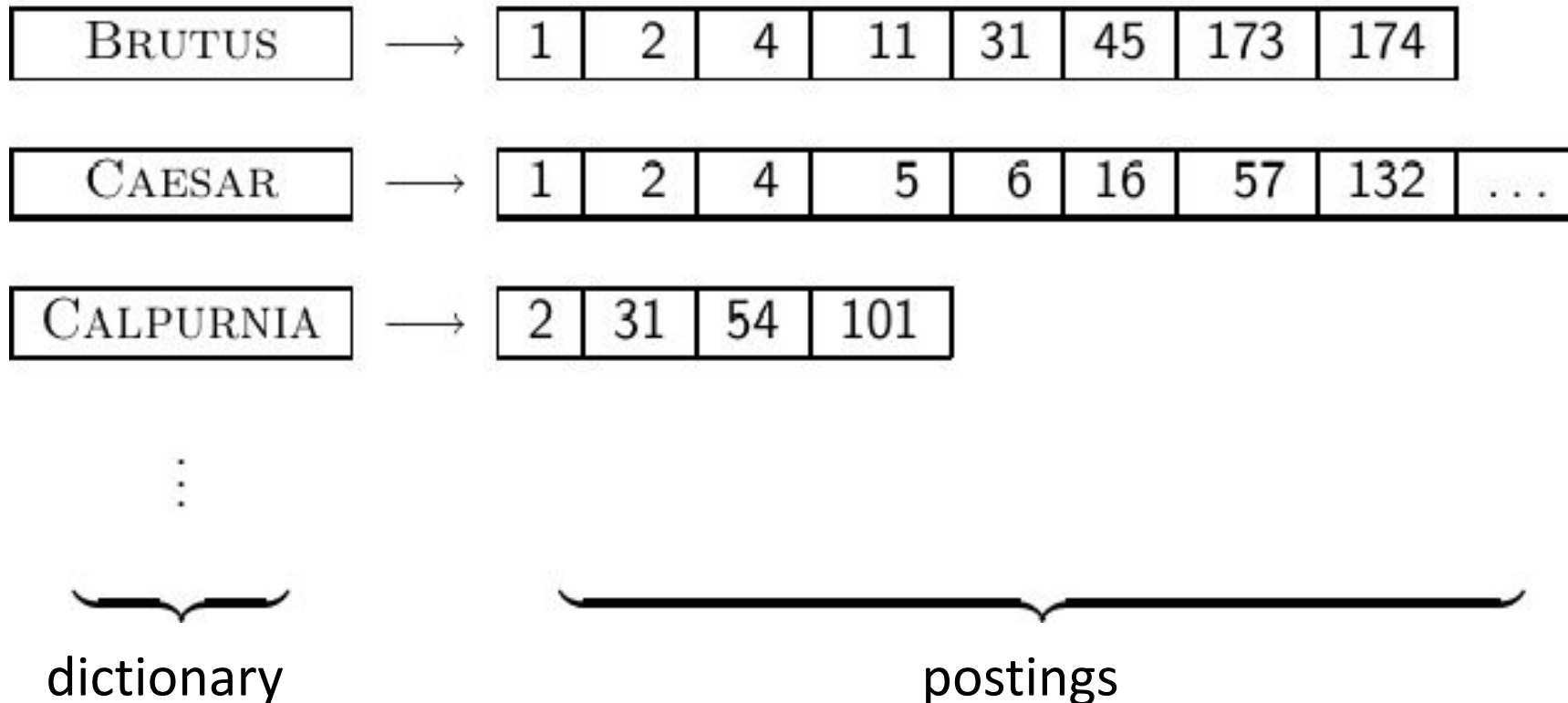
- *Incidence matrix* = $500,000 \times 10^6 = 0.5$ trillion 0s and 1s.
- **Maximum** possible no of 1 in corpus = (#documents)*(#words in each document) = 1 million * 1000 = 1 billion
- So the **matrix has no more than** one billion 1s (around 0.2 %)
- **At least 99.8% of the cells are 0**
 - Matrix is extremely sparse.
- What is a better representation?
 - We only **record the 1s or we simply record the docID.**

Boolean Retrieval: Information Lost

1. Ignoring **how many times** a given word appears in a given document
2. Ignoring **the positions** where a given word appears

Track Only Non-zero Positions of the Incidence Matrix

For each term t , we store a list of all documents that contain t .





THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Boolean Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Inverted Index

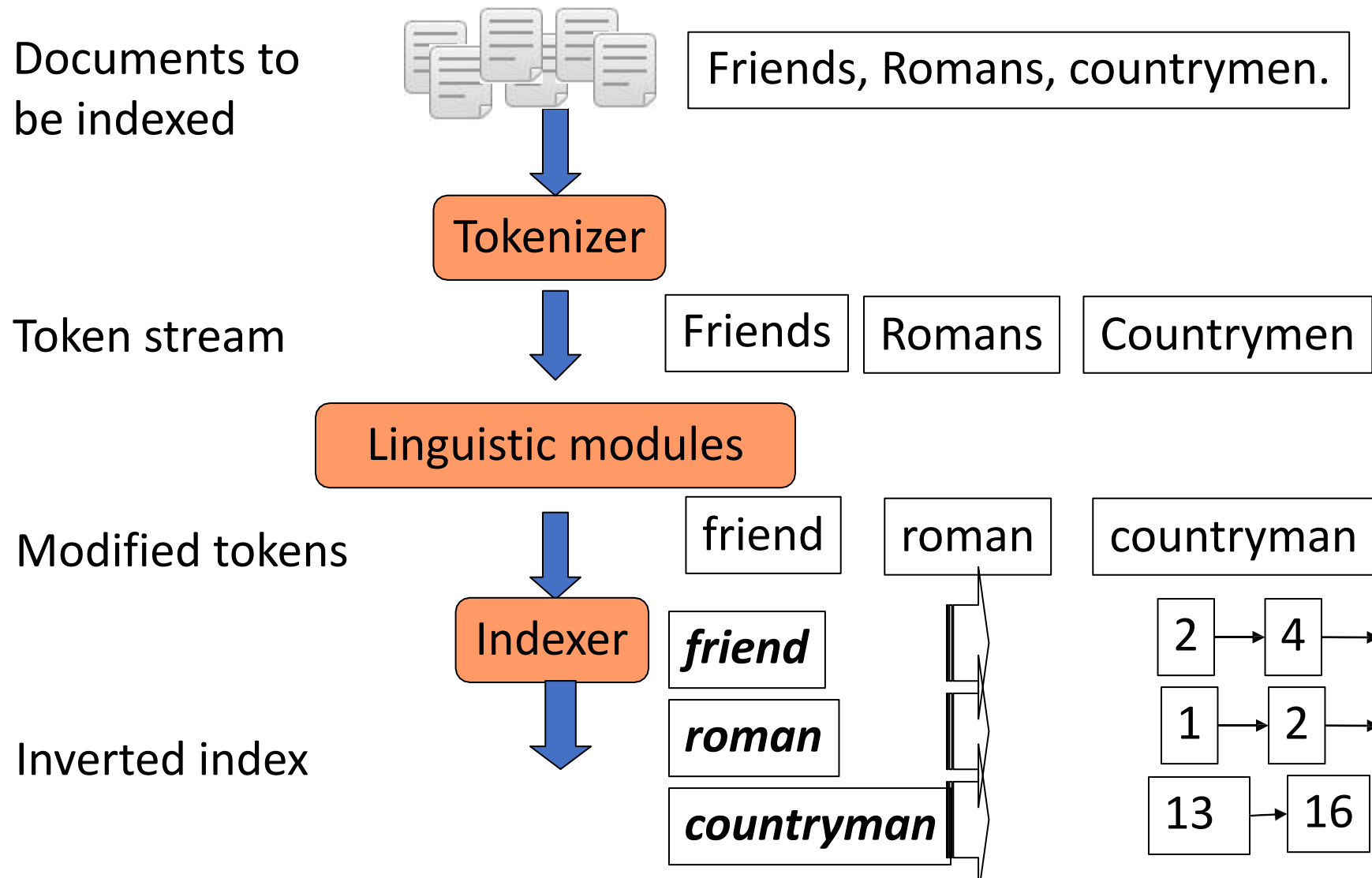
Bhaskarjyoti Das

Department of Computer Science Engineering

Recap of last session : Brutus and Caesar and not Calpurnia

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth ...
--	-----------------------------	------------------	----------------	--------	---------	----------------

ANTHONY	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
...						
result:	1	0	0	1	0	0



Indexer Step : Token Sequence

- Sequence of (Term, Document ID) pairs.

Doc 1

Doc 2



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

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

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

Indexer Step : Sort

- Sort by terms
 - And then docID

Core indexing step

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



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

Indexer Step : Dictionary and Posting

- 1. Multiple term entries are merged.
- 2. Doc. frequency information is added.
- 3. Split into Dictionary and Postings

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

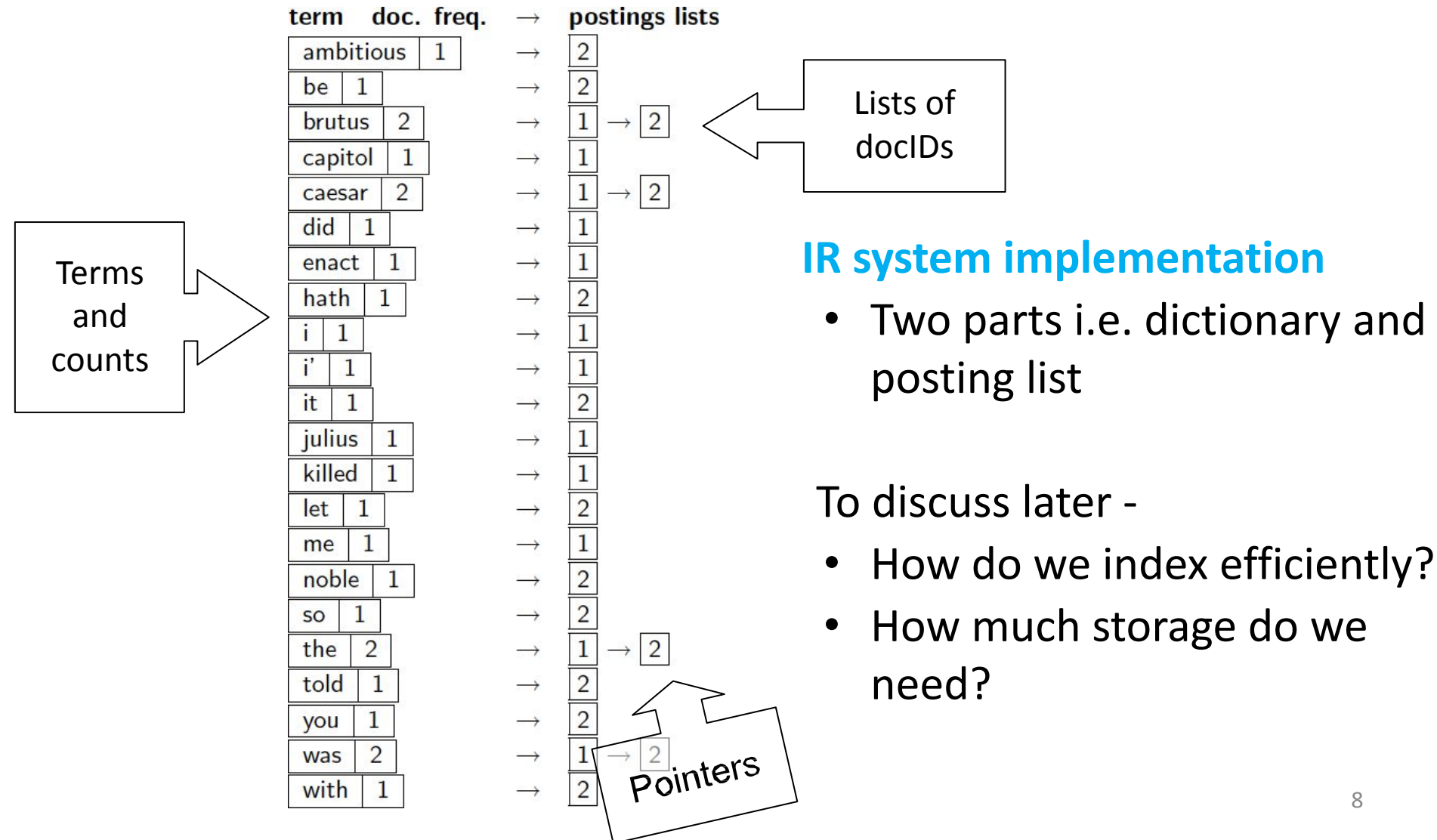


term	doc. fre
ambitious	1
be	1
brutus	2
capitol	1
caesar	2
did	1
enact	1
hath	1
i	1
i'	1
it	1
julius	1
killed	1
let	1
me	1
noble	1
so	1
the	2
told	1
you	1
was	2
with	1

its
→ 2
→ 1 → 2
→ 1
→ 1 → 2
→ 1
→ 1
→ 2
→ 1
→ 1
→ 2
→ 1
→ 1
→ 2
→ 1
→ 2
→ 1 → 2
→ 2
→ 2
→ 1 → 2
→ 2

Why frequency?
Will discuss
later.

Final Inverted Index : Dictionary and Posting Lists



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Boolean Query Processing With An Inverted Index

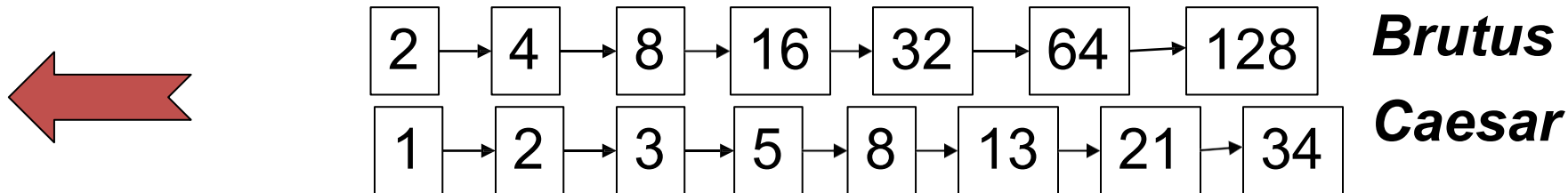
Bhaskarjyoti Das

Department of Computer Science Engineering

Query Processing : AND

- Consider processing the query: *Brutus AND Caesar*

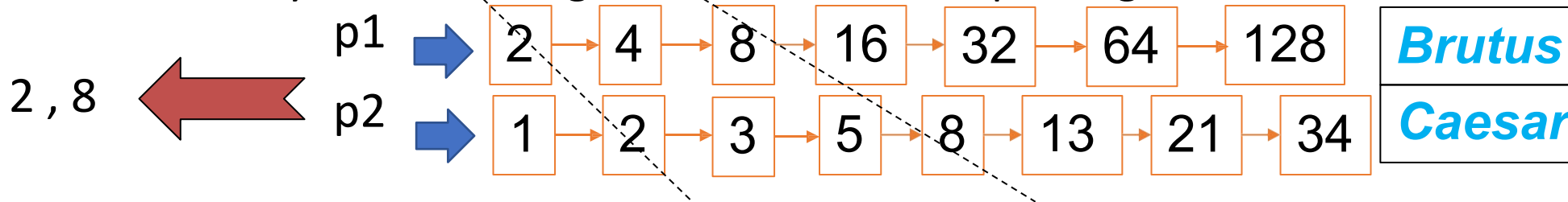
- Locate *Brutus* in the Dictionary;
 - Retrieve its postings.
- Locate *Caesar* in the Dictionary;
 - Retrieve its postings.
- “Merge” the two postings (intersect the document sets):



The MERGE – BRUTUS AND CEASAR

- Walk through the two postings simultaneously, in **time linear** in the total number of postings entries
- Walking in a way such that **each pointer is trying to keep up with the value that the other pointer is pointing to...**

- Essentially we are taking intersection of two posting lists



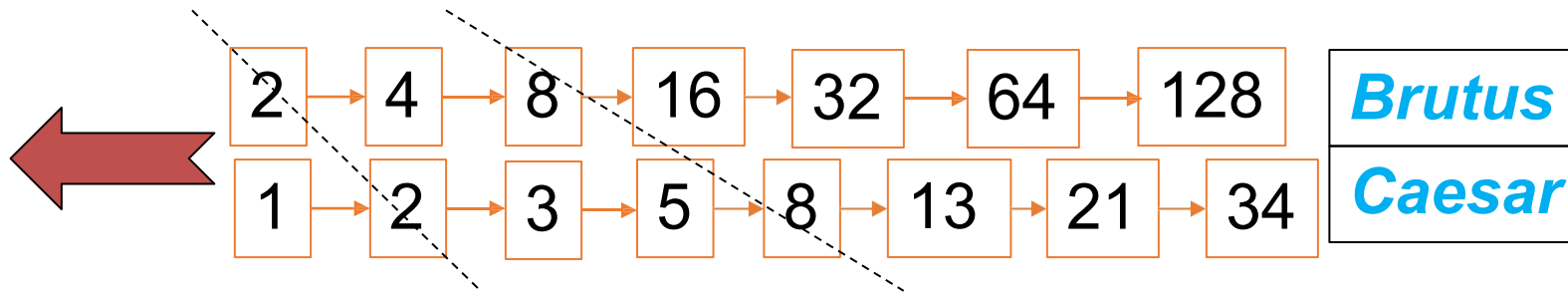
- If the list lengths are x and y , the merge takes $O(x+y)$ operations as we need to make a single pass to each.
- **Crucial: postings sorted by docID.**

Intersecting Two Posting Lists : MERGE Algorithm

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

The Algorithm For BRUTUS OR CEASAR ?

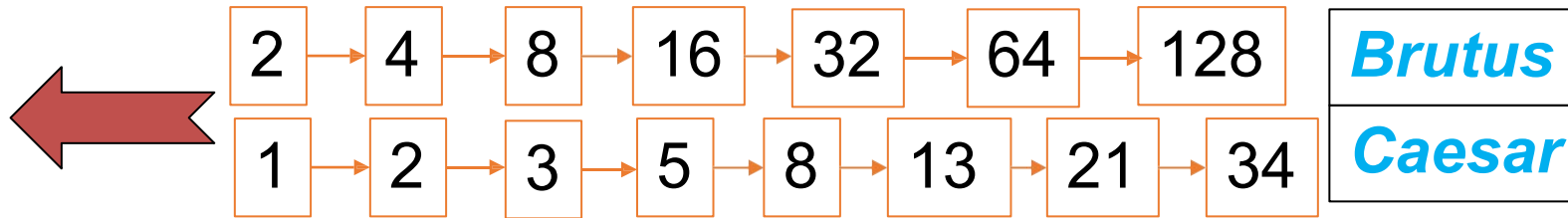
- Instead of taking intersection of two posting lists, we will add every entry of each posting list to our answer



- In the algorithm,
 - when $\text{docID}(p1) < \text{docID}(p2)$, while incrementing pointers, we also add the entry to the answer
 - When completing the loop (as one pointer has reached end of list), we don't forget to add the remaining entries of the other list to the answer
 - The OR operation can also be done in linear time

How Would We Do NOT BRUTUS ?

- We take the BRUTUS posting list first ..



- As we traverse the posting list, **whenever we skip entry** (example - 4 after 2), we add to the answer all the numbers that are skipped (in this case 3)

Can NOT operation be done in the linear time ?

- If the Corpus size C and BRUTUS posting list is B , then NOT B size is $|C-B|$
- If $|C| \gg |B|$, it may not be done in linear time !



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Boolean Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

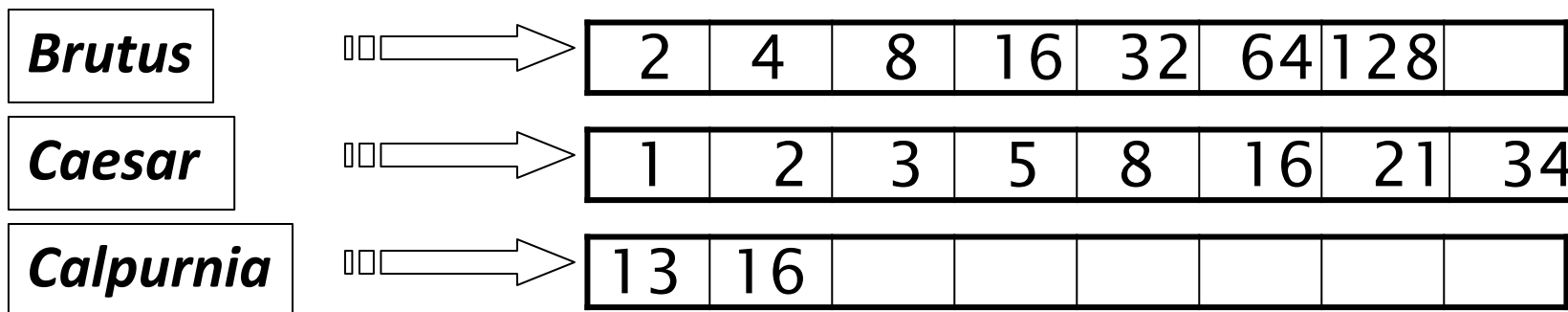
Boolean Query Optimization

Bhaskarjyoti Das

Department of Computer Science Engineering

Query Optimization

- What is the best order for query processing **doing least amount of work ?**
- Consider a query that is an **AND of n terms**.
- For each of the n terms, get its postings, then **AND** them together.
- **More than one way to proceed. Which is the best ?**



Query: *Brutus AND Calpurnia AND Caesar*

What is the Optimization Strategy For AND?

- Process in order of increasing frequency:
 - *start with smallest set* (length of the answer bounded by that), then keep cutting further.

This is why we kept
document freq. in dictionary

Brutus	⇒	2	4	8	16	32	64	128	
Caesar	⇒	1	2	3	5	8	16	21	34
Calpurnia	⇒	13	16						

Execute the query as (**Calpurnia** AND **Brutus**) AND **Caesar**.

What is the Optimization Strategy For OR ?

Example -

(*madding* OR *crowd*) AND (*ignoble* OR *strife*) AND (*noble* OR *great*)

- Get doc. freq.'s for all terms.
- **Estimate the size of each OR** by the sum of its doc. freq.'s (conservative). Why ?
 - $A \cup B = |A| + |B| - A \cap B$
- Process in **increasing order of OR sizes**.

Exercise : Recommend a Query Processing Order

*(tangerine OR trees) AND
(marmalade OR skies) AND
(kaleidoscope OR eyes)*

- Which two terms should we process first?

