Text Processing and Indexing

Sunil Regmi
Lecturer, DoAI
Kathmandu University, Kavre, Nepal

Outline

2. Text Processing and Indexing [5 Hours]

2.1. Text preprocessing (tokenization, stemming, lemmatization)

2.2. Inverted index and other indexing structures

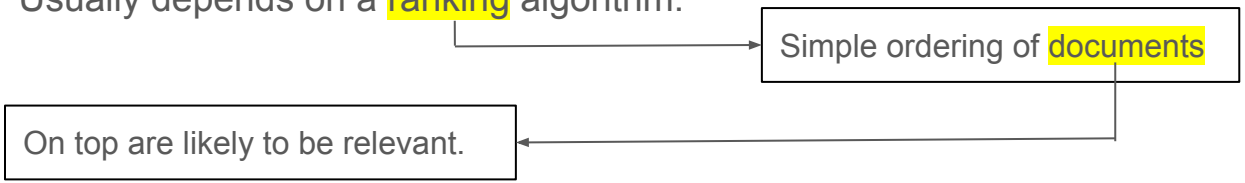
2.3. Compression techniques for IR

Traditional IR

- Adopts index terms to index and retrieve documents
- Works on a fundamental idea that the semantics of the documents and of the user information need can be naturally expressed through the sets of index terms.
- But a lot of semantics will be lost.

One central problem

- Issue of predicting which documents are relevant and which are not.
  - Usually depends on a ranking algorithm.



...

- Different set of premises yield distinct IR models.
- IR models adopted determines the prediction of what relevant is and what is not relevant.

The user Task

- In IR’s users express their information needs through queries using specific keywords.
- Data retrieval systems involve specifying constraints using query expressions like regular expressions.
- Users typically search for useful information by executing a retrieval task.

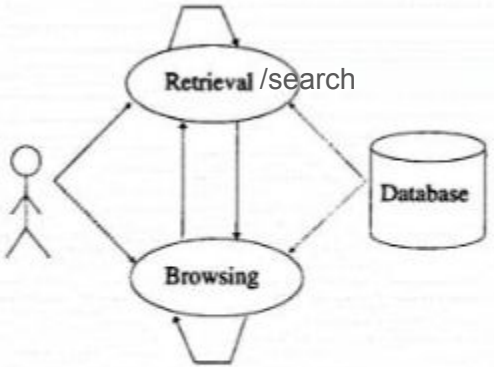
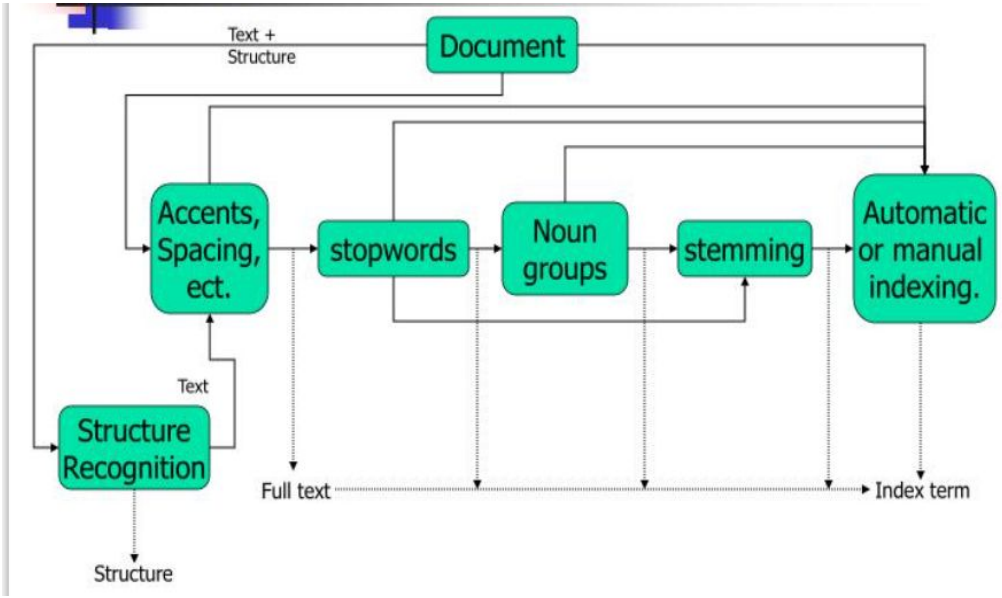


Figure 1.1 Interaction of the user with the retrieval system through distinct tasks.