Term	Freq	
eyes	213312	Smallest
kaleidoscope	87009	300321
marmalade	107913	Largest
skies	271658	
tangerine	46653	2 nd smallest
trees	316812	363465



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Boolean Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

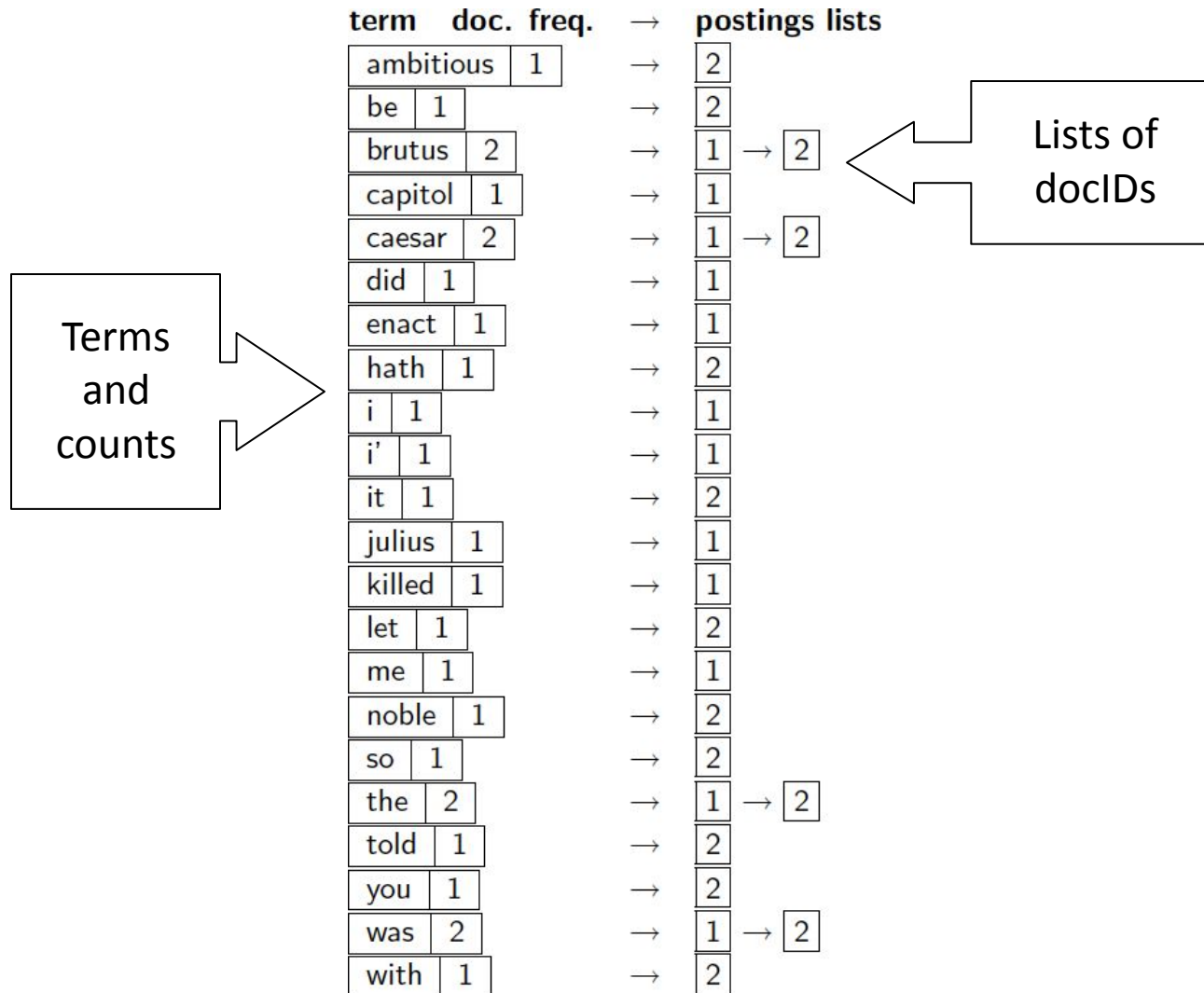
ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Boolean Retrieval Example and Shortcomings

Bhaskarjyoti Das

Department of Computer Science Engineering

Recap of Last Class : Final Inverted Index Design



Recap Of The Last Class



- We started with the **steps of inverted index construction and splitting it into dictionary and posting**
- Then assuming we have the inverted index is in place, we looked at **Boolean Query** - the merge sort like **INTERSECT** algorithm for **AND, OR** and **NOT** cases.

Boolean Queries : Strength

The **Boolean retrieval model** asks a **query** that is a **Boolean expression**:

- Boolean Queries are queries **using AND, OR and NOT** to join query terms
- Views each document as a **set of words**
- Is **precise**: document matches condition or not.
- Perhaps the **simplest model** to build an IR system on

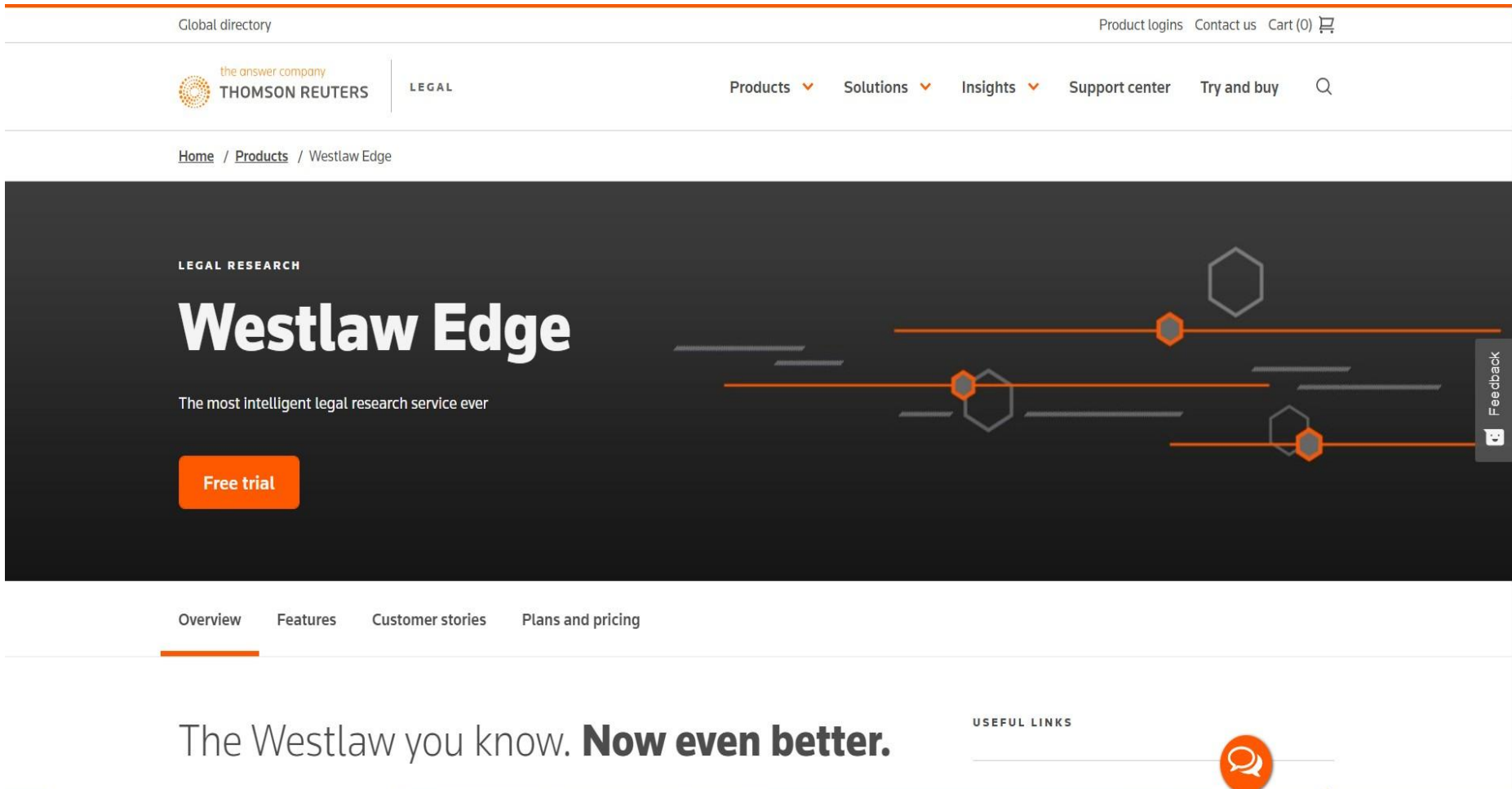
Boolean Retrieval Model is still used

- Primary commercial retrieval tool for 3 decades.
- Many search systems you still use are Boolean:
 - Email, library catalog, Mac OS X Spotlight
- Many Legal Information systems still uses Boolean Retrieval as the lawyers want specific information for the cases (recall the difference between search and browse ?)

ALGORITHMS FOR INFORMATION RETRIEVAL & INTELLIGENT WEB

Example : WestLaw

Started originally in 1975, Boolean Query is still used !!



The screenshot displays the Westlaw Edge website interface. At the top, there is a navigation bar with links for 'Global directory', 'Product logins', 'Contact us', and 'Cart (0)'. Below this, the 'the answer company THOMSON REUTERS' logo is visible on the left, and a menu with 'Products', 'Solutions', 'Insights', 'Support center', and 'Try and buy' is on the right. The main content area features a dark background with the text 'LEGAL RESEARCH' and 'Westlaw Edge' in large white letters. Below this, it says 'The most intelligent legal research service ever' and includes a 'Free trial' button. A decorative graphic of orange lines and hexagons is on the right. At the bottom, there are tabs for 'Overview', 'Features', 'Customer stories', and 'Plans and pricing'. The footer contains the text 'The Westlaw you know. Now even better.' and a 'USEFUL LINKS' section with a chat icon.

Example Boolean Query In WestLaw With Proximity Operators



Information need: *Requirements for disabled people to be able to access a workplace*

Query: **disabl! /p access! /s work-site work-place (employment /3 place)**

Documents having a **word starting with** “disabl!” **in the same paragraph** having **a word starting with** “access!” **in the same sentence** having **either of** “work-site and work-place or employment” **within 3 words of** the word “place” ...

- Note that SPACE is disjunction, not conjunction!
- With **proximity operators**

Example Boolean Query In WestLaw With wild card

Information need: *Information on legal theories involved in preventing legal disclosure of trade secrets by employees formerly employed by a competing company*

Query : “trade secret” /s disclos! /s prevent /s employ! /s

“Exact phrase”, ‘/s’ : same sentence, ‘!’ is wild card

- Long, precise query

Limitations of Boolean Retrieval On Our Design Of Inverted Index

1. Not **tolerant** to **spelling mistake**

2. Can it handle **Phrase Search** ?

- “Stanford University” : University must appear right after Stanford
- Search for “Microsoft’s Bill Gates” in the same sentence having
- Boolean Query based Retrieval can’t handle the above as so far **we are not keeping track of position of the terms**

Limitations of Boolean Retrieval On Our Design Of Inverted Index

3. Also current implementation of Boolean Retrieval **does not** consider **“term frequency”** that can be derived if position information is kept !
4. It cannot handle **Ranking** ?
 - WestLaw returns legal documents **in the order by date**.
 - However **ranking implies relevance**
 - Boolean Retrieval **does not** implement ranking



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Term Vocabulary and Posting Lists

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Text, Tokens And Terms

Bhaskarjyoti Das

Department of Computer Science Engineering

- So far we talked about “**words**” and “**documents**” in the **Incidence Matrix** design
- In IR, documents are represented by **document ID**
- **Words** are not the entities that are used by an IR system
- A **token** is an **instance of a sequence of characters**
- Each such **token** is now a **candidate for an index entry**, after further processing/**normalization** becomes a **term**
- After pre-processing, **terms** are the entities that IR system uses

Tokenization

- Input: “*Friends, Romans and Countrymen*”

- Output: Tokens
 - *Friends*
 - *Romans*
 - *Countrymen*

- Issues in tokenization:

- *Finland's capital* →
Finland AND *s*? *Finlands*? *Finland's*?
- *Hewlett-Packard* → *Hewlett* and *Packard* as two tokens?
 - *state-of-the-art*: break up hyphenated sequence.
- *San Francisco*: one token or two?
 - How do you decide it is one token?

Tokenization Issues : Numbers

- *3/20/91 20/3/91 Mar. 12, 1991*
- *55 B.C.*
- *B-52*
- *My PGP key is 324a3df234cb23e*
- *(800) 234-2333*
 - Often have embedded spaces
 - Older IR systems may not index numbers
 - But often very useful: think about things like looking up error codes/stacktraces on the web
 - Will often index “meta-data” separately
 - Creation date, format, etc.

Tokenization Issues :Language

- **French**
 - *L'ensemble* → one token or two?
 - *L ? L' ? Le ?*
- **German noun compounds** are not segmented
 - *Lebensversicherungsgesellschaftsangestellter*
 - 'life insurance company employee'
- **Chinese and Japanese** have no spaces between words:
莎拉波娃现在居住在美国东南部的佛罗里达。
Not always guaranteed a unique tokenization
- **Arabic (or Hebrew)** is basically written right to left, but with certain items like numbers written left to right

Tokenization Issues : Stop Words

- With a stop list, you exclude from the dictionary entirely the commonest words.
 - They have little semantic content: *the, a, and, to, be*
 - There are a lot of them: ~30% of postings for top 30 words

What Is Tokenization Then ?

- Use a Natural Language Processing **library to convert a collection of text into tokens** (a collection of adjacent characters) while taking care of the issues listed earlier

- A term is a (normalized) word type, which is an entry in our IR system dictionary
- We may need to “normalize” words in indexed text as well as query words into the same form
 - We want to match **U.S.A.** and **USA**
- Equivalence classing
 - deleting periods to form a term
 - **U.S.A., USA --- USA**
 - deleting hyphens to form a term
 - **anti-discriminatory, antidiscriminatory ---- antidiscriminatory**

Case Folding – Another Technique Of Normalization

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

Lemmatization – Another Technique Of Normalization

- Reduce **inflectional/variant forms to base form**
- E.g.,
 - *am, are, is* → *be*
 - *car, cars, car's, cars'* → *car*
- *the boy's cars are different colors* → *the boy car be different color*
- Lemmatization implies **doing “proper” reduction to dictionary headword form by utilizing Lexical Knowledge**

Stemming – Another Technique Of Normalization

- Reduce terms to their “roots” before indexing
- “Stemming” suggests crude affix chopping
 - language dependent
 - e.g., *automate(s)*, *automatic*, *automation* all reduced to *automat*.

for example compressed and compression are both accepted as equivalent to compress.



for exampl compress and compress ar both accept as equival to compress

Does Stemmer Work Very Well ?

- **Porter Stemmer** - commonest algorithm for stemming English
 - **Rule based and Fast**
 - Results suggest it's at least as good as other stemming options
 - Not perfect.
 - **For English:** very **mixed results.**
 - Helps recall for some queries but harms precision on others
- **Other stemmers exist** – Lovins and Snowball
- **Full morphological analysis (lemmatization)**
 - At most modest benefits for retrieval

- Do we handle synonyms and homonyms?
 - by hand-constructed **equivalence classes**
 - *car = automobile* *color = colour*
 - We can **rewrite to form equivalence-class terms**
 - When the document contains *automobile*, index it under *car-automobile* (and vice-versa)
 - we can **expand a query**
 - When the query contains *automobile*, look under *car* as well
- What about spelling mistakes?
 - One approach is **Soundex Algorithm**, which forms equivalence classes of words based on phonetic heuristics
 - Another approach is **Edit Distance Algorithm**

Summary



- Corpus is made up of text
- Text is pre-processed to **tokens** first.
- The tokens are then **normalized to terms** that are input to IR system.
- We discussed various **text pre-processing and normalization strategies**



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Term Vocabulary and Posting List

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Positional Indexes

Bhaskarjyoti Das

Department of Computer Science Engineering

Recap Of Last Class And Today's Plan



1. Discussed **optimization** by means of **Skip Pointers**
2. **Phrase Query** is not offered by Boolean Query.
3. **Biword indexing** and **extended biwordindexing** are methods of implementation of phrase query.
4. Finally, **biword indexing is not a great strategy** as it leads to index blow up !!

Today we will talk about **Positional Index** and revisit **Inverted Index (Dictionary and Posting list) Implementation** as a build up for discussing implementation of Wild Card Query

- **A non-positional** index: each posting in this list is just a docID
- **A positional index**: Postings lists in a each posting is a **docID** and **a list** of positions. Every element in posting list is a structure.
 - **< Term, no of docs containing term;**
 doc1 : position1, position 2,... ,position n; doc1 :
 position1, position 2,... ,position n;

 etc. >

Positional Index : For A Phrase Query

- Example – we are trying to find the phrase : **To be**
- Extract inverted index entry for each distinct term : **to, be,**
- Merge their doc position list to enumerate all positions
- To : 1<...>,2<...>,4<...>,5<...>,7<...>
- Be : 1<...>,4<...>,5<...>
- **First** we find the common doc ids by using Merge Algorithm recursively at the document level
- **Next**, we need to deal with **more than just equality of docID** i.e. we check whether **be** is appearing after **to**

Positional Index : Another Example For Phrase Query

Query: “*University Of Minnesota*”

1. We want all documents with *University* and *Minnesota* such that *Minnesota* appears 1 word after *University*
2. Find Posting lists of *University* and *Minnesota*
3. Use the algorithm we already know to find docID where both appear
4. Now check for *Minnesota* appearing 1 word after *University*

Positional Index : Example of The Advantage It Offers

Query: “to₁ **be**₂ or₃ not₄ to₅ **be**₆”

TO, 993427:

1: <7, 18, 33, 72, 86, 231>;

2: <1, 17, 74, 222, 255>;

4: <8, 16, 190, 429, 433>;

5: <363, 367>;

7: <13, 23, 191>; ... >

BE, 178239:

1: <17, 25>;

4: <17, 191, 291, 430, 434>;

5: <14, 19, 101>; ... >

Which of the docs 1,2,4,5 could contain this phrase ?

A possible first check:

between the 2 “be” , 3 words must appear !

Document 4 is a possible match! **We need to extend this logic for other pairs ...**

We can use the same technique for proximity query !

Another Example For Proximity Search

- Example: **employment /4 place**
- Find all documents that contain EMPLOYMENT and PLACE within 4 words of each other.
 - **Employment agencies that place** *healthcare workers are seeing growth* is a hit.
 - **Employment agencies that have learned to adapt now place** *healthcare workers* is not a hit.
- **Level 1** : posting lists of **employment** and **place** and use INTERSECT algorithm to find docID where both appear
 - **Level 2** : Run the check on the **position constraint**



Recall WestLaw Proximity Query



- Recall what we saw in WestLaw !
- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
 - Again, here, / k means “within k words of”.
- **Positional indexes can be used for such queries**; biword indexes cannot.

How Do We Implement Positional Index?

1. Essentially **reuse the previously used INTERSECT algorithm** to find a **docID** where both terms are appearing
2. **Retrieve the position list of the terms** from the matching docID
3. Check if one term is appearing in a **window of size K as defined by the proximity query**
4. **Twice we should use pointers to traverse a list-**
appropriately modify INTERSECT algorithm

Modified Intersect For Proximity Constraint K

```
POSITIONALINTERSECT( $p_1, p_2, k$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4      then  $I \leftarrow \langle \rangle$ 
5           $pp_1 \leftarrow \text{positions}(p_1)$ 
6           $pp_2 \leftarrow \text{positions}(p_2)$ 
7          while  $pp_1 \neq \text{NIL}$ 
8              do while  $pp_2 \neq \text{NIL}$ 
9                  do if  $|\text{pos}(pp_1) - \text{pos}(pp_2)| \leq k$ 
10                     then  $\text{ADD}(I, \text{pos}(pp_2))$ 
11                     else if  $\text{pos}(pp_2) > \text{pos}(pp_1)$ 
12                         then break
13                      $pp_2 \leftarrow \text{next}(pp_2)$ 
14                     while  $I \neq \langle \rangle$  and  $|I[0] - \text{pos}(pp_1)| > k$ 
15                         do  $\text{DELETE}(I[0])$ 
16                     for each  $ps \in I$ 
17                         do  $\text{ADD}(answer, \langle \text{docID}(p_1), \text{pos}(pp_1), ps \rangle)$ 
18                      $pp_1 \leftarrow \text{next}(pp_1)$ 
19              $p_1 \leftarrow \text{next}(p_1)$ 
20              $p_2 \leftarrow \text{next}(p_2)$ 
21         else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
22             then  $p_1 \leftarrow \text{next}(p_1)$ 
23             else  $p_2 \leftarrow \text{next}(p_2)$ 
24 return  $answer$ 
```

Implementation
of the window of
size K for the case
where common
docID is found

Positional Index Is Popular

1. A positional index *expands postings storage substantially* even though indices can be compressed
2. Nevertheless, a positional index is **a standard commonly followed** because of the power and usefulness of phrase and proximity queries ...
3. May be **used explicitly or implicitly** in a ranking retrieval system.

Positional Index Size

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

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

Rule Of Thumb

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

Combination Scheme

- **Biword indexes and positional indexes** can be profitably combined.
- Many biwords are **extremely frequent**: Michael Jackson, Britney Spears etc
 - For these **biwords, increased speed** compared to positional postings intersection is substantial.
- **Combination scheme:**
 - Include **frequent biwords as vocabulary terms** in the index.
 - Do all other phrases **by positional intersection**.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Term Vocabulary and Posting List

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Positional Indexes

Bhaskarjyoti Das
Department of Computer Science Engineering

Posting list – array vs. linked list ?

- Can be **an array or linked list of a structure ?**
- **Array of fixed size :** Every posting list is same size and will lead to lot of waste
- **Array of variable size:** Addresses the above wastage!
- **Linked List vs. array :**
 - If the corpus is static (mostly read action), array may work well (caching due to locality of reference). If corpus is dynamic, may not work well !
 - Pointers will require additional space but makes sense if this index is dynamic

Posting list - big vs. small ?

- If the list is very big as the corpus is big, then **it cannot reside in memory fully** and we need many disk seeks.
 - In this case, the **linked list in a bad option** as disk read time will be more (not contiguous on the disk)
 - In such scenario, it is **better to use array**.

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary Implementation

Bhaskarjyoti Das

Department of Computer Science Engineering

- **Dictionary:** the **data structure** for storing the term vocabulary. We need to access this structure efficiently to access the terms
- For each term, we need to store a couple of additional items:
 - document frequency
 - pointer to postings list . .

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

- Two main classes of data structures for index design : **hash tables and binary search trees**
- Some IR systems use hashes, some use trees. Why ?
 - There are trade offs in advantages (**sorted info. vs. access speed**)...

How Does Hash Table Work ?

1. Each **vocabulary term is hashed** into an **integer** using which we access the data stored
2. Try to **avoid collisions**
3. At query time, do the following: **hash** query term, **resolve** collisions, **locate** entry in fixed-width array

- **Pros: Lookup in a hash is faster** than lookup in a tree.
 - Lookup time is constant.
- **Cons :**
 - no way to find **minor variants** (*resume vs. résumé*)
 - no **prefix search** (all terms starting with *automat* i.e. *automat**)
 - need to **rehash everything periodically** if vocabulary keeps growing

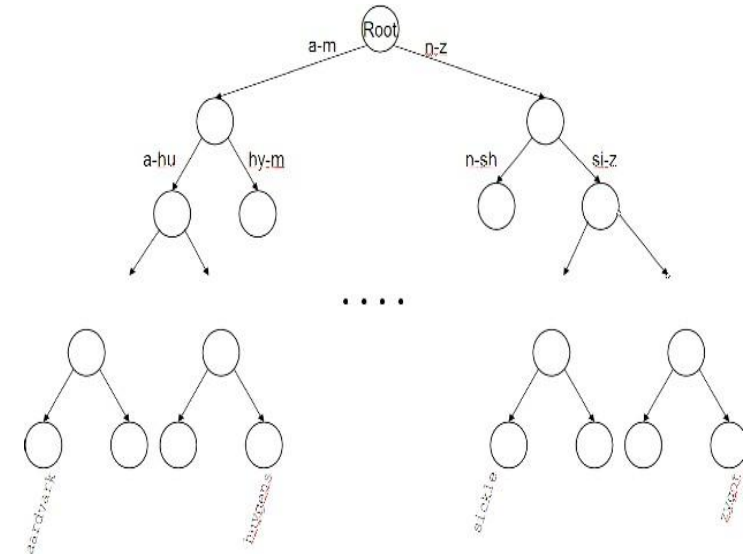
Binary Search Tree

The **left subtree of a node** contains only nodes with **keys lesser** than the **node's key**.

The **right subtree of a node** contains only nodes with **keys greater** than the **node's key**.

The **left and right subtree** each must also be a **binary search tree**.

Unlike Hash table, here **$O(\log(n))$** . We can get all **keys in sorted order** by just doing **In Order Traversal** of BST

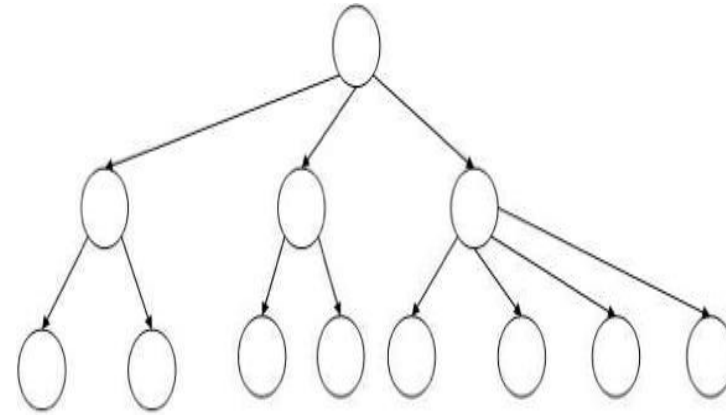


Binary Search Trees Pros And Cons

- **Trees solve the prefix problem** (find all terms starting with *automat*).
- BST Search is **slightly slower than in hashes**: $O(\log M)$, where M is the size of the vocabulary.
 - $O(\log M)$ only holds for **balanced trees**.
 - If we just insert elements **in sorted order**, it may get added like a linked list (it will be still a BST but no more balanced)
- **Rebalancing** binary trees (keeping the height optimal) is **expensive**.

B Tree

- **B-trees** mitigate the **rebalancing problem**. It is a balanced binary search tree. It is one of the balanced BST
- **B-tree definition**: every internal node has a **number of children in the interval $[a, b]$** where a, b are appropriate positive integers, e.g., $[2, 4]$ means internal nodes having 2-4 children.
 - **Leaf nodes at** last level and **internal nodes** having children
- If **b is large**, tree gets **more flattened** and internal gets more children (**more comparison** with many possible paths)
- Strength: it allows you to do **prefix search like BST and solves rebalancing issue**



Criteria For Choosing B-Tree Vs. Hash

Table

Aspect to consider	Binary Search Tree	Hash table
Fixed no of terms	Larger access time	Faster access and is a better option.
Growing no of terms	Insert/removes leaves it in sorted condition and is <u>not an issue as long as tree is balanced</u>	Moving all terms to a new hash table is $O(n)$ operation.
Supporting prefix such xyz*	Supports prefix based query	Not supported
Adding items in a pre-sorted manner	Is an issue for BST that will need rebalancing. <u>Balanced BST(Btree)</u> addresses this issue.	Not an issue.

A balanced BST like **B-Tree is the default option** in dictionary implementation.

Going forward, we will assume that a B-tree based implementation is used.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Term Vocabulary and Posting Lists

Bhaskarjyoti Das

Department of Computer Science Engineering

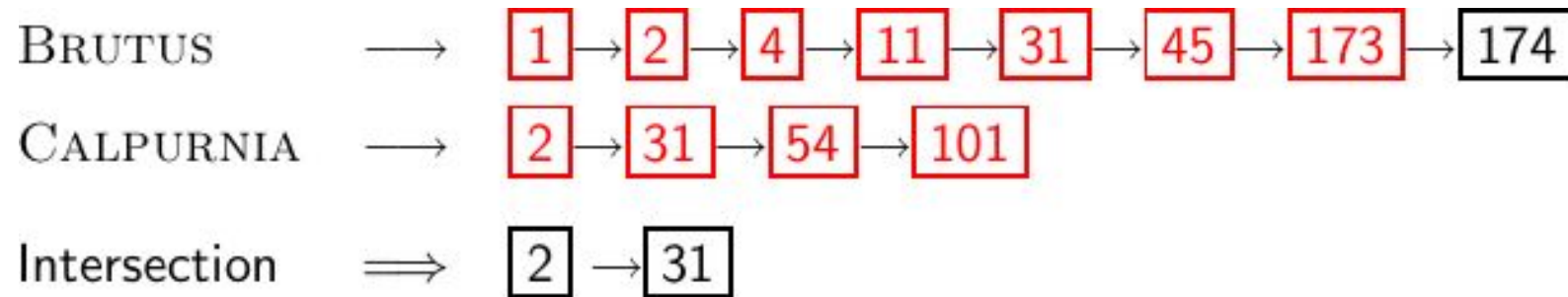
ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Skip Pointer For Faster Postings Merge

Bhaskarjyoti Das

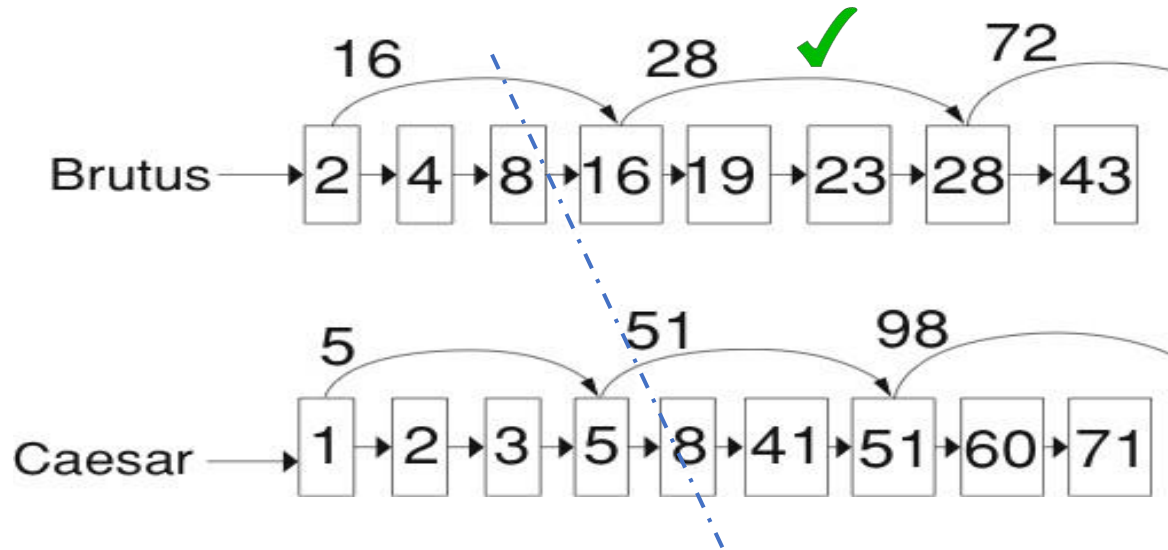
Department of Computer Science Engineering

Recall Basic Intersection Algorithm



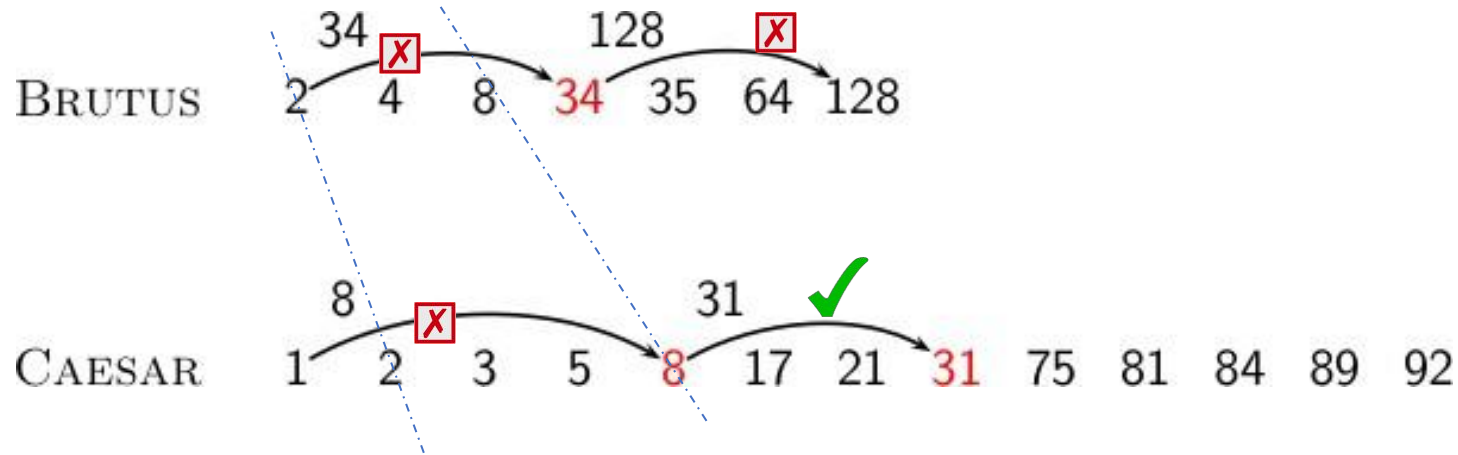
- **Linear in the length** of the postings lists i.e. $O(m+n)$
- **Can we do better?**

Skip Pointer Example



- Suppose both pointers **stepped through the list till 8**.
 - They then increment to **41 and 16**.
- By **incrementing 4 times**, it goes up to 28 which is still less than 41 !
- What has been the utility of **16-> 19-> 23->28** ? None !!
- Could have used the **skip pointer to come to 28 even earlier and save cost ?**

Skip Pointers



- Skip pointers are additional pointers that skip positions on the list
- Generated at index generation time.

Skip Pointers For A More Efficient Merge

- Skip pointers allow us to **skip postings that will not figure in the search results**. This makes postings lists more efficient (saves cost).

Where it helps ?

- Some postings lists contain several million entries – so efficiency can be an issue even if basic intersection is linear.

Intersection With Skip Pointers

Comparison result asks for an increment in pointer! You have a skip pointer available, consider feasibility of skipping!

```
INTERSECTWITHSKIPS( $p_1, p_2$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4      then  $\text{ADD}(answer, \text{docID}(p_1))$ 
5           $p_1 \leftarrow \text{next}(p_1)$ 
6           $p_2 \leftarrow \text{next}(p_2)$ 
7  else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
8      then if  $\text{hasSkip}(p_1)$  and  $(\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))$ 
9          then while  $\text{hasSkip}(p_1)$  and  $(\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))$ 
10             do  $p_1 \leftarrow \text{skip}(p_1)$ 
11             else  $p_1 \leftarrow \text{next}(p_1)$ 
12      else if  $\text{hasSkip}(p_2)$  and  $(\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))$ 
13          then while  $\text{hasSkip}(p_2)$  and  $(\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))$ 
14             do  $p_2 \leftarrow \text{skip}(p_2)$ 
15             else  $p_2 \leftarrow \text{next}(p_2)$ 
16  return  $answer$ 
```

Where Is The Trade Off ?

- **Many skip**: Each skip pointer skips **only a few items**, but we can frequently use it.
 - If you have **too many (small) skips**, you will have lot more comparison to do .
- **Large skips**: Each skip pointer skips **many items**, we can NOT use it very often.
 - If you have **too few (large) skips**, you may not even utilize skip pointers !
- **Tradeoff**: **number of items skipped (= skip length)** vs. **frequency of skip**

Where Do We Place Skips ?

- **Simple heuristic:** for postings list of length L , use **evenly-spaced \sqrt{L} skip pointers**. This ignores the distribution of query terms.
- **Easy if the index is static:** If static, the skip pointers continue to work.
- **Hard if the index is dynamic:** Harder in a dynamic environment because of updates (you may end up deleting skip pointers during update).
- How much do skip pointers help? They **used to help** a lot.
- With today's **fast CPUs, they don't help** that much anymore.

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Phrase Queries Using Biword Indexes

Bhaskarjyoti Das

Department of Computer Science Engineering

- The concept of **phrase query has been popular**.
- About 10% of web queries are phrase queries. Many are **implicit phrase queries** (not using quote)
- We want to answer a query such as **[stanford university]** – as a phrase.
 - Thus “*The inventor Stanford Ovshinsky never went to university*” should not be a match.

Consequence For Inverted Index

- **Consequence for inverted index position without information:** it no longer suffices to store postings lists. docIDs in
- **Two ways of extending** the inverted index for Phrase Query :
 - biword index
 - positional index

- Index **every consecutive pair of terms** in the text as a phrase.
- For example, “**Friends, Romans, Countrymen**” would generate two biwords: “**friends romans**” and “**romans countrymen**”
- Each of these **biwords** is now a **vocabulary term**.
- **Two-word phrases** can now easily be supported. How do we support longer phrase ?

Longer Phrase Queries ?

- Split **Longer Query** into “**Biword Queries**”
- A long phrase like “**STANFORD UNIVERSITY PALO ALTO**” can be represented as multiple Boolean query “**STANFORD UNIVERSITY**” AND “**UNIVERSITY PALO**” AND “**PALO ALTO**”
 - We need to do **post-filtering** of hits to identify subset that actually contains all three bi-word queries i.e. the 4-word phrase.
- There **can be false positives** – a phrase that has all three sub- phrases and but they are NOT consecutive !!

INSIGHT : It turns out that most **queries** are based on **NOUNS**

- Parse each document and perform part-of-speech tagging
- Bucket the terms into (say) **nouns (N)** and **articles/prepositions (X)**

Now deem any string of terms of the form **NX^*N** to be an ***extended biword***

Example: **catcher in the rye**

N X X N

king of Denmark

N X N

Include extended biwords in the **term vocabulary**

Queries are processed accordingly

Biword Indexes: Cons

- False positives, as noted earlier
- Index blowup due to very large term vocabulary
- Biword indexes rarely used



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

Recap



1. In the last class, we discussed implementation options for **Dictionary and Posting Lists**.
2. B-tree is a preferred data structure for Dictionary implementation as it supports **prefix type of query, is always balanced and after any update, remains in sorted condition**.
3. Prefix kind of queries are important as that is what **Wildcard Queries** do.
4. Today we will discuss **methods of supporting wild card queries** assuming we have a B-tree based implementation of dictionary.

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Wild Card Queries

Bhaskarjyoti Das

Department of Computer Science Engineering

Tolerant retrieval: What to do if there is no exact match between query term and document term

- Wildcard queries
- Spelling correction

Trailing Wild Card Queries : * Appears At The End

1. ***mon****: find all docs containing any word beginning with “mon”.
2. Easy with B-tree based implementation : retrieve all terms in range: ***mon ≤ w < moo***
3. We retrieve posting list for all such terms
4. list of document is then result of an OR operation on all such lists.

Leading Wild Card Queries : * Appears At The Beginning

1. ***mon**: find words ending in “mon”: harder
2. Instead of prefix, suffix is specified.
3. Maintain an **additional B-tree** for reverse terms (terms *backwards*) to utilize B-tree prefix query ability.
4. Use **reverse B tree (tree with reverse terms)** to retrieve all terms in range: $nom \leq w < non$
5. Point to note : **we now have two B-tree**
 - a) Standard Inverted Index
 - b) Reverse B tree based index

Exercise

- How can we enumerate all terms meeting the wild-card query ***pro*cent*** ?
 - How can we enumerate all documents containing such terms ?
1. Evaluate pro^* using normal B tree
 2. Evaluate $*cent$ using the reverse B tree
 3. Intersect (AND) these two list of terms
 4. Then retrieve documents that contain any of these terms
(OR all their posting lists)

A Compound Wild Card Query

Get all the terms : **se*ate AND fil*er**

1. **se*ate** : we have an enumeration of all terms in the dictionary that match the wild-card query by using **AND** of term list by using normal and reverse B tree. We just do **AND** of the two term lists.
2. **fil*er** : Similar operation as above
3. We will do a further **AND** on the term list that we got from processing both **se*ate and fil*er**
4. This may result in the execution of many Boolean **AND** queries.

A Compound Wild Card Query

Get all the documents : *se*ate* AND *fil*er*

1. ***se*ate*** : we have an enumeration of all terms in the dictionary that match the wild-card query by using **AND** of term list by using normal and reverse B tree. We still have to look up the postings for each enumerated term and take **OR** of them.
2. ***fil*er*** : Similar operation as above (**AND** of terms and **OR** of posting lists)
3. We will do a further **AND** on the document list that we got from processing both
4. This may result in the execution of many Boolean *AND* queries.

Using Additional Reverse B-tree Is Expensive

Expensive operation of many AND operations

The solution: transform wild-card queries so that the *'s occur at the end
i.e. convert suffix type to prefix type so that B tree can handle.

This gives rise to the **Permuterm Index**.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

Recap

- In the last class, we discussed one kind of “**tolerant retrieval**” using wild card query
- Discussed Wild Card Query implementation **based on two B- tree** i.e. one usual and the other reverse
- Because of performance issue by repeated AND, introduced **Permuterm Index** (rather tree) which is an additional index
- **Today's agenda :**
 - We will introduce **another method for wild card query**
 - Introduce the topic of **error correction for tolerant retrieval**

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Wild Card Query By K-gram Index

Bhaskarjyoti Das

Department of Computer Science Engineering

K-gram

- K-gram is sequence of k characters
- 2-grams are called **bigrams**.
 - Example: from “**April** is the cruelest month” we get the 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, as before.
- Enumerate all character k -grams occurring in a term

K-gram Indexes For $k=2$

- For every term in the standard inverted index, we will generate bigram or K-grams for $k=2$
- Maintain an inverted index from k-grams to the terms that contain the k-gram
 - Unlike the Permuterm Index posting list, posting list may contain more than one entry i.e. term
- Note that we have two indexes i.e. standard inverted index and bigram index



- Posting list of the k-gram index is also lexicographically sorted as we need to execute INTERSECT algorithm.

K-gram (bi-gram, tri-gram,..) Indexes

- Note that we now have two different types of inverted indexes
 - The term-document inverted index for finding documents based on a query consisting of terms
 - The k -gram index for finding terms based on a query consisting of k -grams

Processing Wild-carded Terms In Bigram (k=2) Indexes

- Query **mon*** can now be run as: **\$m** AND **mo** AND **on** AND **n\$**
- Posting list of the k-gram index is also lexicographically sorted
- We need to execute INTERSECT algorithm to find the common docIDs where all bigrams are appearing.
- Gets us all terms with the prefix *mon* . .
 - Many “false positives” like MOON that does not satisfy the query **mon*** but it is in posting list of all the k-grams
 - We must post-filter these terms against query.
- **Surviving terms** are then looked up in the **term-document inverted index** for the set of documents.

- k -gram index vs. permuterm index
 - k -gram index is more space efficient and fast.
 - Each term rotated may lead to many permuterm
 - Bigrams of k -grams are common across terms
- Permuterm index does not require post-filtering.

Does Google Use Wild Card Queries ?

- Google has very **limited support for wildcard queries**.
- For example, this query doesn't work very well on Google: [gen* universit*]
 - Intention: you are looking for the **University of Geneva**, but don't know which accents to use for the **French words for university and Geneva**
- Clearly due to the additional indexes, wild card query requires lot more work and hence **search engines hide/does not offer** this facility.
 - As of 2019, the only supported "wildcard" place holder in Google is "*" for missing words, not missing characters.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Wild Card Query By Permuterm Index

Bhaskarjyoti Das

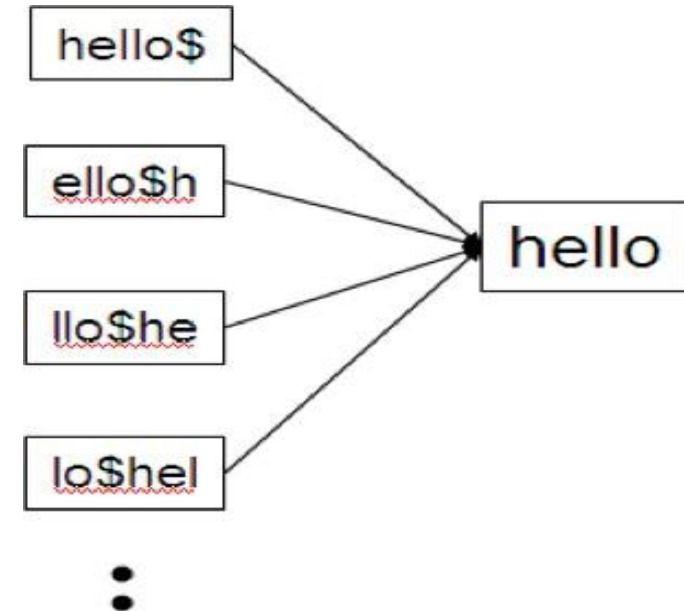
Department of Computer Science Engineering

How To Handle * In The Middle Of The Term ?

1. **Basic idea:** Rotate every wildcard query, so that the * occurs at the end for B-tree to handle.
2. Store **each of these rotations** in a dictionary i.e. in a B-tree where posting list will consist of the original unrotated term

Permuterm Index : Permuterm -> Term Mapping

- How to handle a non-prefix type wild card query ?
 - For term “hello”, support *ello, *iio, *lo, *o ?
 - We will rotate the terms around a symbol \$.
- Add *hello\$, ello\$h, llo\$he, lo\$hel, and o\$hell*



Permuterm -> Term Mapping

- For term “hello”: add *hello\$*, *ello\$h*, *llo\$he*, *lo\$hel*, and *o\$hell* to the B-tree where \$ is a special symbol
 - B Tree forms the dictionary with all the **rotated terms**
 - Each rotated term in the dictionary will **correspond to the same posting list that will have a single entry** i.e. original unrotated term
- Note that we have **two indexes**
 - standard inverted index
 - permuterm index

Permuterm Index For “hello”

- For HELLO, we’ve stored: *hello\$, ello\$h, llo\$he, lo\$hel, and o\$hell*
- Queries
 - For X, look up X\$ but you can also use standard inverted index
 - For X*, look up X*\$ but you can also use standard inverted index
 - For *X, look up X\$*
 - For *X*, look up X*
 - For X*Y, look up Y\$X*
 - Example: For hel*o, look up o\$hel*

Permuterm Index Is Really A Tree

- Permuterm index would better be called a **permuterm tree**. But **permuterm index** is the more common name.
- In effect, we are **converting a wild card query where * appears at the beginning into a query where * appears at the end** that B tree already supports

1. Rotate query wildcard to the right
2. Use B-tree lookup as before
3. Since there are many rotated terms for each term, this leads to bloated size of dictionary

Problem: Permuterm more than **quadruples** the size of the dictionary compared to a regular B-tree. (empirical number)

Empirical observation for English.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Introduction To Spelling Correction

Bhaskarjyoti Das

Department of Computer Science Engineering

Spelling Correction : Uses

- Two principal uses
 - Correcting **documents** being indexed
 - Correcting user **queries**

Two Methods Of Spelling Correction

- **Isolated word** spelling correction
 - Check each word on its own for misspelling
 - Will **not** catch typos resulting in correctly spelled words
 - e.g., *an asteroid that fell **form** the sky*
- **Context-sensitive** spelling correction
 - Look at surrounding words
 - Can correct *form/from* error above
 - e.g., *I flew form Heathrow to Narita.*

- **Document error correction** is primarily for OCR'ed documents. (OCR = optical character recognition)
 - Correction algorithms are tuned for this
 - 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).
- **Assumption :**
 - The dictionary contains fewer misspellings
 - But often we don't change the documents **and instead fix the query**

Query Mis-spelling

- Our principal focus here
 - E.g., the query *Alanis Morissett*
- Two options
 - Retrieve documents **indexed by the correct spelling** (user has no responsibility)
 - Return **several suggested alternative queries with the correct spelling** (user has responsibility)
 - *Did you mean ... ?*

- **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**
 - E.g., all words on the web
 - 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 will study **several alternatives**
 1. Edit distance and Levenshtein distance
 2. Weighted edit distance
 3. k -gram overlap

■ User interface

- automatic vs. suggested correction
 - “*Did you mean*” only works for one suggestion.
- What about multiple possible corrections?
- Tradeoff: simple vs. powerful UI

■ Cost

- Spelling correction is potentially expensive.
- Avoid running on every query? Maybe just on queries that match few documents.
- **Guess:** Spelling correction of major search engines is efficient enough to be run on every query.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Minimum Edit Distance

Bhaskarjyoti Das

Department of Computer Science Engineering

Introducing 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)
 - **Transposition** (with a pair of characters, not discussed in this course)
 - Convert GAOL to GOAL
 - Without transpositions you must do two edits: delete 'A', insert 'A' after 'O'.
 - With transpositions you must do one edit: transpose 'A' and 'O'
- The **edit distance** from *dof* to *dog* is 1
 - From *cat* to *act* is 2
 - from *cat* to *dog* is 3.

What Is The Minimum Number Of Operations ?

For the following strings and using back tracing, list the edit operations.

- source = a b c d e f replace, replace, replace, delete ?
- target = a z c e d

- source = a b c d e f replace, delete, replace ?
- target = a z c e d

- There can be **more than one way** to do the same. **Idea is to find the minimum.**

What Are We Trying To Do ?

Searching for an **optimal path (sequence of edit operations)** from a string **S1** to string **S2**

- **Initial state** : source string we are transforming
- **Goal state** : target string we are trying to get
- **Operations** : insert, delete, substitute
- **Cost** : we are trying to minimize the cost (in terms of total number of operations)

Naïve approach (too many recursive calls)

- Try **all possible edit operations** for all characters in the source string and **find the minimum**
 - **Space of all possible edit operations** is very large
 - **Lot of paths** may end up delivering the same result.

Edit Distance Defined

- The **Edit Distance (Levenshtein distance)** is a metric for measuring the number of operations required to transform the first string into the other.

Note: We are not considering these now

- We can also think of operation such as transpose ($ab \rightarrow ba$)
- We can think of having different cost for each operation

Dynamic Programming – Subproblem and Subsolution

1. **Overlapping sub-solutions:** The sub-solutions overlap. These subsolutions are computed over and over again when computing the **global solution** in a brute-force algorithm. It is Dynamic Programming.
2. **Optimal substructure:** The **optimal solution** to the problem contains within it **sub-solutions**, i.e., optimal solutions to subproblems. It is Divide and Conquer.

In case of Edit Distance :

1. **Sub-problem in the case of edit distance:** what is the edit distance of two prefixes
2. **Overlapping sub-solutions:** The solutions of the above sub-problems.

The Memoization Matrix

1. We reach end state by traversing through some **intermediate states**. There are **many paths for reaching each of these but we need to remember only the shortest path**
2. This concept of **remembering and reuse** of the optimum solution for a sub problem is Memoization. We implement in a matrix.

For Edit Distance :

1. Make **memoization matrix** by having source string along row and target string along column or vice versa. What we store in each cell is the optimum solution to each subproblem
2. Time complexity : $O(nxm)$; Space Complexity : $O(nxm)$ for strings X of size n and Y of size m

Defining Edit Distance

1. Given : X of length n and Y of length m.
2. We are trying to do $X \rightarrow Y$
3. Define $D(i,j)$ as edit distance between first i chars of X and j chars of X

Bottom up approach :

- Compute $D(i,j)$ for small values of i and j
- Compute $D(i, j)$ for larger i and j using smaller values stored in table
- Keep computing $D(i, j)$ for all $0 < i < n$ and $0 < j < m$ till we can calculate $D(n,m)$

Minimum Edit Distance Algorithm

Initialization :

$$D(i,0) = i$$

$$D(0,j) = j$$

Recurrence Relation :

For each $j = 1, \dots, m$

For each $i =$

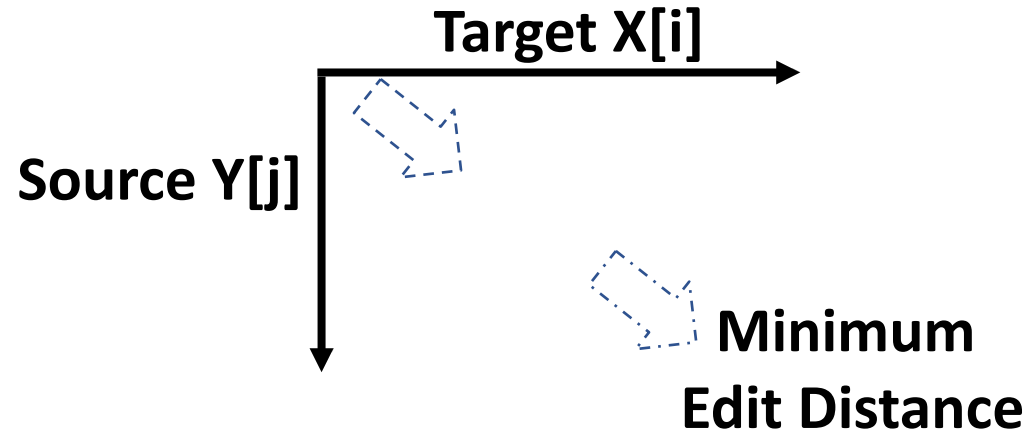
$1, \dots, n$

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1) + \begin{cases} 1 & \text{if } X(i) \neq Y(j) \\ 0 & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

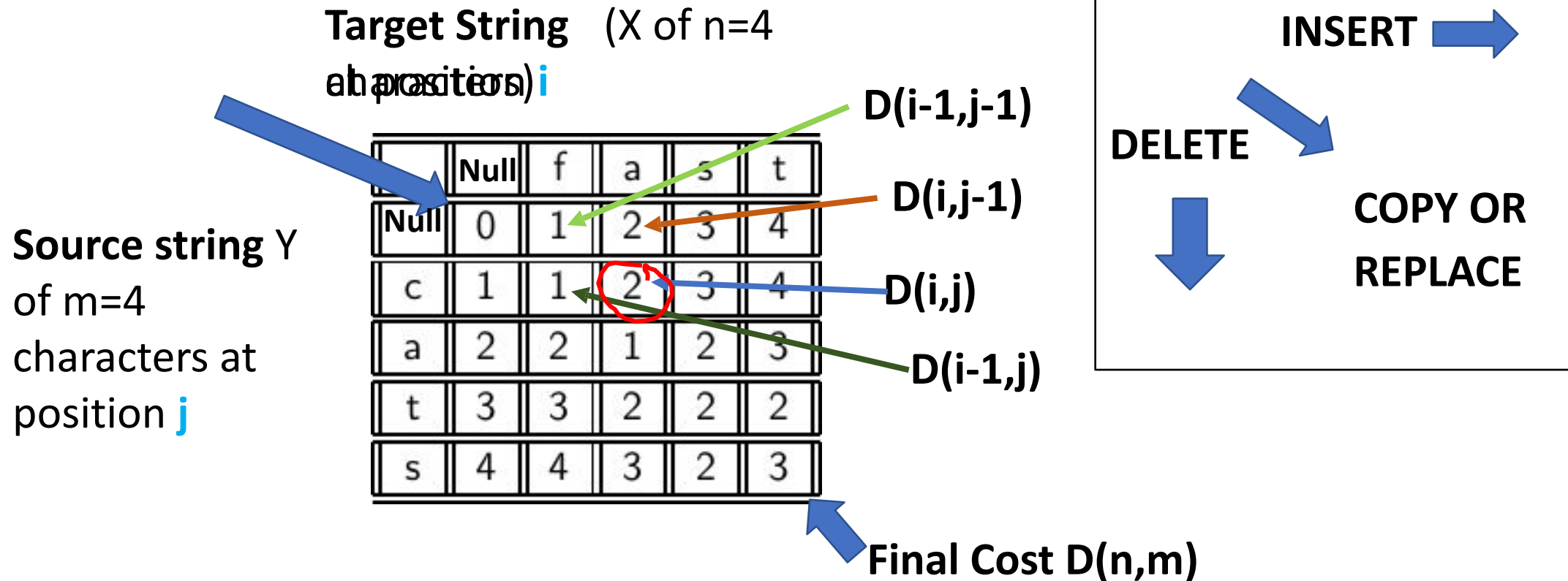
Termination :

$D(n,m)$ is the distance

Our aim is to calculate the termination value !



Problem Setting – Memoization Matrix



- We move from **left to right** for the two strings
- The other three cells correspond to smaller sub problems

Smaller sub problems and sub solutions

How to calculate the minimum edit distance given the three sub-solutions ?

1. **$D(i-1, j)$** correspond to edit distance between same prefix of Y but **one less character on X** (**delete** one character in $X \rightarrow i$)
2. **$D(i, j-1)$** correspond to edit distance between same prefix of X but **one less character on Y** (like **insert** one character in $y \rightarrow j$)
3. **$D(i-1, j-1)$** correspond to edit distance between one less character on both X and Y
 - So if $X[i] = Y[j]$, then essentially no work
 - else we have to do **a replace** operation

Adding Backtrace In The Memoization Matrix

1. In the memoization matrix, **store the pointers of from where we have come (for the operation chosen) along with the calculated minimum edit distance** so that we can **backtrace the optimum path** to arrive at the minimum edit distance



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Edit Distance Examples, Weighted Edit Distance

Bhaskarjyoti Das

Department of Computer Science Engineering

Minimum Edit Distance Example - 1

Source string : a d c e g Target string : a b c f g

	NULL	a	b	c	f	g
NULL	0	→ 1	→ 2	→ 3	→ 4	→ 5
a	↓ 1	↘ 0	→ 0+1	→ 1+1	→ 2+1	→ 3+1
d	↓ 2	↓ 0+1	↘ 0+1 ✓	→ 1+1	→ 2+1	→ 3+1
c	↓ 3	↓ 1+1	↓ 2	↘ 1	→ 2	→ 3
e	↓ 4	↓ 2+1	↓ 2+1	↓ 2	↘ 2 ✓	→ 3
g	↓ 5	↓ 3+1	↓ 4	↓ 3	↓ 3	↘ (2)

Minimum Edit Distance = 2 (as there are two “replace”) assuming
 copy=0, Insert, Delete, Replace = 1

Minimum Edit Distance Example - 2

Source string : a d c e g Target string : a b c f g

	NULL	a	b	c	f	g
NULL	→ 0	→ 1	→ 2	→ 3	→ 4	→ 5
a	↓ 1	↘ 0	→ 1	→ 2	3 →	4 →
d	↓ 2	↓ 1	↘ 1 ✓	↘ 2	↘ 3	↘ 4
c	↓ 3	↓ 2	↘ 2	↘ 1	↘ 2	↘ 3
e	↓ 4	↓ 3	↘ 3	↓ 2	↘ 2 ✓	↘ 3
g	↓ 5	↓ 4	↓ 4	↓ 3	↓ 3	↘ 2

Even if we made different choices, minimum edit distance is the same !

Minimum Edit Distance Example -3

Source string : a b c d e f Target string : a z c e d

	NULL	a	z	c	e	d
NULL	0	→ 1	→ 2	→ 3	→ 4	→ 5
a	↓ 1	↘ 0 (C)	→ 1	→ 2	→ 3	→ 4
b	↓ 2	↓ 1	↘ 1 (R)	→ 2	→ 3	→ 4
c	↓ 3	↓ 2	↘ 2	↘ 1 (C)	→ 2	→ 3
d	↓ 4	↓ 3	↘ 3	↓ 2 (D)	→ 2	→ 2
e	↓ 5	↓ 4	↘ 4	↓ 3	↘ 2 (C)	→ 3
f	↓ 6	↓ 5	↘ 5	↓ 4	↓ 3	↘ 3 (R)

T: a z c e d
 | R | D | R
S: a b c d e f

Minimum Edit Distance :
 Two Replace and 1 Delete

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Weighted Edit Distance

Bhaskarjyoti Das

Department of Computer Science Engineering

All Edit Distance May Not Be The Same

1. Some edit errors (and spelling mistakes) are **common errors** and some are **uncommon** as some words are more likely to be mistyped.
2. **Common errors** are well known and hence **easier to correct** !
3. **Cost of common error should be lower** than that of uncommon errors.
4. If the **cost is low** for **common words**
probability of correct prediction **will be higher for common words !**
5. How do we capture this ?

Weighted Edit Distance

- Weight of an operation depends on the **characters involved**.
- **Meant to capture keyboard errors**,
 - m more likely to be mistyped as n than as q .
 - Therefore, replacing m by n is a smaller edit distance than by q .
- We now require a **weight matrix** as input that the **algorithm can look up**.

Weighted Edit Distance

- Modify dynamic programming to handle weights to incorporate “commonality” of errors ?
- Replace **fixed cost** with **function** based on string characters involved

$D(i-1, j) + \text{del}(X(i))$

$D(i, j-1) + \text{ins}(Y(j-1))$

$D(i-1, j-1) : \text{subs}(X(i-1), Y(j-1))$



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Phonetic Spelling Corrections

Bhaskarjyoti Das

Department of Computer Science Engineering

- Idea is to detect errors **due to similar sounding words**. This happens frequently for names of persons.
- Class of heuristics to expand a query into **phonetic** equivalents
 - Language specific – mainly for names
 - E.g., ***chebyshev*** Vs. ***tchebycheff***
- Invented for the U.S. census ... in 1918

Soundex Algorithm

- Turn every **term** in corpus 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)
- We are talking about yet another index !

1. Retain **the first letter** of the word.
2. Collapse **all similar sounding letters to a single digit**
 - a. Change all occurrences of the following letters to '0' (zero): 'A', 'E', 'I', 'O', 'U', 'H', 'W', 'Y'.
 - b. 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 Algorithm

3. Remove all **pairs of consecutive digits (2 consonants sounding similar)** and replace with a single digit.
4. Remove **all zeros** from the resulting string.
5. 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. So is **Hermann**

Herman : H0rm0n -> H06505 -> H655

Hermann : H0rm0nn -> H065055 - > H06505 -> H655

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, ...)
- How useful is soundex?
 - Not very useful for information retrieval. Designed originally for similar sounding names but **may not work well for terms that are not names**
- Okay for **“high recall”** tasks (e.g., Interpol), though biased to names of certain nationalities. May not yield good precision
- Zobel and Dart (1996) show that other algorithms for phonetic matching perform much better in the context of IR



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

1. In the last class, we understood in detail a popular approach of Isolated Word Error Correction i.e. **Minimum Edit Distance** and looked at a variant - weighted edit distance
2. Today,
 1. In the first half, we will discuss what may be the possible difficulties with Minimum Edit Distance and then propose an alternative approach based on **n-gram index** that we learnt while covering wild card query
 2. After break, we will briefly review “**context sensitive error correction**”

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

K Gram Index For Spelling Corrections

Bhaskarjyoti Das

Department of Computer Science Engineering

Using Edit Distance – Approach 1

1. Given query, first enumerate **all possible character sequences** within a pre-set (optionally weighted) edit distance (e.g., 2)
2. **Intersect** this set of **candidate words** with list of “**correct**” words that we have in lexicon (corpus or otherwise)
3. Show terms you found **to user as suggestions and act on his input**
4. User takes control !

1. IR system can look up **all possible corrections in its inverted index** and return **all docs** ... slow
2. Refine this with a **single most likely correction** (by considering collection frequency)
3. The alternative disempowers the user, but saves a round of interaction with the user
4. IR system takes control

Issue With Both Approaches

- Expensive and slow as there are **too many candidate dictionary terms** for a given edit distance.
- It may explode with increasing edit distance and corpus!

Alternative: **cut the set of candidate dictionary terms**

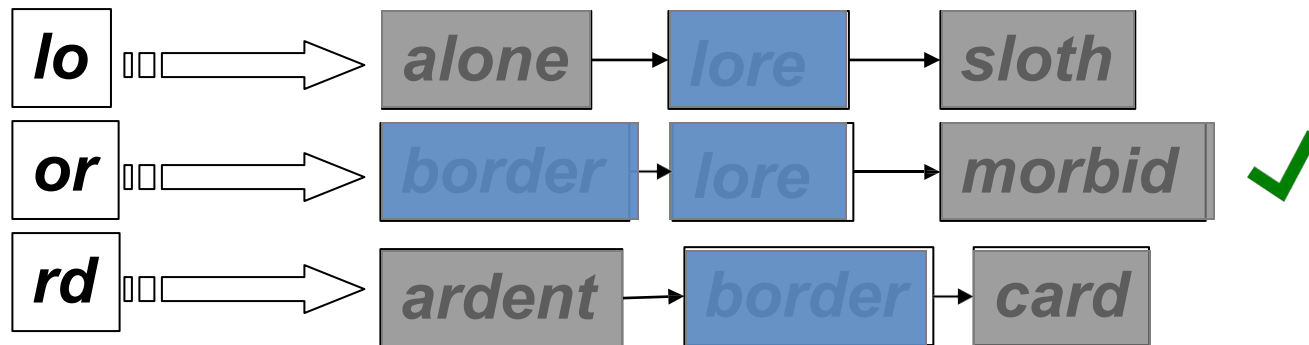
- One possibility is to use n -gram overlap for this

N-gram Overlap – Another Approach

- Enumerate all the ***n*-grams**
 - in the **query string**
 - as well as in the **lexicon**
- Recall we have **two dictionary** i.e. standard inverted index and **n-gram index**
- Use the ***n*-gram index** (recall wild-card search) to retrieve all **lexicon terms** matching any of the query *n*-grams. This gives a way to rank candidate terms.
- **Threshold by number of matching *n*-grams**

Matching Bigrams

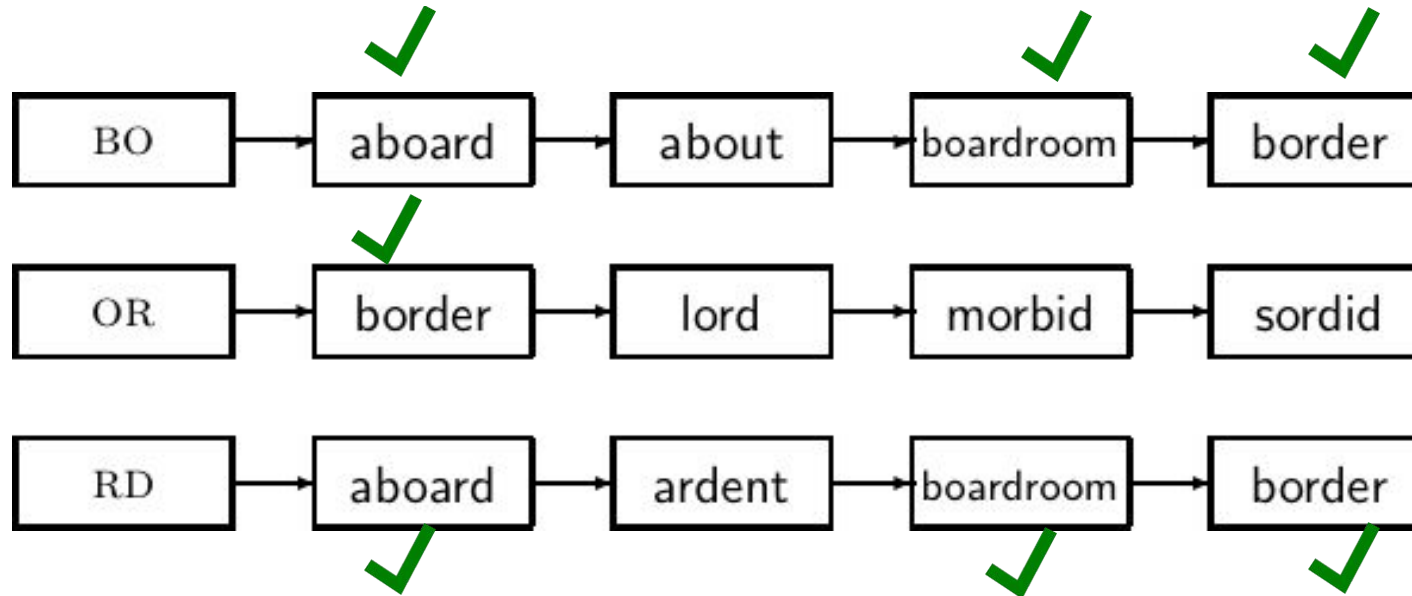
- Consider the query **lord** – we wish to identify words matching 2 of its 3 bigrams (**lo**, **or**, **rd**)



Standard postings “merge” will enumerate ...

Adapt this to using Jaccard (or another) measure.

Bi-gram index for spelling correction : bordroom



- Words matching at least 2 out of 3 bigrams –
aboard, boardroom, border
- Words matching at least 3 out of 3 bigrams – border
- Amusing thing is : if we choose the 2nd option, we will still
get a wrong answer !

Need To Have A Normalized Measure Of Overlap

- Suppose the term in standard inverted index is **november**

- Trigrams are: *nov, ove, vem, emb, mbe, ber.*

- The query term is **december**

- Trigrams are: *dec, ece, cem, emb, mbe, ber.*

- So **3 trigrams overlap** out of total **9 trigrams** in both terms (6 in each term)

- For two **very long words, we can however meet this threshold by chance !!**

- How can we turn this into a normalized measure of overlap?
Here comes Jaccard Coefficient !!

One Option : Jaccard Coefficient

- A commonly-used measure of overlap
- Let X and Y be two sets; then the J.C. is

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

- Equals 1 when X and Y have the same elements and zero when they are disjoint
- X and Y don't have to be of the same size
- Always assigns a number between 0 and 1
 - Now threshold to decide if you have a match
 - E.g., if J.C. > 0.8, declare a match



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary and Tolerant Retrieval

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Context Sensitive Spelling Corrections

Bhaskarjyoti Das

Department of Computer Science Engineering

- **Correct Text:** *I flew from Heathrow to Narita.*

- **Phrase query** “*flew form Heathrow*”

- **We’d like to respond**

Did you mean “*flew from Heathrow*”?

because no docs matched the query phrase.

Approach – 1 (Hit Based Approach Using Minimum Edit Distance)

- Need **surrounding context** to catch this.

- **First idea:**

1. Retrieve dictionary terms close (**in weighted edit distance**) to **each query term**
2. Now try all possible resulting phrases **with one word “fixed” at a time**

- **flew** *form heathrow*

- **fled** *from heathrow*

- **flea** *form heathrow*

Many Alternatives

Suppose that for “*flew form Heathrow*” we have 7 alternatives for flew, 19 for form and 3 for heathrow.

- How many “corrected” phrases will we enumerate?
- $7 * 19 * 3 = 199$
- Too many of them !

Hit-based refinement : Suggest the alternative that has lots of hits (occurrence). Search engines can find that.

1. **Hits in corpus** : How many times the “suggestion” appears in corpus
2. **Hits in query log** : How many times the “suggestion” appears in query logs of users
3. Since we are doing spelling correction in the query, **option 2 makes more sense**

- Divide and Conquer : **Correct at biwords level first**
 1. Break phrase query into a conjunction of biwords
 2. Look for biwords that need only one term corrected.
 3. Enumerate only phrases containing “common” biwords.

This is a refinement over **past approach that worked at the term level using Minimum Edit Distance.**

1. We enumerate **multiple alternatives** for “Did you mean?”
2. Need to **figure out which to present** to the user
 1. The alternative hitting most **docs in corpus**
 2. The alternative hitting most in **Query log**
3. More generally, **rank alternatives probabilistically**

$$\operatorname{argmax}_{corr} P(corr \mid query)$$

From Bayes rule, this is equivalent to

$$\operatorname{argmax}_{corr} P(query \mid corr) * P(corr)$$



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Construction

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Blocked Sort Based Indexing Part 1

Bhaskarjyoti Das

Department of Computer Science Engineering

1. Make **a pass** through the collection **assembling** all **term-docID pairs**.
2. **Sort the pairs** with the **term** as the primary key and **docID** as the secondary key.
3. Organize the docIDs for each term into a **postings list** and **compute statistics** like term and document frequency.

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

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

[illegible]

Reuter's 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 (incl. spaces/punct.)	6
•	avg. # bytes per token (without spaces/punct.)	4.5
•	avg. # bytes per term	7.5
•	non-positional postings	100,000,000

Key Step : Sort



- After all documents have been parsed, we have to sort **100M postings** in RCV1!.
- We **focus on this sort step**.

Challenge of In Memory Sorting



- **In-memory index construction** does not scale
 - In case of RCV1, it is only 100 Million term-docid pairs. Typical collections are much larger !
 - **Can't** stuff entire collection into memory, sort, then write back
- The earlier discussion assumed **“internal sorting”** (main memory based) and is not going to work.
- We need an **external** (data stored external to main memory) **sorting algorithm**.

Sort Using Disk As Memory ?



- Can we use the same index construction algorithm **using disk instead of memory?**
 - Even if **virtual memory** gets used, **disk seeks** will come into play
 - No: Sorting $T = 100,000,000$ records on disk is **too slow – too many disk seeks.**
- Example - If every comparison took 2 disk seeks, and N items could be sorted with $N \log_2 N$ comparisons, how long would this take?
 - Every disk seek takes in order of milli seconds and this translates to $100M \log_2(100M)$ ms which may be in years !!

Term vs. Term_id in Posting List



- To make index construction more efficient, we can represent **term** as **termid** where each *termid* is a **unique serial number**.
 - **Savings** : Term takes more number of bytes than term id

Creation of the term-termid dictionary:

- **Two-pass approach**
 - we compile the vocabulary in the **first pass**
 - construct the inverted index in the **second pass**.
 - **Multi-pass algorithms** are preferable in certain applications, for example, **when disk space is scarce**
- **Single pass approach**: We can build the **mapping from term to termid** on the fly while we are processing the collection

Challenge in Construction of Posting List



- As we build the index, we **parse docs one at a time**.
- Assume **12-byte (4+4+4) records or posting entries**(*termid, docid, freq*).
- The **final postings** for **any of 100 Million postings are incomplete until the end** (term may occur in the last document !).
- At **12 bytes per non-positional postings entry** (*termid, docid, freq*), demands a **lot of space** for large collections.
- $T = 100,000,000$ in the case of RCV1
- Solution : We **need to store intermediate results on disk** as we do not have so much main memory !

Basic Idea of the BSBI Algorithm



- **12-byte (4+4+4) records** (termid, docid, freq). These are generated as we parse documents.
- Must now **sort** 100M such 12-byte records **by termid** for RCV1 corpus

Basic idea of the proposed algorithm:

1. Divide and conquer
2. Define a **Block ~ 10M** such records
3. Will have **10 such blocks** to start with. Can easily fit a couple of them into memory.
4. **Accumulate postings for each block, sort, write index for each block** to disk.
5. Then **merge the index blocks** into one long index file.

Sorting 10 Blocks of 10 M Records Each



- First, read **each block and sort within**:
 - **Quicksort** is an **in-place** sort algorithm
 - Quicksort takes **$2N \ln N$** expected steps
 - In our case **$2 \times (10M \ln 10M)$** steps
- 10 times this estimate – gives us 10 sorted runs of 10M records each.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Construction

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Blocked Sort Based Indexing Part 2

Bhaskarjyoti Das

Department of Computer Science Engineering

Recap: External Sort with BSBI



- **12-byte (4+4+4) posting records** (termid, docid, freq).

These are generated as we parse docs.

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

Basic idea of the proposed algorithm for RCV1:

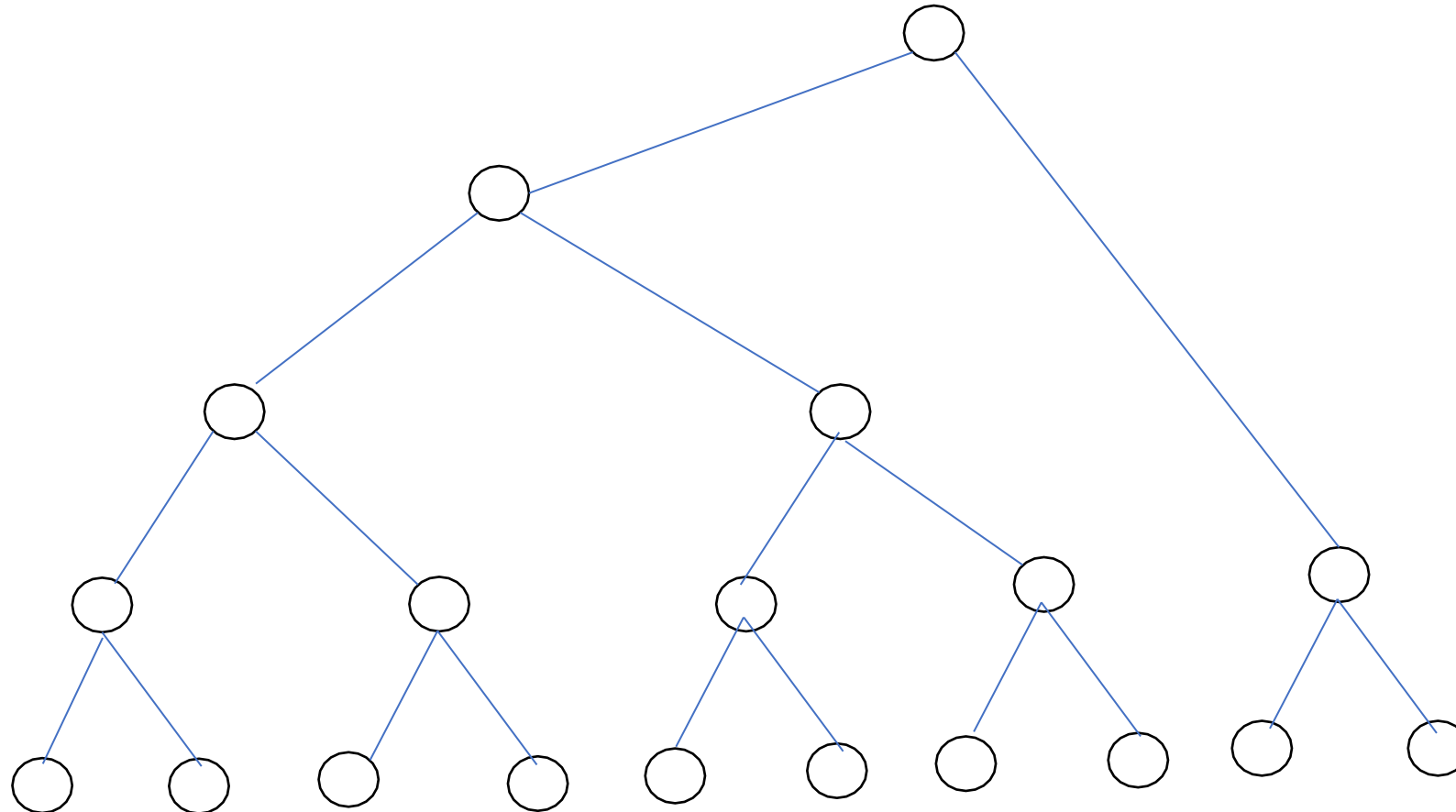
1. Divide and conquer
2. Define a **Block ~ 10M** such postings
3. Will have **10 such blocks** to start with. Can easily fit a couple of them into memory.
4. **Accumulate** postings for each block, **sort, write** to disk. **Each is an inverted index** for one part of the collection !
5. Then **merge the 10 index blocks** into one long index file

Main Steps

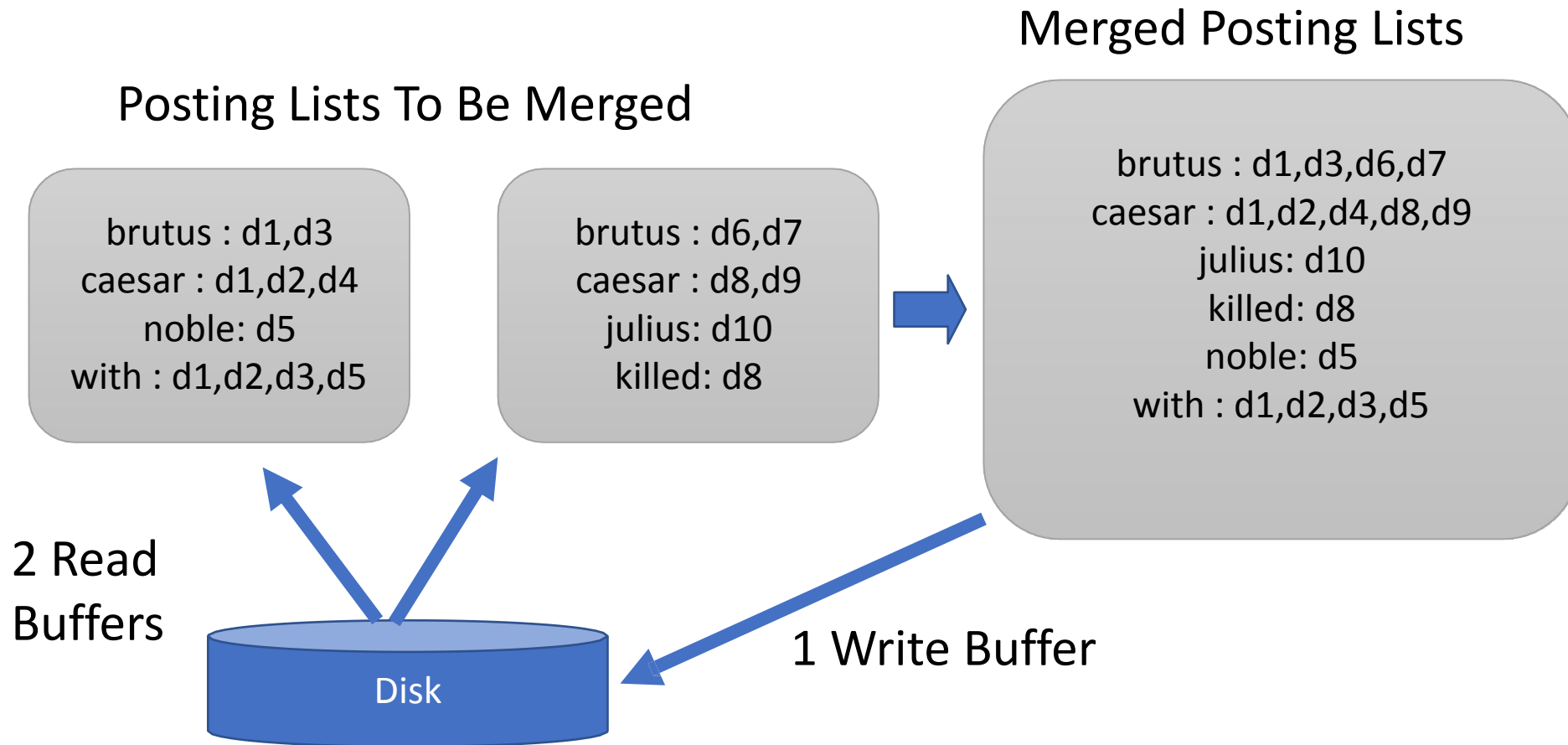
- **Step 1** : Sort the termid-docid pairs
- **Step 2** : Collect all such pairs with same termid into a posting list where posting is nothing but docid
- **Step 3**: Merge all the blocks

BSBI MERGEBLOCKS : Binary Merge

Binary Merge is **efficient**. For 10 blocks, we have $\log_2 10 = 4$ **layers or steps** to complete the merging !



MERGEBLOCKS : Blocks of Postings To Be Merged



Multi-way MERGEBLOCKS



1. Open each block file and maintain **read buffer** for each
2. Open a **write buffer** for the final merged index
3. For **each iteration**,
 - Read from all read buffers **simultaneously**
 - Select the **lowest term id that is not processed yet** using some data structure
 - **All posting lists** for this term id **are read and merged**
 - **Merged posting list** is **written back** using the write buffer
 - **Refill all read buffers** from respective file when required

This will **minimize disk seeks** because we are doing only **linear scans of all the blocks** (can't do better than that)

Blocked Sort Based Index Construction (BSBI)

BSBIINDEXCONSTRUCTION()

1 $n \leftarrow 0$

2 **while** (all documents have not been processed)

3 **do** $n \leftarrow n + 1$

4 $block \leftarrow \text{PARSENEXTBLOCK}()$

5 $\text{BSBI-INVERT}(block)$

6 $\text{WRITEBLOCKTODISK}(block, f_n)$

7 $\text{MERGEBLOCKS}(f_1, \dots, f_n; f_{\text{merged}})$

Block Number

Generate term id, doc id
pair to fill the block

1

Sort the Block and create index

2

Write the Block

3

Merge all Blocks
Into large index

4

Components of BSBI Algorithm

Components of BSBI:

- PARSENEXTBLOCK
- BSBI-INVERT
- WRITEBLOCKTODISK
- MERGE_BLOCKS

How Expensive is BSBI ?



- **Out of the 4 steps**, the step with highest time complexity is the SORT operation is **$O(T \log T)$** where T is the number of pairs (termid, docid) just like any merge sort time complexity
- However, more time is spent in parsing the document (**PARSENEXTBLOCK**) and final merge (**MERGEBLOCKS**)

BSBI Vs. Merge Sort



- **BSBI is external Merge Sort** when the data being sorted cannot be accommodated in memory.
- Here we **do not come down to size 1 for each constituent** block (typical merge sort in a recursive execution) but we are coming down to a **size of 10M or some size** that can be accommodated in main memory
- It addresses the following dependencies
 - does NOT depend on **memory size**
 - works **as long as the size of the files generated can fit into the hard disk**

- Collection can be **fitted into** **hard disk**
of
the machine making the index
- The **corpus is static**.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Construction

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Single Pass In-memory Indexing (SPIMI)

Bhaskarjyoti Das

Department of Computer Science Engineering

Opportunities for Improvement with BSBI Sort Based Algorithm



1. BSBI sort based indexing has excellent scaling properties
2. But ...
 - We work with a **term id and get term-> term id from a mapping**. We always have to carry this **dictionary** in main memory.
 - **Our assumption** was: we can keep the large dictionary in memory. But that may not be feasible !
 - In BSBI, **each block was sorted** and kept before the merge step. **Can we avoid this expensive step ?**

Single Pass In-memory Indexing (SPIMI)



- **Key idea 1:** Generate **separate dictionary for each block** i.e. no need to maintain term-term ID mapping **across blocks**. Use term instead of termid in posting list ?
- **Key idea 2: Don't sort**. Accumulate postings in postings lists as they occur instead of first collecting all termid-docid pairs and then sorting them.

Single Pass In-memory Indexing (SPIMI)



1. We are also creating a **separate dictionary for each block** (no need to carry the dictionary in memory across blocks)
2. Since docids are sequentially accessed, the **posting list will be pre-sorted on docid** ! We eliminate a “sort” step.
3. **Compression technique** can be used to store a compact version of the index for a block (require less space on disk)

Advantages:

1. We are able to directly track term-> docid through this dynamic (growing) posting list . **So it saves memory**
2. Overall it is **faster due to elimination of one sorting step**
3. Compression allows Individual **blocks to be larger** and **overall algorithm efficiency higher** (eliminate more disk seek time)

How Do these Ideas work ?



- **Till we run out of main memory, work with the current block**
 - docid **sequentially accessed**
 - Let's say a document (**docid n**) is **getting parsed**
 - If the **term is new**, add it to the **dictionary for the current block**
 - else **identify the record** for the particular term that is **already existing**
 - If initially allocated **small posting list is full**, double it
 - Add the **docid n to the posting list** for this term
 - **Sort on terms for lexicographic ordering**
 - Write the **index block (dictionary and posting list) into the disk**
 - **Start the next block with a new dictionary**

SPIMI-INVERT

Parse document and generate **token stream** of (term-docid) pair
.....

SPIMI-INVERT(*token_stream*)

Each call writes a Block to disk

```
1  output_file ← NEWFILE()
2  dictionary ← NEWHASH()
3  while (free memory available)
4  do token ← next(token_stream)
5      if term(token) ∉ dictionary
6          then postings_list ← ADDTOdictionary(dictionary, term(token))
7          else postings_list ← GETPOSTINGSList(dictionary, term(token))
8          if full(postings_list)
9              then postings_list ← DOUBLEPOSTINGSList(dictionary, term(token))
10     ADDTOPOSTINGSList(postings_list, docID(token))
11 sorted_terms ← SORTTERMS(dictionary)
12 WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13 return output_file
```

Generate the next token pair (term, doc id)

Merging of blocks is analogous to BSBI.

- *single-pass in-memory indexing* or *SPIMI* is more scalable.
 - SPIMI uses **term** instead of **termIDs**, writes **each block's dictionary and posting list to disk, and then starts a new dictionary** for the next block.
 - **Time complexity of SPIMI is $O(T)$** since no sorting of tokens involved and all operations are at best linear to size of collections
 - **Compression** can help make **SPIMI even better**

SPIMI Compression is an Added Feature !

- **Compression makes SPIMI even more efficient.**
 - Compression of terms
 - Compression of postings

Same Limitations Like BSBI

- Collection should **fit into hard disk of the computer generating the index**
- The corpus should be **static**

More Advanced Index Construction ?



- Corpus need not be static. The corpus can continuously change.
- It may not be possible to fit the Corpus even in a large hard disk of the computer processing the index.
 - For example, web corpus will never be static and will be so large that it cannot be fitted in a hard disk of a computer
- This is where the Distributed and Dynamic Index come in !!



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Index Construction

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Hardware Basics

Bhaskarjyoti Das

Department of Computer Science Engineering

Why Hardware Basics

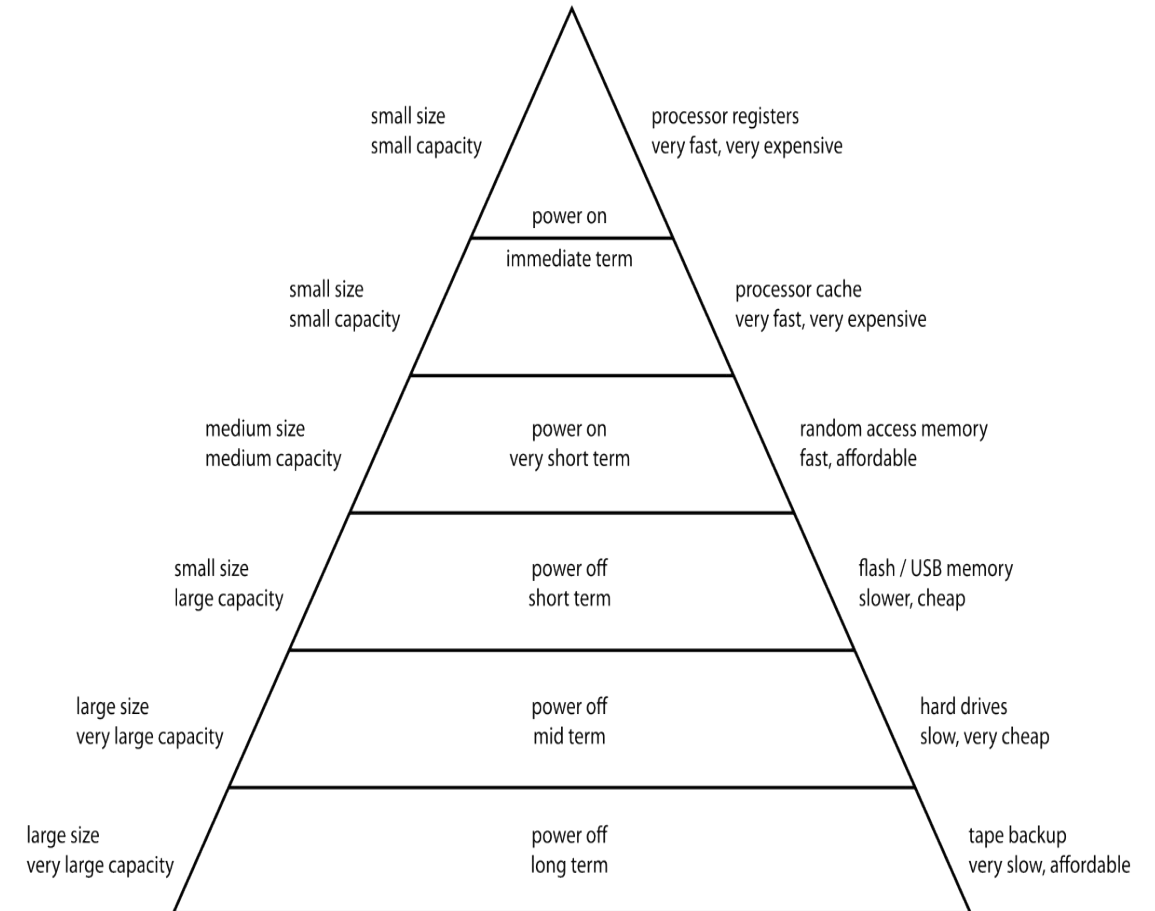
- Many design decisions in information retrieval are based on the characteristics of hardware
- We begin by reviewing hardware basics

Memory Hierarchy

- At higher level, memory is smaller in size, faster, more expensive and nearest to processor.
- At upper level, data can only be a subset of the data at a lower level



Computer Memory Hierarchy



Hardware Assumptions



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

1. Access to data in **memory is much** faster than access to **data on disk**.
2. **Disk seeks**: No data is transferred from disk while the **disk head is being positioned** (Disk seek time + Rotational delay vs. Transfer time)
3. **Disk seek time** in order of **milli sec** whereas **transfer time** in order of **microsecond**
4. Transferring **one large chunk** of data from disk to memory is **faster** than transferring **many small chunks**. Hence the need to have contiguous data on disk

- **Disk I/O is block-based:** Reading and writing of entire blocks (as opposed to smaller chunks) as **reading a chunk may be more efficient** than reading many chunks. Block data being read or written is stored in *buffer* (an area of main memory)
- Data transfers **from disk to memory** are handled by the **system bus**, not by the processor.
- This means that the processor is **available** to process data during disk I/O.
- Assuming an **efficient decompression algorithm**, the total time **of reading and then decompressing compressed data is usually less than reading uncompressed data.**

- 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 Reuter's RCV



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.

Reuter's 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 (incl. spaces/punct.)	6
•	avg. # bytes per token (without spaces/punct.)	4.5
•	avg. # bytes per term	7.5
•	non-positional postings	100,000,000



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Compression

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Term Statistics Part 1

Bhaskarjyoti Das

Department of Computer Science Engineering

Why Compression in General ?



- Use less disk space (saves money)
- ✓ ■ Keep more stuff in memory (increases speed)
- ✓ ■ Reading compressed data and decompressing that in memory is faster than reading uncompressed data with many disk seek
 - **Premise:** Decompression algorithms are fast.
 - This is true of the decompression algorithms we will use.

Why Compression in Information Retrieval ?



- **Dictionary**

- **Main motivation for dictionary compression:** make it small enough to keep in main memory

- **Postings list**

- **Motivation:** reduce disk space needed, decrease time needed to **read from disk**
- **Note:** Large search engines keep significant part of postings in memory (compression techniques help)
- We will devise **various compression schemes** for dictionary and postings.

symbol	statistics	value
N	documents	800,000
L	avg. # tokens per document	200
M	word types	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term (= word type)	7.5
T	non-positional postings	100,000,000

Effect of Pre-processing

size of	word types (term)			non-positional postings			positional postings (word tokens)		
	dictionary			non-positional index			positional index		
	size	Δ	cml..	size	Δ	cml..	size	Δ	cml..
unfiltered	484,494			109,971,179			197,879,290?		
no numbers	473,723	-2%	-2%	100,680,242	-8%	-8%	179,158,204	-9%	-9%
case folding	391,523	-17%	-19%	96,969,056?	-3%	-12%	179,158,204?	-0%	-9%
30 stop w's	391,493	?-0%	-19%	83,390,443?	-14%	-24%	121,857,825	-31%	-38%
150 stop w's	391,373	?-0%	-19%	67,001,847	-30%	-39%	94,516,599	-47%	-52%
stemming	322,383?	-17%	-33%	63,812,300?	-4%	-42%	94,516,599?	-0%	-52%

Lossless vs. Lossy Compression



- **Lossy compression:** Discard some information
- Several of the **preprocessing steps** we frequently use can be viewed **as lossy compression**:
 - Down casing, stop words, porter stemmer, number elimination
- **Lossless compression:** All information is preserved.
 - What **we mostly do in index compression**
 - Prune posting entries that are unlikely to turn up in the top K list for a query with no loss of quality



**THANK
YOU**

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Construction

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Distributed Indexing – MAP REDUCE Based Implementation

Bhaskarjyoti Das

Department of Computer Science Engineering

Distributed Indexing

- Distributed Indexing works well for collections that **cannot be fitted into a single machine**
- First implemented using **MAP REDUCE** on Distributed File System
- Maintain a **master machine directing the indexing** job – considered “safe”

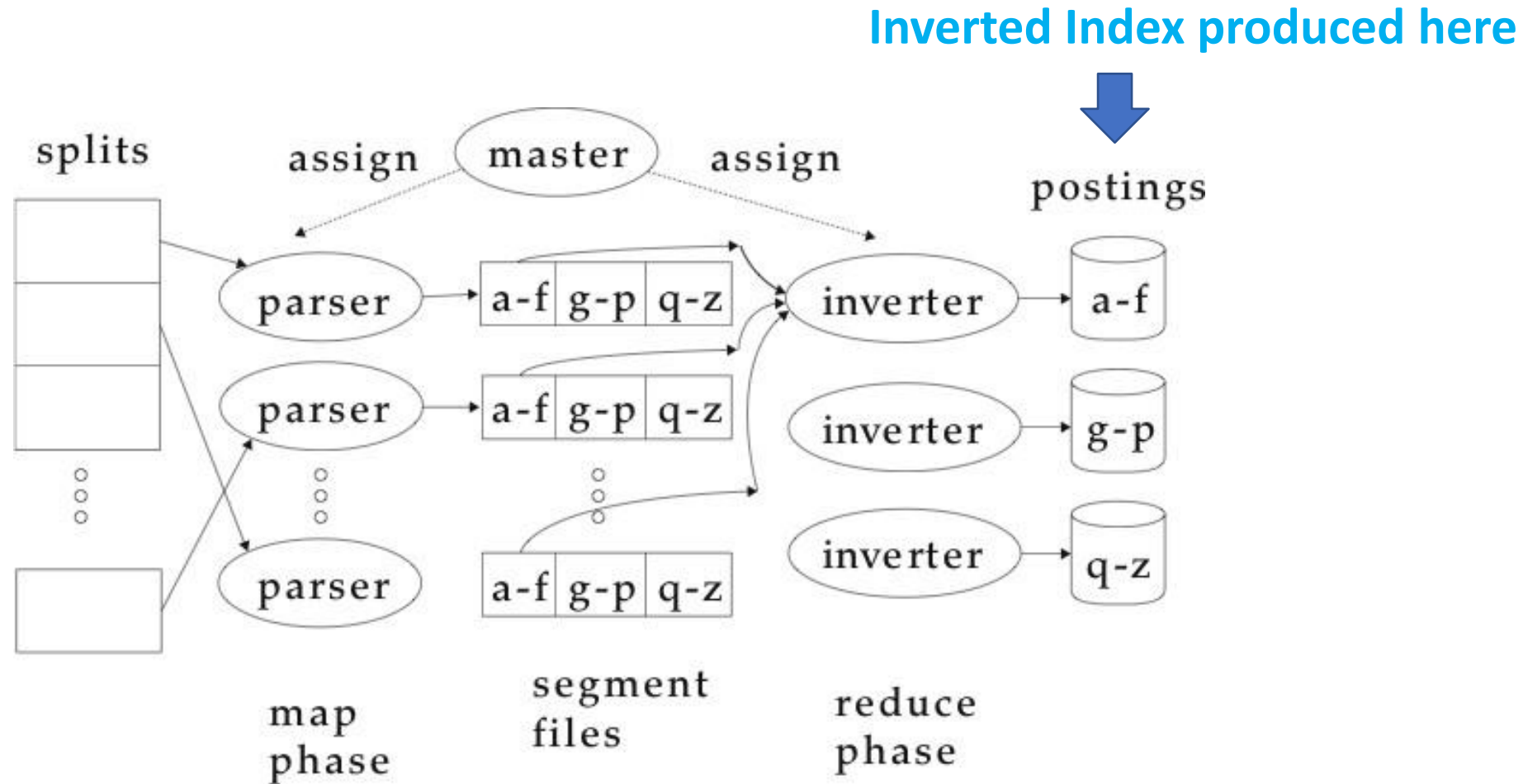
- We will define **two sets of parallel tasks** and deploy **two types of machines/tasks** to solve them:
 - **Parsers**
 - **Inverters**
- Break the **input document collection** into **splits** (corresponding to **blocks** in BSBI/SPIMI)
- Each **split is a subset of documents**.

- **Master** assigns a **split** to an **idle parser** machine.
- **Parser**
 - reads a document at a time and **emits** (term,docID) pairs.
 - writes pairs into j **term-partitions**
- Each for a **range of terms' first letters**
 - Example : a-f, g-p, q-z (here: $j = 3$)

Inverter (REDUCER nodes)



- An **inverter** collects **all key-value (term,docID) pairs** (= postings) **for one term-partition** (e.g., for a-f).
- **Sorts and writes to postings lists**



Schema for Index Construction



- **Schema of map and reduce functions**

- map: $\text{list}(k, v) \rightarrow \text{reduce: } (k, \text{list}(v)) \rightarrow \text{output}$

- **Instantiation of the schema for index construction**

- map: $\text{collection} \rightarrow \text{list}(\text{termID}, \text{docID})$

- reduce: $(\langle \text{termID1}, \text{list}(\text{docID}) \rangle, \langle \text{termID2}, \text{list}(\text{docID}) \rangle, \dots) \rightarrow (\text{postings list1}, \text{postings list2}, \dots)$

Example For Index Construction



■Map:

- d1 : C came, C c'ed.
- d2 : C died. →
- <C,d1>, <came,d1>, <C,d1>, <c'ed, d1>, <C, d2>, <died,d2>

■Reduce:

- (<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <c'ed,(d1)>) →
(<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c'ed,(d1:1)>)

Few Points To Be Noted



- Key–value pair is (term id : doc id) and **term-> term id map on distributed nodes for BSBI**? A possible solution is to pre-compute and have them on all MAP nodes
- **Parsing** of documents on MAP nodes is **same as Parsing** using BSBI and SPIMI
- **MAP nodes parse and write the key-value pairs** into intermediate local segment files (number of segment files = number of partitions).
- Each segment file requires only a sequential read as all **data relevant to an inverter** are only in **written to a single segment file** by the parser. This **minimizes network traffic during indexing**

Few Points To Be Noted -2



- 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.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Compression

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Term Statistics Part 2

Bhaskarjyoti Das

Department of Computer Science Engineering

How Big is Term Vocabulary ?



- That is, how many distinct words are there?
- **Can we assume there is an upper bound?**
- Not really: The vocabulary will **keep growing with collection size**. But there is some heuristics.
 - **Heap's Law: $M = kT^b$**
 - **M** is the size of the vocabulary, **T** is the number of **tokens** in the collection.
 - Typical values for the parameters **k and b** are: $30 \leq k \leq 100$ and $b \approx 0.5$.
 - Heaps' law is **linear in log-log space**.
 - It is the simplest possible relationship between **collection size and vocabulary size in log-log space**.
 - Empirical law

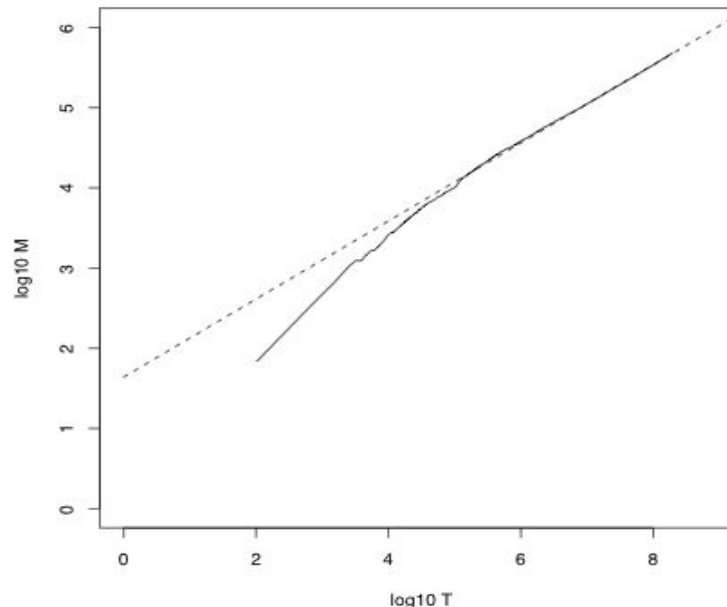
Heap's Law for Reuters Dataset

Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1.

For these data, the dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the **best least squares fit**. Thus,

$$M = 10^{1.64} T^{0.49}$$

and $k = 10^{1.64} \approx 44$ and $b = 0.49$



Exercise

- ① What is the effect (on M and T) of including spelling errors vs. automatically correcting spelling errors on Heaps' law?
- ② Compute **vocabulary size** M in the following case
 - Looking at a collection of web pages, you find that there are **3000 different terms** in the **first 10,000 tokens** and 30,000 different terms in the first 1,000,000 tokens.
 - Assume a search engine indexes a total 20,000,000,000 of (2×10^{10}) tokens on average pages, containing 200
 - What is the **size of the vocabulary** of the indexed collection as predicted by Heaps' law?

Zipf's Law



- Using Heap's Law, we have characterized the growth of the vocabulary in collections.
- We also want to know **how many frequent vs. infrequent terms** we should expect in a collection.
- In natural language, there are a **few very frequent terms** and **many very rare terms**
 - **Long Tail** Phenomena

Zipf's Law



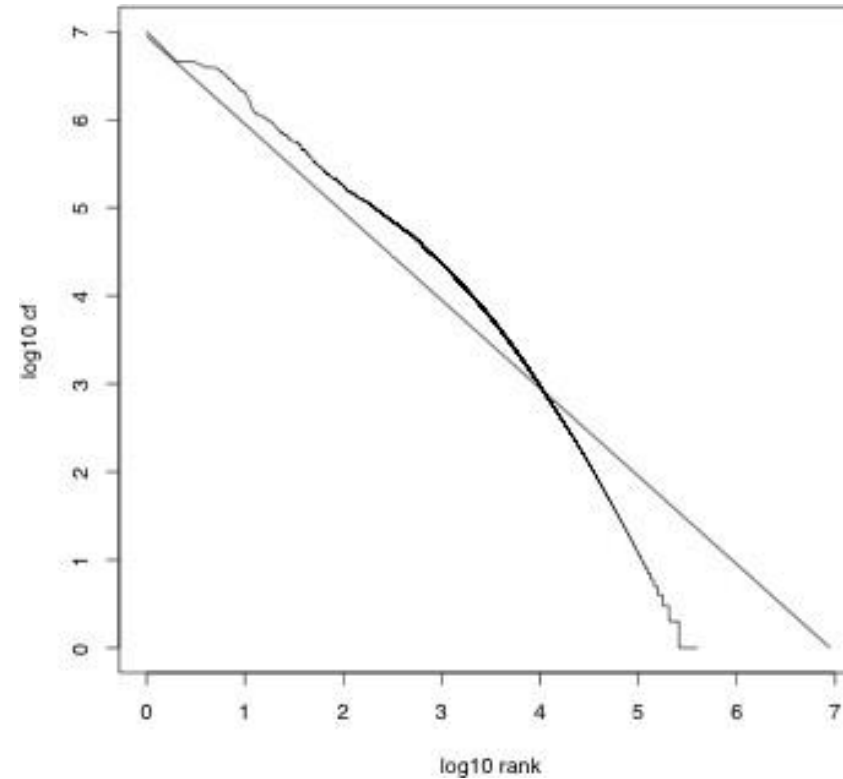
- Zipf's Law: The i^{th} most frequent term has frequency cf_i , proportional to $1/i$
 - cf_i is collection frequency: the number of occurrences of the term t_i in the collection.
 - So if the most frequent term (**the**) occurs cf_1 times, then the second most frequent term (**of**) has half as many occurrences ($cf_2 = cf_1/2$)
 - . . . and the third most frequent term (**and**) has a third as many occurrences ($cf_3 = cf_1/3$)
- If $K = cf_1$, Equivalent: $cf_i = K/i$ and $\log cf_i = \log(K) - \log(i)$
Example of a power law

Zipf's Law For Reuters Dataset

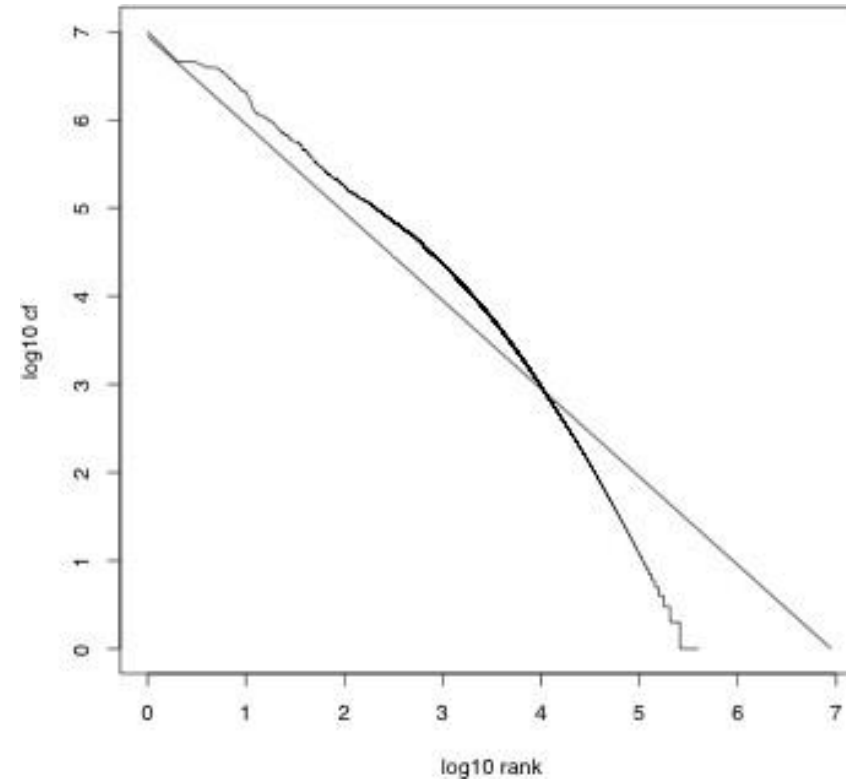
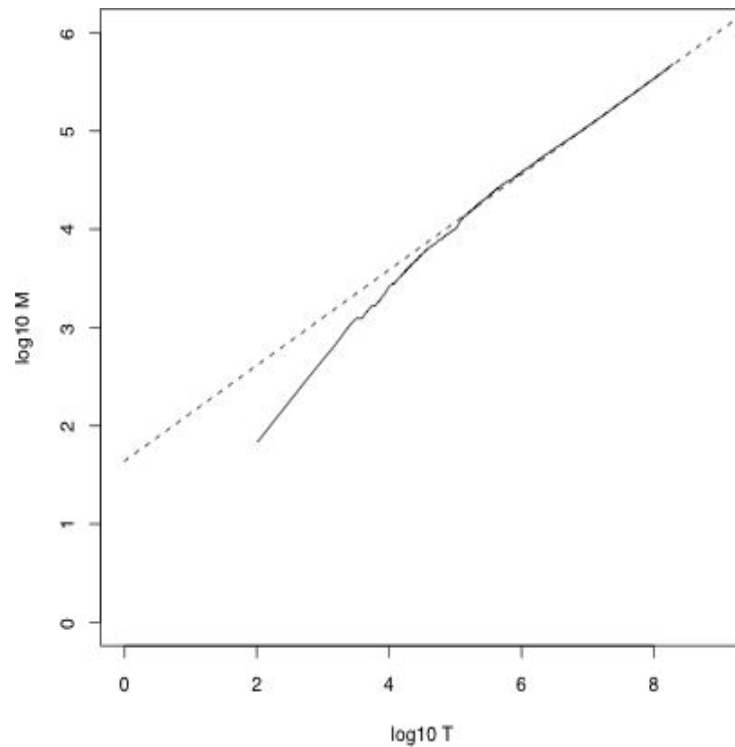
Straight Line Fit is not great.

What is important is the key insight:

Few frequent terms, many rare terms.



Heap's Law and Zipf's Law are both Power Laws





**THANK
YOU**

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Construction

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Distributed Indexing – MAP REDUCE and HDFS Recap

Bhaskarjyoti Das

Department of Computer Science Engineering

Google example (hypothetical)

- **10 Billion web pages**
- Average size of web page = 20 KB
- Total 10 B * 20 KB = 200 TB
- Disk I/O channel bandwidth = 50 MB/sec
- **Total time to read 200 TB** = 4 Mill sec = 47 days !!

Computation Failure



- As of **2011 news report**, Google used around **9 Million servers for search indexing**
- Single server can stay up for around 1000 days
- If a cluster has 1000 servers, 1 failure/day
- For **1 M servers, 1000 failures /day**
- What is the fault-tolerance strategy ?

Network Bandwidth is a Bottleneck

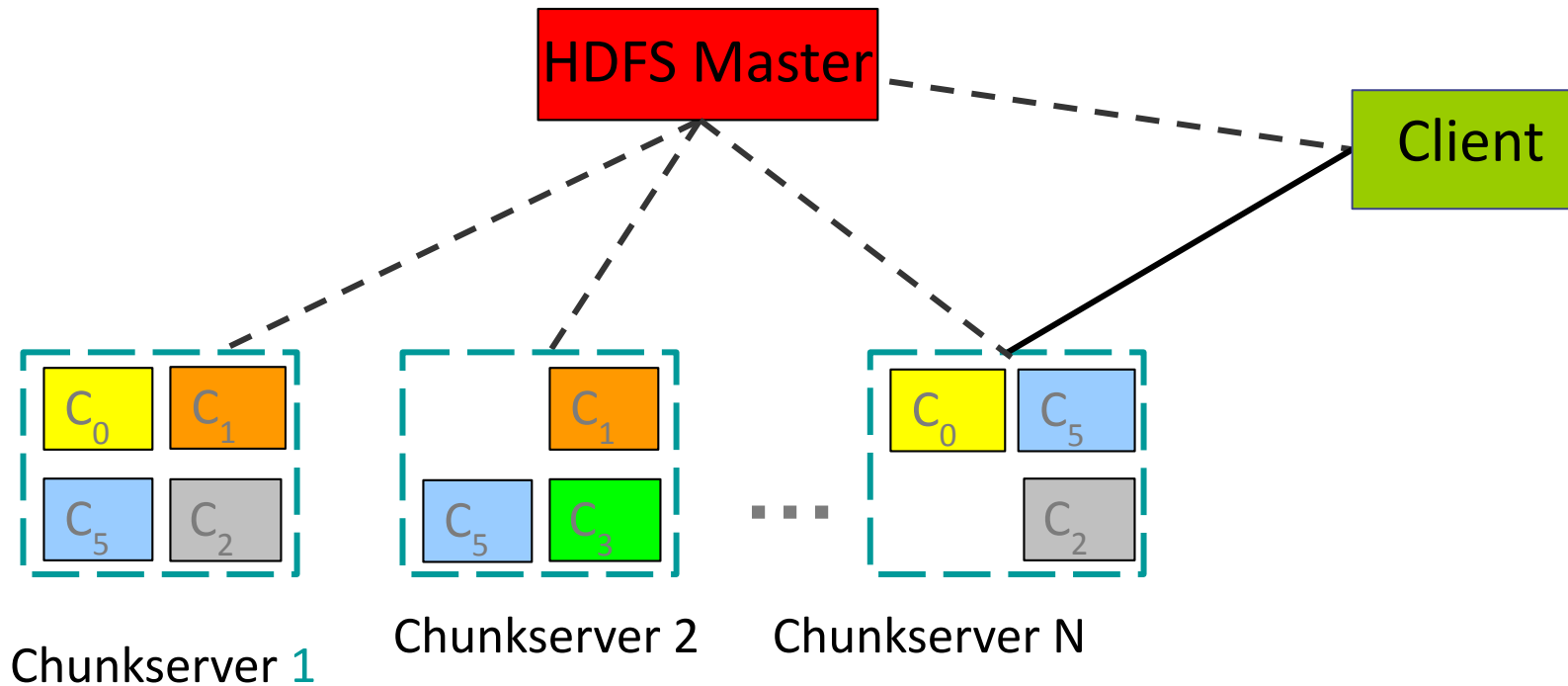
- Typically **switch provides 1 GBPS**
- **Moving 10 TB** will take around **a day** !

- Store **persistently on multiple nodes** to address **PERSISTENCE and AVAILABILITY**
- **Move computation to data** (traditional: data moves to computation) to **minimize data movement** and **address data locality issue**
- **Simple programming model** to hide all these complexity. This programming model is designed to **minimize data movement** across the switches.

Redundant Storage Infrastructure

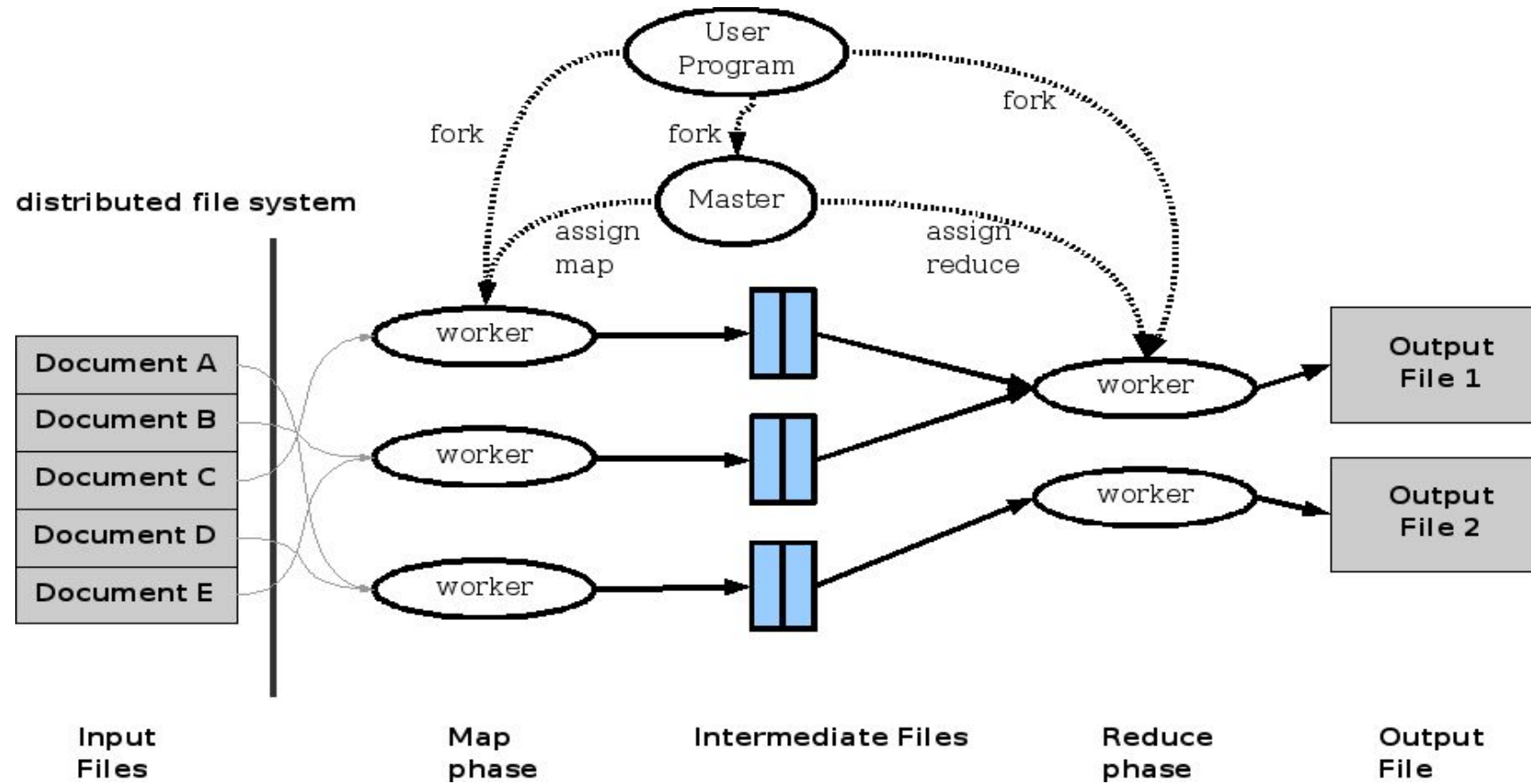
- **Google GFS , Hadoop HDFS addressing availability and persistence**
 - Data mostly **read and appended** , rarely updated
- **Chunk Servers ensure availability**
 - File is **split into contiguous chunks** (16-64 MB)
 - Each **Chunk is replicated** (2 x or 3x)
 - Even try to keep the **replicas in different racks**
- **Master node / Name node in HDFS to conduct the orchestra**
 - Stores **metadata about where files** are stored
 - May be replicated
- **Client library** to access data
 - **Talk to master** to get **meta data** (where are the chunks)
 - Then talks directly to **chunk server to get the data**

Redundant Storage Infrastructure



- Provides a **platform over which other systems like MapReduce** operate

MAP REDUCE EXECUTION OVERVIEW



- The **partitioning phase takes place after the map phase and before the reduce phase.**
 - The **number of partitions** is equal to **the number of reducers.**
 - The **data gets partitioned across the reducers** according to the partitioning function .
- Often a **MAP task will produce many pairs of the form (k1,v1), (k2, v2) ..for same key K**
 - Can save network time by **pre-aggregating the values in the mapper**..combiner is usually same as Reducer function
 - Benefit : much less data need to be copied and shuffled

MAP REDUCE Conclusion



- Provides a **general-purpose model** to simplify **large- scale computation**
- Allows users to **focus on the problem** without worrying about nuts and bolts of details
- MapReduce breaks a **large computing problem into smaller parts** by **recasting** it in terms of manipulation of *key-value pairs*
- **Master** machine **assigns each task** to an **idle machine from** a pool.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Construction

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dynamic Indexing

Bhaskarjyoti Das

Department of Computer Science Engineering

Why Dynamic Indexing ?



- Up to now, we have assumed that **collections are static**.
 - The corpora rarely are **static** : Documents are inserted, deleted and modified.
- **Dynamic Corpus** means that the dictionary and postings lists have to be **dynamically modified**.
- All search engines support **dynamic indexing**
- In this lecture, we are **NOT** talking about the general scenario - **Distributed Dynamic Index**

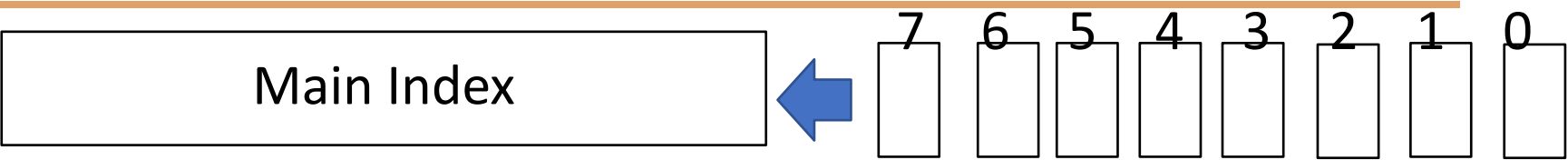
- Maintain big **main index on disk**
- New docs go into **small auxiliary index in memory**.
 - Search **across both, merge results**
 - Periodically, **merge auxiliary index** into **big index** when it grows too big for main memory
- How do we handle **Deletions** ?
 - **Invalidation bit-vector** for deleted docs
 - Filter docs returned by index using this bit-vector

Issues with Main and Auxiliary Indexes



- **Frequent merges** – cost of merging depends on how we store the index in the file system
- Poor **search performance** (due to extensive merge operation) during index merge
- **Alternative ?**
 - Merging of the auxiliary index into the main index is not that costly if we keep a **separate file for each postings list**.
 - Merge is the same as a **simple append**.
 - But then **no of files = no of terms !!**
- **In reality: Index is really several big files** (use a scheme that splits very large postings lists into several files, collect small postings lists in one file etc.)
- **Merging is always an issue !!**

Logarithmic Merge



At t = 0	writing into Z_0 (main memory)			
At t = 1	$Z_0 > n$, write into I_0	I_0	0000 0001	
At t = 2	$Z_0 > n$, merge with I_0 and write into $I_1 = 2 I_0$	I_1	0000 0010	
At t = 3	$Z_0 > n$, write into I_0	I_1 I_0	0000 0011	
At t = 4	$Z_0 > n$, we have $I_1, I_1 = 2 I_0$ So, we create $I_2 = 2 I_1$	I_2	0000 0100	

Logarithmic Merge



- Maintain a series of indexes, **each twice as large as the previous one**
 - **One or more of these indexes** may exist at a time
 - 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)** and write merger to I_1 etc.
- Logarithmic merging amortizes the cost of merging indexes over time.
 - → Users see smaller effect on response times.

How a Query Will Be Answered ?



- It is still **like the Primary and Auxiliary index scenario**
- **Query will be submitted to all the indexes** (including the one in the memory)
- Final **MERGE will be done on the answers**

Naïve Non-logarithmic Approach



- Let n be the size of the auxiliary index and T is total number of postings finally
- We store the index as **one large** file i.e. concatenation of all posting lists
- We do the **merging of index (T/n) times** and we process **each posting (T/n) times during each of the T/n merges**
- So, overall time complexity is **approximately $O(T^2/n)$**

Logarithmic Merge vs. Naïve Merge



- **Logarithmic Merge** can do better than Naïve Approach
- Since $I_0, I_1, I_2 \dots$ are of size $2^0 \times n, 2^1 \times n, 2^2 \times n \dots$ etc. , we are introducing $\log_2 (T/n)$ indexes
 - **Postings percolate up this sequence** of indexes and are **processed only once on each level.**
- **Pro:** Since each posting of T is processed only once on each of the $\log_2 (T/n)$ indexes, overall index construction time is $O(T \log_2 (T/n))$. This is an **efficiency gain**
- **Con:**
 - We **trade this gain** for a **slow down of query processing as multiple indexes are there.**
 - Also we now have to merge $\log_2 (T/n)$ indexes instead of two (main and auxiliary)

- **Having multiple indexes** complicates the maintenance of collection-wide statistics
 - **Affects spelling correction** (multiple indexes + invalidation bit vector ... no more simple)
- **In reality, often a combination**
 - Frequent **incremental changes**
 - **Occasional complete rebuild** (**becomes harder with increasing size** – not clear if Google can do a complete rebuild) when query processing is switched to the new index and the old index deleted



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Compression

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Posting List Compression Part 1

Bhaskarjyoti Das

Department of Computer Science Engineering

- The postings file is much larger than the dictionary, **factor of at least 10**.
 - Key desideratum: **store each posting compactly**.
- A posting for our purpose 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 posting list will be concatenation of 20 bit docid**
- **Our goal:** use far fewer than 20 bits per docID.

Two conflicting forces : Rare Term vs. Common Term

- **Rare term** like *arachnocentric* occurs in maybe one doc out of a million – we would like to store this posting using $\log_2 1M \sim 20$ bits.
- A **common term** like *the* occurs in virtually every doc, so 20 bits/posting is too expensive.
- **20 bits design does not make sense for frequent term**
 - Say “the” appears in every one of 800k documents . So it is **800K*20**
- **0/1 bitmap vector ?**
 - If we use bit vector for “the”, it is just **800K** ! **Multiply with no of frequent terms ?**
- We want to compress posting list : common terms are stored using fewer than 20 bits and rare terms can be represented using 20 bits

Idea : Store the Delta



- Is there a scheme that works for both rare and common words ?
 - **Idea:** instead of storing docids, **we can store the “delta” (run-length encoding ?)**
- We store the list of docs containing a term in increasing order of docID.
 - **Original:** 33,47,154,159,202 ...
 - **Consequence:** it suffices to store *gaps*.
 - 33,14,107,5,43 ...
- **Assumption(Hope):** most gaps can be encoded/stored with far fewer than 20 bits.

Three Posting 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			

Gap for common words is less, for rare words it is more !!

- So now it is about how best to encode the gap!
- Aim:
 - For *arachnocratic*, we will use **~20 bits/gap** entry.
 - For *the*, we will use **~1 bit/gap** entry (i.e. small no of bits)
- Key challenge: encode every integer (gap) with about **as few bits as needed** for that integer.
- If the average gap for a term is G , we want to use $\sim \log_2 G$ bits/gap entry.
- **This requires a variable length encoding**
 - Variable length codes achieve this by using short codes for small numbers

Variable Byte code corresponds to integer number of bytes

- For a gap value G , we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- What we do if we need more than 1 byte ?
- Begin with **7 bits 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 other bytes $c = 0$.

Idea : Variable Gap Encoding for Rare vs. Common Terms

docIDs	824	829	215406
gaps		5	214577
VB code	<u>0000110</u> <u>10111000</u>	10000101	<u>00001101</u> <u>00001100</u> <u>10110001</u>

Postings stored as the byte concatenation

000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Compression

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Posting List Compression Part 2

Bhaskarjyoti Das

Department of Computer Science Engineering

Recap : Rare Term vs. Common Term



- **4 Bytes / docid** : For Reuters (**800,000 documents**), instead of **using 4 bytes** per docID, we can use **$\log_2 800,000 \approx 20$ bits** per docID.
- **Strategy 1 - 20 bits / docid**: We would like to store posting using $\log_2 800,000 \approx 20$ bits. **20 bits design does not make sense for frequent term** . Say “the “ appears in every one of 800k documents , it is **800K*20** . But makes sense for rare term.
- **Alternative : 0/1 bitmap vector of size 800K not an option** : Mostly empty for rare term
- **Strategy 2: Common** terms should spend less and **rare** terms can spend more
- **Strategy 3 : Store gap instead of docid** - Size of gaps mostly less than

- Instead of **bytes**, we can also use a different “**unit of alignment**”:
32 bits (**words**), 16 **bits**, 4 bits (**nibbles**).
- **Byte alignment wastes space** if you have **many small gaps**
– nibbles do better in such cases.
- But still variable byte codes:
 - Used by **many commercial/research** systems
 - Good **low-tech blend of**
 - **variable-length coding**
 - **sensitivity to computer memory alignment matches** (vs. bit-level codes, which we look at next).

Idea : 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, 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**
- 1110,101 - How to read this ?

Gamma Code Example

number	length	offset	γ -code
?	0		none
1	0		0
?	2	0	10,0
3	10	1	10,1
4	110	00	110,00
?	9	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001

Gamma Code Properties

- G is encoded using $2 \lfloor \log G \rfloor + 1$ bits
 - Length of “offset” is $\lfloor \log G \rfloor$ bits after stripping off 1 bit
 - Length of “length” is $\lfloor \log G \rfloor + 1$ bits after representing in Unary
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$. *It is great since most gaps are small*
- Gamma code is uniquely prefix-decodable, like VB
 - Decode from L to R (length field is sequence of 1 followed by 0; since no of 1 = length of offset, I can read the offset and will prepend 1 to it.)
Example for 9 [1001], gamma is 1110, 001
- Gamma code can be used for any distribution (rare or common)
- Gamma code is corpus parameter-free (recall we came up with 20 bit from $\log(800K)$ for RCV1 Corpus)

Gamma Seldom Used In Practice – Why ?



- Gamma Codes **are variable bit codes** !!
- 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 slow
 - **Variable byte encoding is aligned** and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Comparison for RCV1 Corpus

Data structure	Size in MB	
dictionary, fixed-width	11.2	← *
dictionary, term pointers into string	7.6	
with blocking, k = 4	7.1	
with blocking & front coding	5.9	←
RCV1 collection (text, xml markup etc)	3,600.0	
collection (text) 800K doc* 200 tokens * 6 bytes/token	960	← *
postings, uncompressed (32-bit words) (100 Million * 4 bytes)	400	
postings, uncompressed (20 bits) (100 Million * 20/8 bytes)	250	
postings, variable byte encoded	116	←
postings, g-encoded	101	

- We can now **create an index** that is very **space efficient by using compression**
- Only **4% of the total size** of the original text (including XML and markup) collection (**3.6 GB vs. 101 MB**)
- Only **10-15%** of the total size of the **text** in the collection (**960 MB vs. 101 MB**)
- However, **we've ignored positional information**
- Hence, **space savings are less for indexes used in practice**
 - **But techniques substantially the same.**



THANK
YOU

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Compression

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Dictionary Compression Part 1

Bhaskarjyoti Das

Department of Computer Science Engineering

Dictionary Compression

- The dictionary is **small compared to the postings file**.
- But we want to **keep it in memory**.
- Also: **competition with other applications**, cell phones, onboard computers, fast startup time
 - So compressing the dictionary is important.

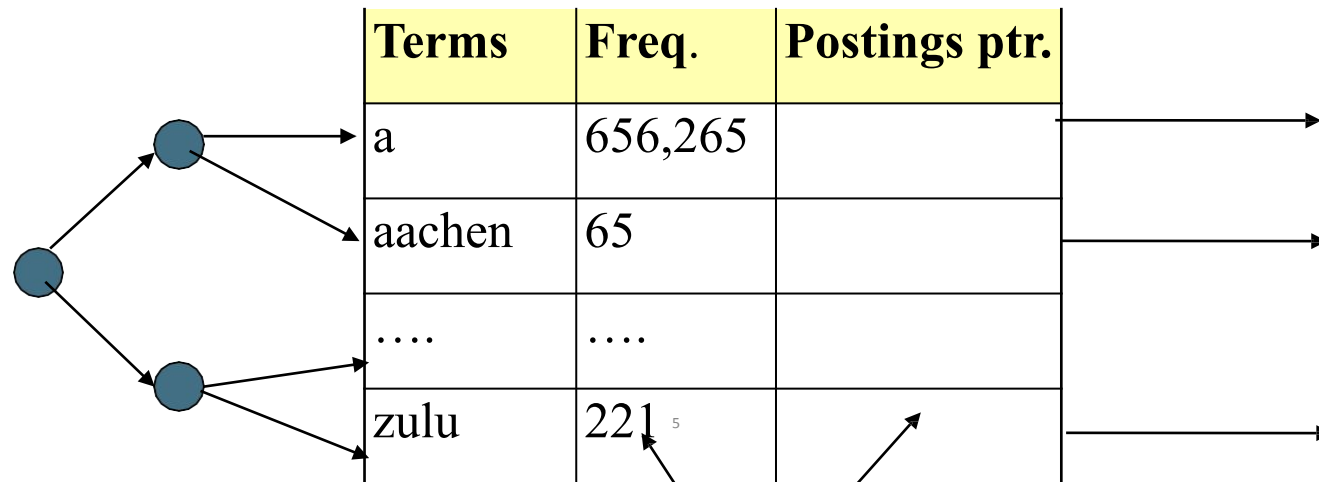
Dictionary as Array Of Fixed Width Entries

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

Space needed: 20 bytes 4 bytes[?] 4 bytes

Dictionary Storage First Cut

- Array of fixed-width entries
 - ~400,000 terms; 28 bytes/term = **11.2 MB**
 - **Dictionary size without compression !**



20 bytes

4 bytes each

Dictionary search
structure

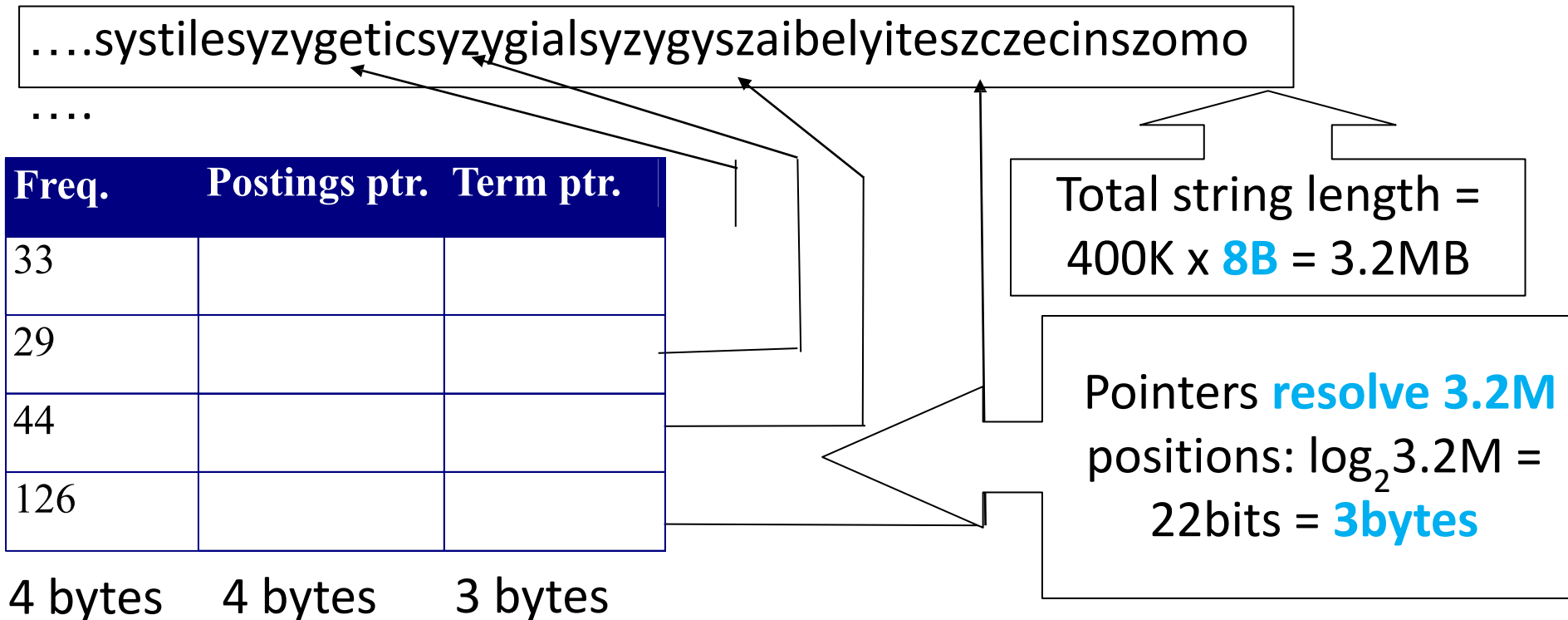
Fixed Width Terms are Wasteful



- 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?
- Short words dominate token counts but not type average.
- Ave. dictionary word in English: ~**8** characters
 - How do we use ~8 characters per dictionary term?

Compression Technique – 1 : Dictionary as 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.



Savings in Compression Technique – 1



1. **String** : Avg. 8 bytes per term in term string * 400k term = **3.2 MB** string size

2. **Freq + Posting Ptr + Term Ptr** :
 - 4 bytes per term for Freq.
 - 4 bytes per term for pointer to Postings.
 - 3 bytes per term pointer
 - 400K terms x 11 bytes for dictionary table = **4.4 MB**

3. **Total Dictionary size** : 4.4 MB + 3.2 MB = **7.6 MB** (against 11.2MB for fixed width)
 - **Previous** 20 bytes/term (28 bytes * 400K = **11.2 MB**)



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Index Compression

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

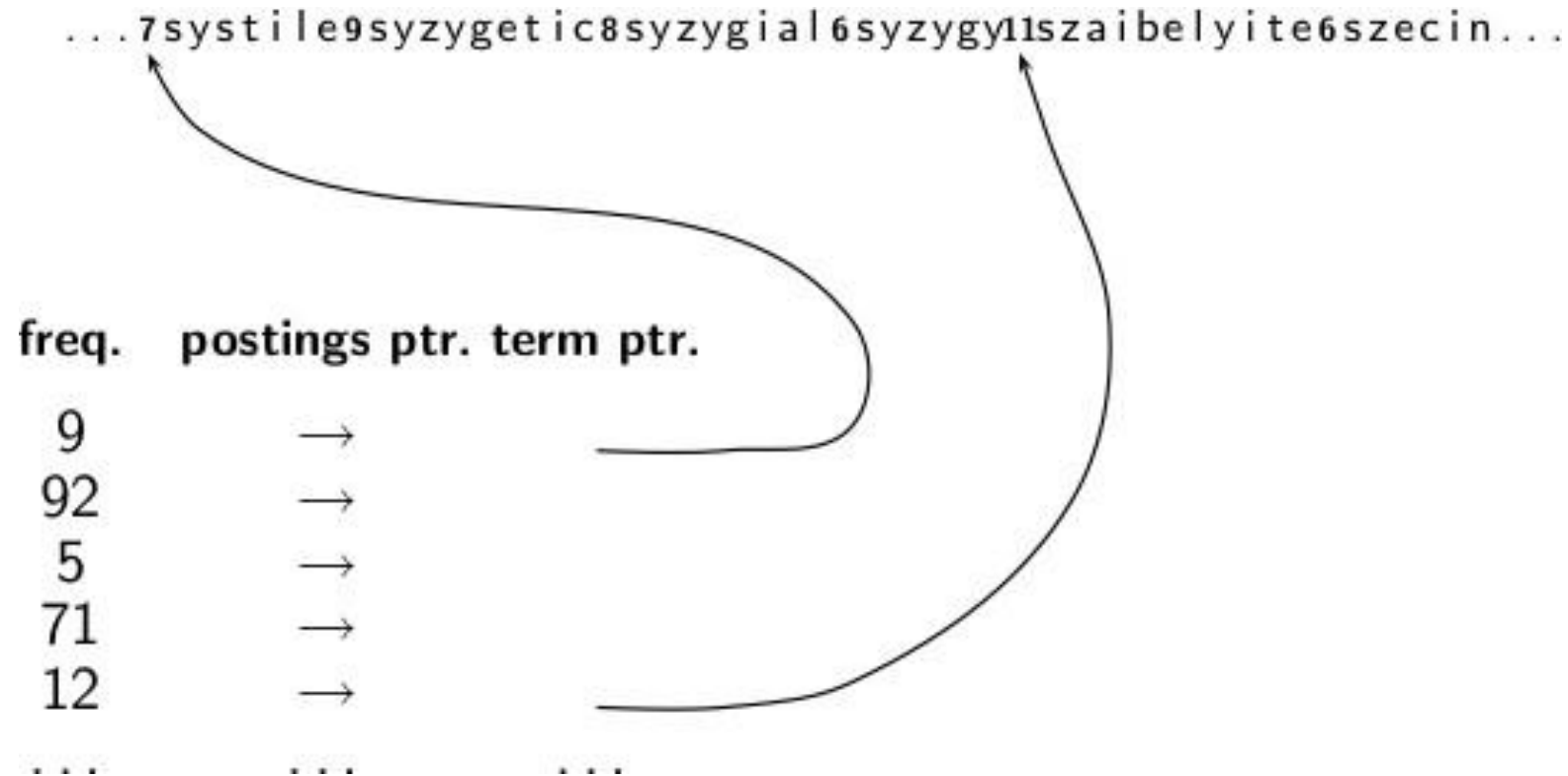


Dictionary Compression – Part 2

Bhaskarjyoti Das

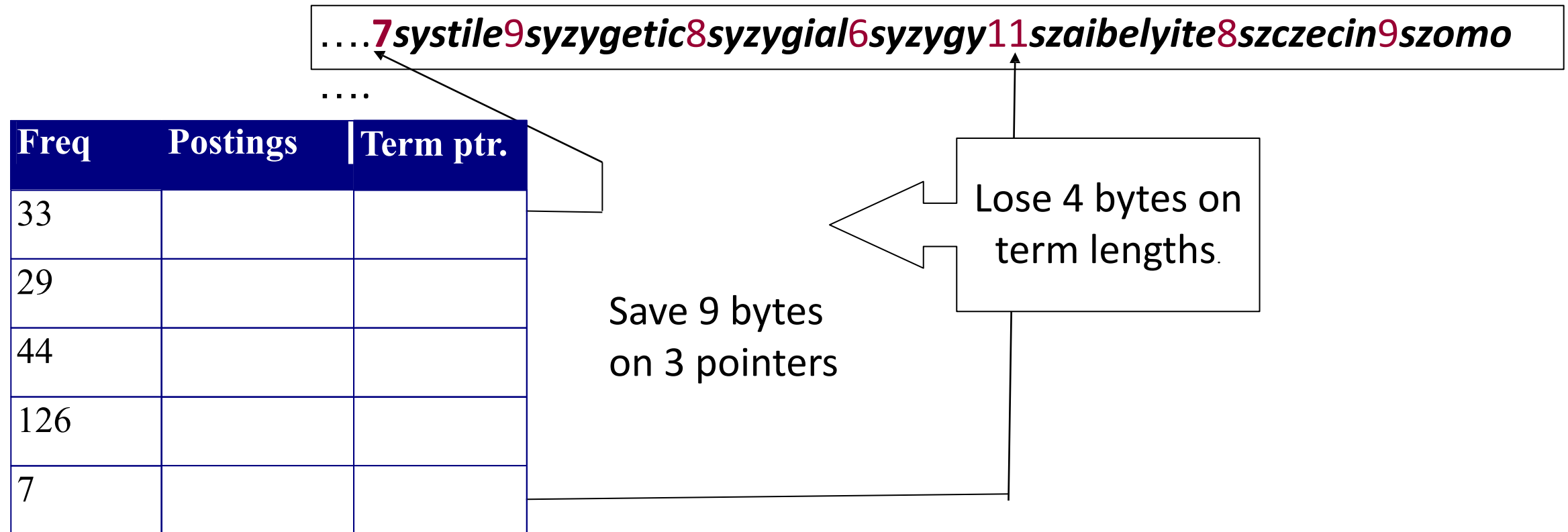
Department of Computer Science Engineering

Compression Technique 2 : Dictionary as a String with Blocking



Compression Technique 2 : Blocking to Save Space on Term Pointers

- Store pointers to **every k th term string** and saving on **$(k-1)$ terms**. Example below: $k=4$.
- Need to store term lengths (1 extra byte) in no of characters



Compression Technique 2 : Net Savings



Example for block size $k = 4$

▪ Previously without blocking for 400K terms, ~~terms~~ were using **$400K * 3 \text{ bytes} = 1.2 \text{ MB}$**

▪ **With blocking**, for 400K terms, it is now **$100K * 3 \text{ bytes} = 0.3 \text{ MB}$**

▪ **Additional space** used for this **$1 \text{ byte /term} = 400K * 1 \text{ byte} = 0.4 \text{ MB}$**

▪ **Net Savings** = **$1.2 \text{ MB} - 0.3 \text{ MB} - 0.4 \text{ MB} = 0.5 \text{ MB}$**

5

Shaved another $\sim 0.5 \text{ MB}$. This reduces the size of the dictionary from **7.6 MB to 7.1 MB** . We can save more with larger k

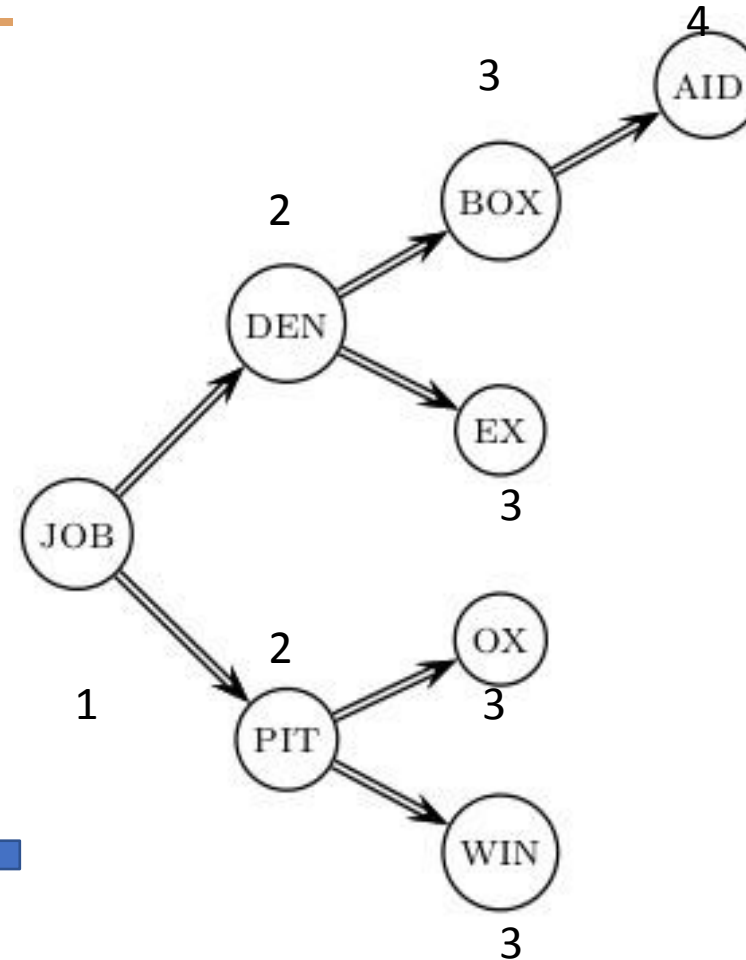
Why not go with larger K ?

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

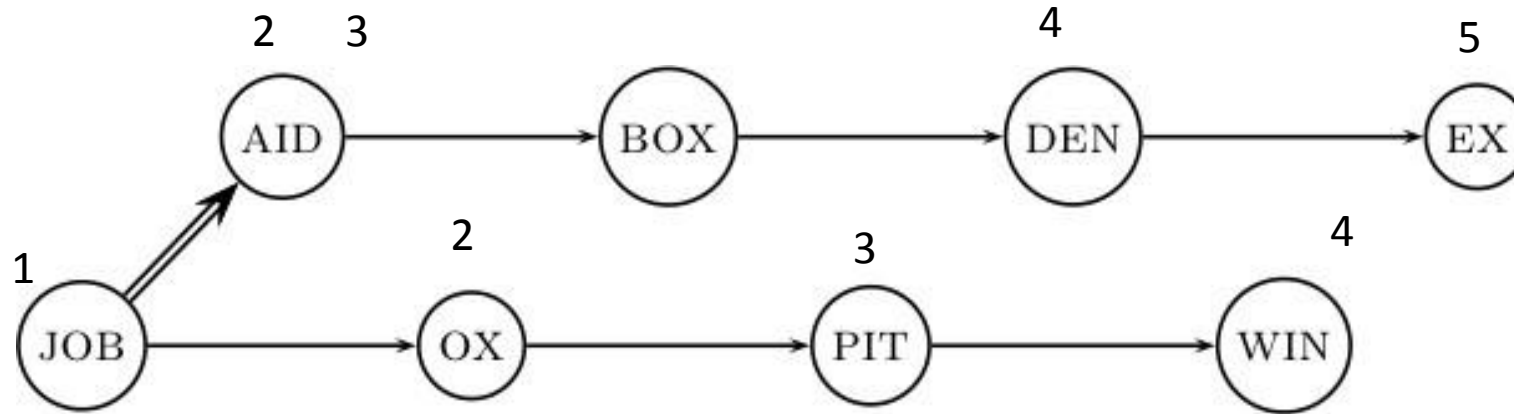
Effect on Computation Time : Lookup for a Term without Blocking

1. Since we are “binary searching” the dictionary, where the terms are stored in lexicographic order as an array of size 8
2. Assuming *each term is equally likely*, **average no of comparisons** =
 $(1+2*2+3*4+4)/8 = 2.6$

{
AID
BOX
DEN
EX
JOB
OX
PIT
WIN
}



Effect on Computation Time : Lookup for a Term with Blocking



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

▪ **only two pointers are involved** to get the corresponding strings “**3AID3BOX3DEN2EX**” and **2OX3PIT3WIN**”


▪ Blocks of 4 (binary tree),

▪ avg. = $(1+2 \cdot 2+2 \cdot 3+2 \cdot 4+5)/8 = 3$ compares

▪ **Saving on space but sacrificing on time !!**

Compression Technique 3 : Front Coding

▪ Front-coding:

- Can we save in the dictionary string ?
- **Sorted words** commonly have **long common prefix** – store differences only
 - (for last $k-1$ in a block of k)
8*automata***8***automate***9***automatic***10***automation*  *Previous design*

One block in blocked compression ($k = 4$) ..

8 a u t o m a t a **8** a u t o m a t e **9** a u t o m a t i c **10** a u t o m a t i o n
U

... further compressed with front coding.

8 a u t o m a t * a **1** ♦ e **2** ♦ i c **3** ♦ i o n **4**

Common prefix

Extra Length
Suffix

Data structure	size in MB
dictionary, fixed-width (baseline)	11.2
dictionary, term pointers into string (method 1)	7.6
~, with blocking, k = 4 (method 2)	7.1
~, with blocking & front coding (method 3)	5.9

Summary :

1. We have saved almost **half of the space** !!
2. Given a term in the dictionary, we have to **design an algorithm that will do additional decompression** in **addition to binary search**
3. This is a **lossless compression** (not losing any information here) !!



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Scoring, Term Weighting, Vector Space Model

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Scoring and Take 1 on Query Document Matching Score

Bhaskarjyoti Das

Department of Computer Science Engineering

Boolean Query– is it Good For Everybody ?



- Thus far, our queries have all been Boolean.
 - Documents **either match or don't**.
- **Good for expert users** with precise understanding of their needs and the collection.
- **Also good for applications:** Applications can easily consume 1000s of results.
- **Not good for the majority of users.**
 - Most users incapable of writing Boolean queries (or they are, but they think it's too much work).
- **Most users don't want to wade through 1000s of results.**
 - This is particularly true of web search.

Problem With Boolean Search – Feast or Famine ?



- Boolean queries often result in either **too few (=0) or too many (1000s) results**.
- Query 1: “*standard user dlink 650*” → 200,000 hits
- Query 2: “*standard user dlink 650 no card found*”: 0 hits
- It takes a **lot of skill to come up with a query** that produces a manageable number of hits.
 - **AND gives too few; OR gives too many**

Answer : Ranked Retrieval Models



1. An ordered retrieval : rather than a set of documents satisfying a query expression, the system returns **an ordering over the (top) documents** in the collection for a query
2. 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**

In principle, **there are two separate choices here**, but in practice, **ranked retrieval** has normally been associated with **free text queries and vice versa**



- When a **system produces a ranked result set, large result sets are not an issue**
 - Indeed, the size of the result set is not an issue
 - **We just show the top k** (≈ 10) results
 - We don't overwhelm the user
- **Premise**: the ranking algorithm works

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 for a Single Term Query



We need a way of assigning a score to a query/document pair

1. Let's start with a **one-term query**

- 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)**

We will look at a number of alternatives for this....

Scoring Strategies Heads Up



Take 1: Consider both query and document as a **set of terms** (not bag !!) and calculate similarity

Take 2: Consider both query and document as a **bag of words** (not set !!), calculate their similarity or distance in a vector space model

- **Question :**

- are all terms considered equal ?
- how do we aggregate the contribution of all terms in a query or a document ?

Take 1 : Jaccard Coefficient



- Recall from Lecture 3: A commonly used measure of overlap of two sets A and B
- $\text{jaccard}(A, B) = |A \cap B| / |A \cup B|$
- $\text{jaccard}(A, A) = 1$
- $\text{jaccard}(A, B) = 0$ if $A \cap B = 0$
- **Good News:**
 - A and B don't have to be the same size.
 - Always assigns a number between 0 and 1.

Jaccard Coefficient : Scoring Example



- What is the **query-document match score** that the Jaccard coefficient computes for each of the two documents below?
 - Query: *ides of march*
 - Document 1: *caesar died in march*
 - Document 2: *the long march* ✓

Issue : Jaccard Coefficient biased against longer sentence ?

Issues with Jaccard Coefficient for Scoring



1. It doesn't consider *term frequency* (how many times a term occurs in a document)
2. **Rare terms** in a collection are more informative than frequent terms. Jaccard doesn't consider this information
3. We also need a more sophisticated way of **normalizing for length** (previous slide)



THANK
YOU

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Scoring, Term Weighting, Vector Space Model

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Term Weighting : Towards Take 2 on Query Document Matching Score

Bhaskarjyoti Das

Department of Computer Science Engineering

Recall : Binary Term Document Incidence Matrix



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

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

Term Document Count Matrices

- Consider the **number of occurrences** of a term in a document:
 - Each document is a count vector** in \mathbb{N}^V : a column below



	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

Bag of Words Model – Vector Representation



- bag of words represented as vector
- Vector representation doesn't consider the ordering of words in a document
 - *John is quicker than Mary and Mary is quicker than John*
have the same vectors
 - In a sense, this is a step back: The positional index was able to distinguish these two documents.
 - We will look at “recovering” positional information later in this course. There are alternative approaches like “language model” that can address this issue
- For now: bag of words model

Handling Frequency : 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.**

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 :**
- 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.

Handling Rare Terms : 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*.

Frequent Terms vs. Rare Terms



1. Frequent terms are less informative than rare terms
2. 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**.
3. For frequent terms, w
 - We want **high positive weights** for words like *high, increase, and line*
 - **But lower weights than for rare terms**.
4. We will use document frequency (**df**) to capture this.

- 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
 - **$df_t \leq N$**
- We define the idf (inverse document frequency) of t by

- We use **$\log(N/df_t)$** instead of N/df_t to “**dampen**” the effect of idf.

$$idf_t = \log_{10}(N/df_t)$$

base of the log is immaterial.

Example of idf for N = 1 Million

$$\text{idf}_t = \log_{10} \frac{1,000,000}{\text{df}_t}$$

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1000	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.

Combining the Factors : 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
 - ...

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth ...
ANTHONY	5.25	3.18	0.0	0.0	0.0	0.35
BRUTUS	1.21	6.10	0.0	1.0	0.0	0.0
CAESAR	8.59	2.54	0.0	1.51	0.25	0.0
CALPURNIA	0.0	1.54	0.0	0.0	0.0	0.0
CLEOPATRA	2.85	0.0	0.0	0.0	0.0	0.0
MERCY	1.51	0.0	1.90	0.12	5.25	0.88
WORSER	1.37	0.0	0.11	4.15	0.25	1.95
...						

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

Effect of idf on Ranking



- **Rarity** is a **relative** phenomena
- 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.

- Example:

Word	Collection frequency	Document frequency
<i>insurance</i>	10440	3997
<i>try</i>	10422	8760

Q: **Which word is a better search term** (and should get a higher weight)?

- Both are appearing almost same no of times in collection but “insurance” is a rarer word from document perspective !! Looks like it is appearing many times in insurance related documents !!

Store TF IDF To Calculate Score Of Document for a Query ?



- You can have more than one design for the Index to store the tf-idf score .
- **Dictionary** has the **terms**
 - can also have the **inverse document frequency(idf)** score for a **particular term** !
- **Posting List** has **docids** for a **particular term**
 - can also store the term specific **term frequency (tf)**
 - Can also store the **final tf-idf score for “term-docid” pair**
- We will discuss the algorithm for tf-idf calculation later ..



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Scoring, Term Weighting, Vector Space Model

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Vector Space Model for Scoring: how to Measure the Distance between Document Vectors ?

Bhaskarjyoti Das

Department of Computer Science Engineering

- So we have a **$|V|$ -dimensional vector space**
 - Terms are **axes of the** space
 - Documents are **points or vectors** in this space
 - **Very high-dimensional:** tens of millions of dimensions when you apply this to a web search engine
 - **These are very sparse vectors** - most entries are zero.

Queries as Vectors



- Key idea 1: Do the **same for queries i.e.** represent them as vectors in the space
- Key idea 2: Rank documents **according to their proximity** to the query in this space

Recall: We want to get away from the **you're-either-in-or-out Boolean model.**

Instead: rank **more relevant documents higher** than less relevant documents

proximity = similarity of vectors \approx inverse of distance

Formalizing Vector Space Proximity



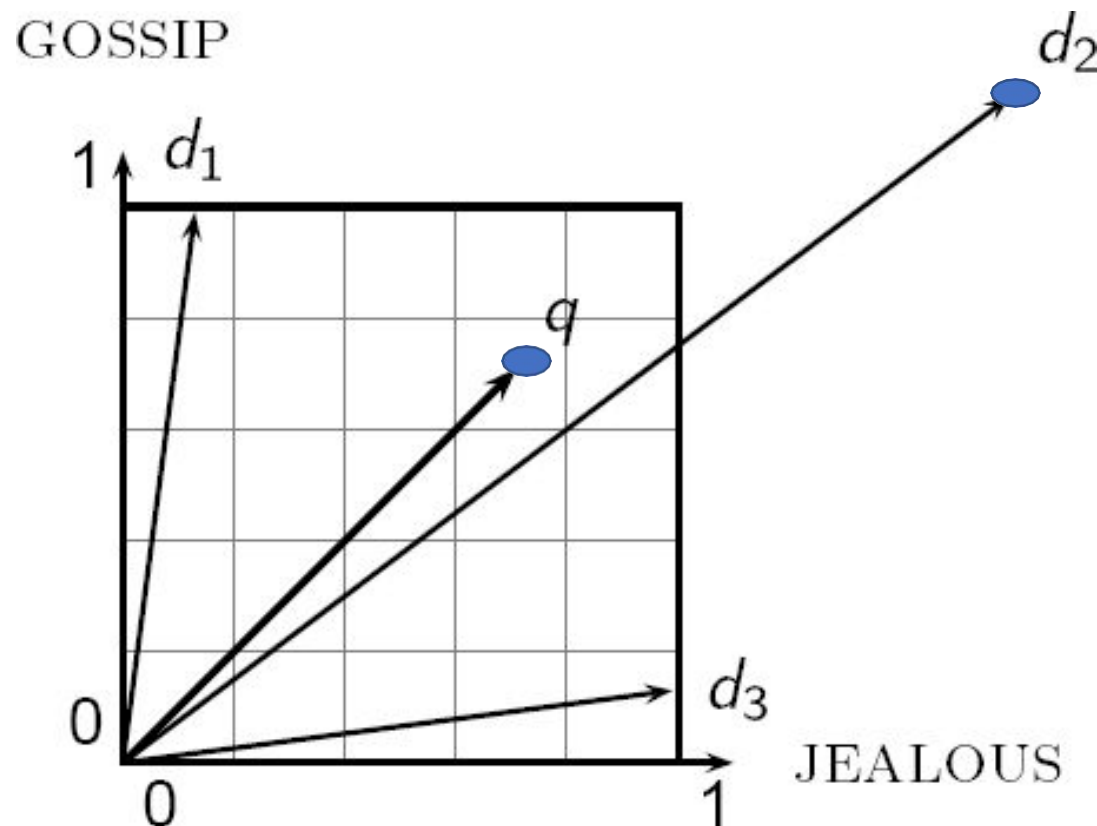
- **First cut:** distance between two points (=2 vectors)
 - (= distance between the end points of the two vectors)
- **Euclidean distance?**
- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is large for vectors of different lengths but they contain similar information with different intensity /magnitude

Why Distance is a Bad Idea ?

The Euclidean distance (that considers the end points of two vectors) 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.



Use Angle instead of Distance

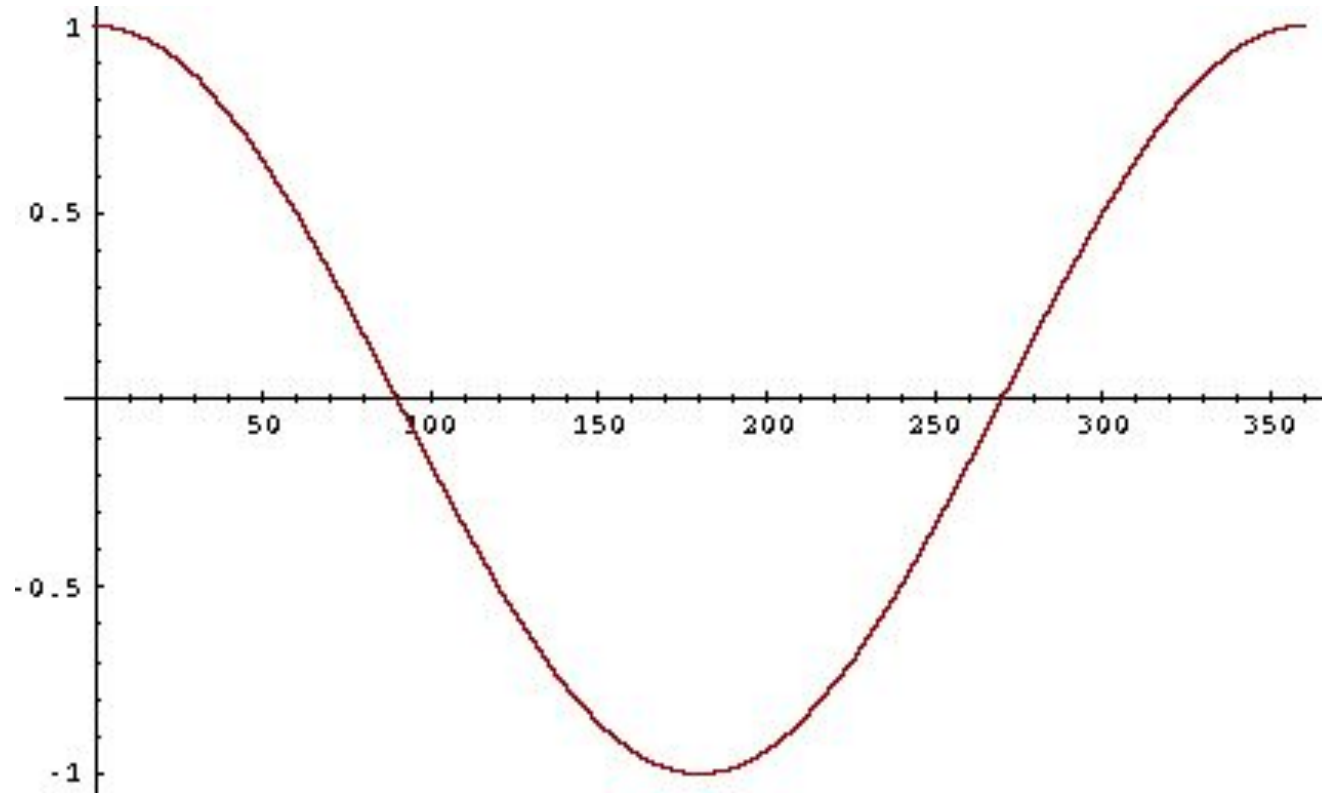


Example: take a document d and append it to itself. Call this document d' .

- “Semantically” d and d' have the same content but **Euclidean distance between the two documents can be quite large**
- **But the angle between the two documents is 0 !**
- **Key idea:** Rank documents according to angle with query

- Instead of angle, we can use the cosine of the angle
- Cosine is a monotonically decreasing function for the interval $[0^\circ, 180^\circ]$
- The following **two notions are equivalent**.
 - Rank documents in decreasing order of the angle between query and document
 - Rank documents in increasing order of cosine(query,document)

From Angles to Cosines



- But how – *and why* – should we be computing cosines?

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:

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

- 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
 - Same content but different intensity – normalized !

Cosine (Query, Document)

Dot product

Unit vectors

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

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

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

Cosine for Length Normalized Vectors



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

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

for q, d length-normalized.

Point to note



1. Cosine Similarity between two document vectors = dot product (or scalar product) of length-normalized vectors
2. It gives you similarity of content but loses the intensity of it ;-
 - Hero to Heroine in favorite **Movie1** : “I love you!”
 - Hero to Heroine in favorite **Movie2** : “I love you , I love you , I love you!”
 - **Query** : “love”
 - Both **Movie 1 and Movie 2 will have same cosine similarity. Preferred answer is Movie2** to do justice to the intensity !
- No worry .. **we take care in the way we design the document vector (tf-idf)**
-



THANK
~~YOU~~

Bhaskarjyoti Das

Department of Computer Science Engineering



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Scoring, Term Weighting, Vector Space Model

Bhaskarjyoti Das

Department of Computer Science Engineering

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB



Vector Space Model for Scoring: the tf-idf Algorithm

Bhaskarjyoti Das

Department of Computer Science Engineering

Cosine Similarity Among Three Documents



How similar are the novels ?

- **SaS**: *Sense and Sensibility*
- **PaP**: *Pride and Prejudice*
- **WH**: *Wuthering Heights?*

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6
wuthering	0	0	38

Term frequencies (counts)

Note: To simplify this example, we don't do idf weighting.

Three Documents Example



Log frequency weighting

Assume $\text{idf} = 1$

term	SaS	PaP	WH
affection	3.06	2.76	2.30
jealous	2.00	1.85	2.04
gossip	1.30	0	1.78
wuthering	0	0	2.58

After length normalization

term	SaS	PaP	WH
affection	0.789	0.832	0.524
jealous	0.515	0.555	0.465
gossip	0.335	0	0.405
wuthering	0	0	0.588

$\cos(\text{SaS}, \text{PaP}) \approx$

$0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0 \approx$

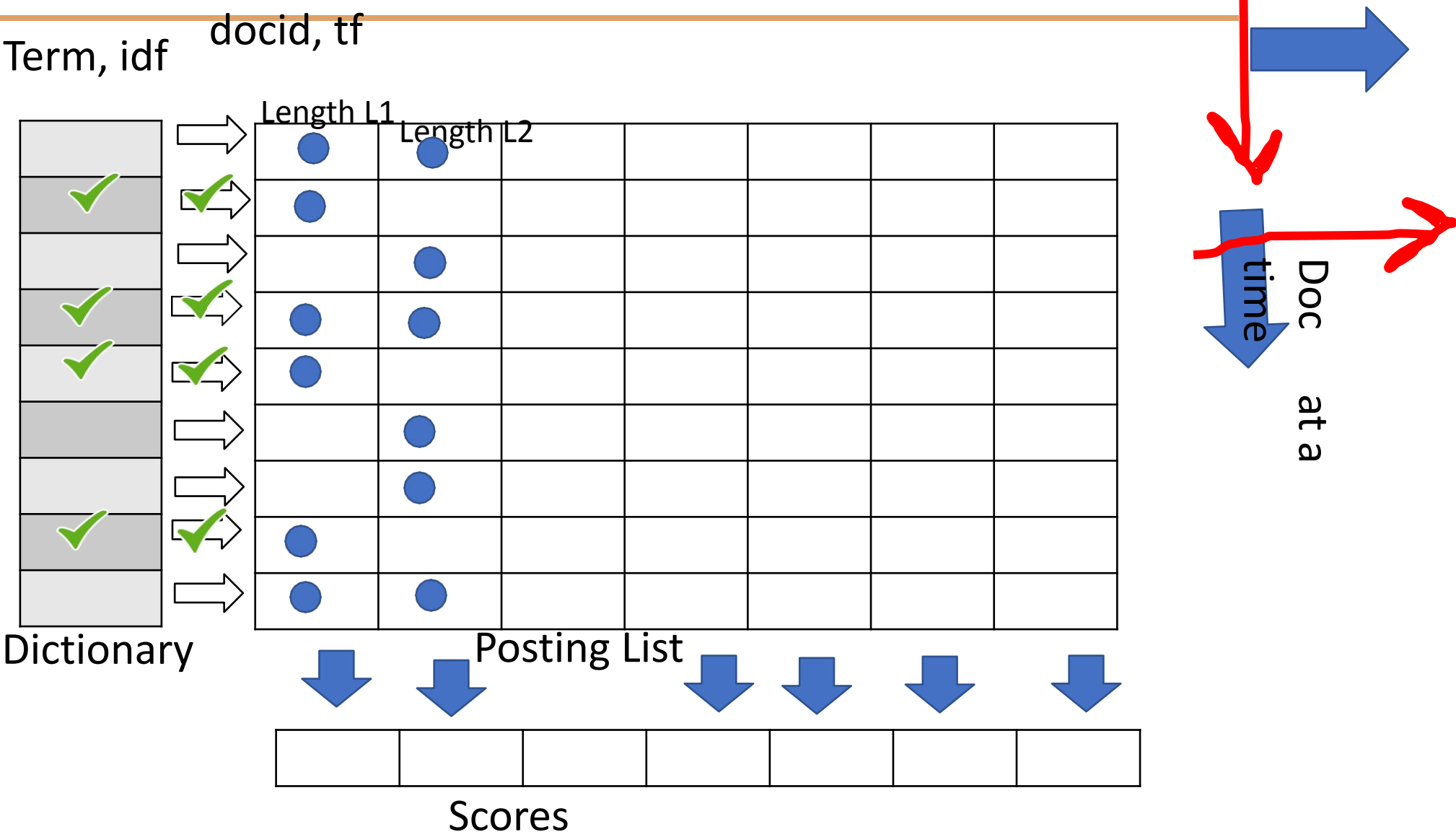
0.94

$\cos(\text{SaS}, \text{WH}) \approx 0.79$; $\cos(\text{PaP}, \text{WH}) \approx 0.69$

Why do we have $\cos(\text{SaS}, \text{PaP}) > \cos(\text{SaS}, \text{WH})$?

Jane Austen (not Emily Bronte): similar vocabulary !

Computing Cosine Scores



Computing Cosine Scores



COSINESCORE(q)

```
1  float Scores[ $N$ ] = 0
2  float Length[ $N$ ]
3  for each query term  $t$ 
4  do calculate  $w_{t,q}$  and fetch postings list for  $t$ 
5      for each pair( $d, tf_{t,d}$ ) in postings list
6      do Scores[ $d$ ] + =  $w_{t,d} \times w_{t,q}$ 
7  Read the array Length
8  for each  $d$ 
9  do Scores[ $d$ ] = Scores[ $d$ ] / Length[ $d$ ]
10 return Top  $K$  components of Scores[]
```


Many Variants for “tf-idf”

Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \dots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N - df_t}{df_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^\alpha$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}_{t \in d}(tf_{t,d}))}$				

Columns headed ‘n’ are acronyms for weight schemes.

Weighting may Differ in Query vs. Documents



1. 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.qqq*, using the acronyms from the previous table
2. A very standard weighting scheme is: Inc.ltc

Summary – Vector Space Ranking



1. Represent the query as a weighted tf-idf vector
2. Represent each document as a weighted tf-idf vector
3. Compute the cosine similarity score for the query vector and each document vector
4. Rank documents with respect to the query by score
5. Return the top K (e.g., $K = 10$) to the user



**THANK
YOU**

Bhaskarjyoti Das

Department of Computer Science Engineering