- When the user’s interest is broad or poorly defined (e.g., "car racing"), they might browse documents rather than search for specific information (tourism).
- Browsing involves exploring related topics, where the user’s focus may shift during interaction (e.g., from car racing to tourism).
- Browsing is still a form of information retrieval but without a clearly defined objective from the beginning.

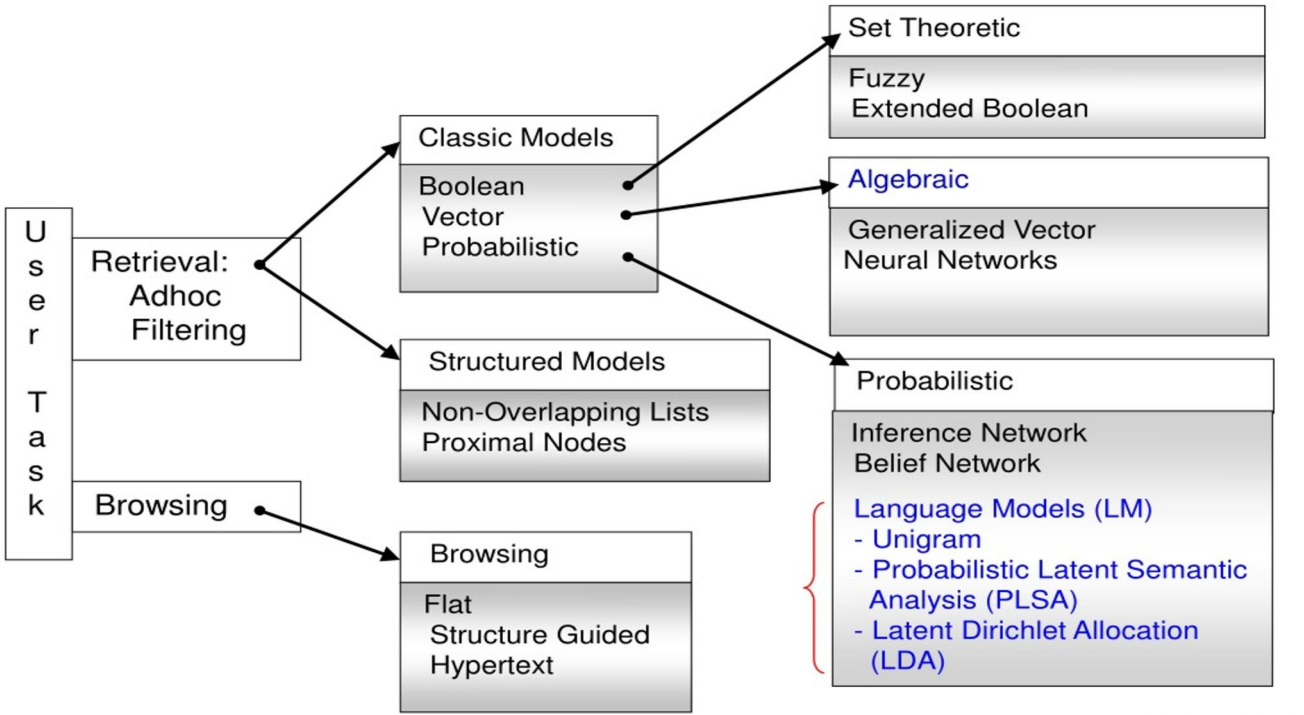
Logical View of Documents

- Documents can be represented as:
  - Full set of keywords or indexing terms
  - Full text



- With a very large collections,
  However modern computer might have to reduce set of index terms.
  - This reduction can be achieved by text-processing.

Taxonomy of Classic IR models



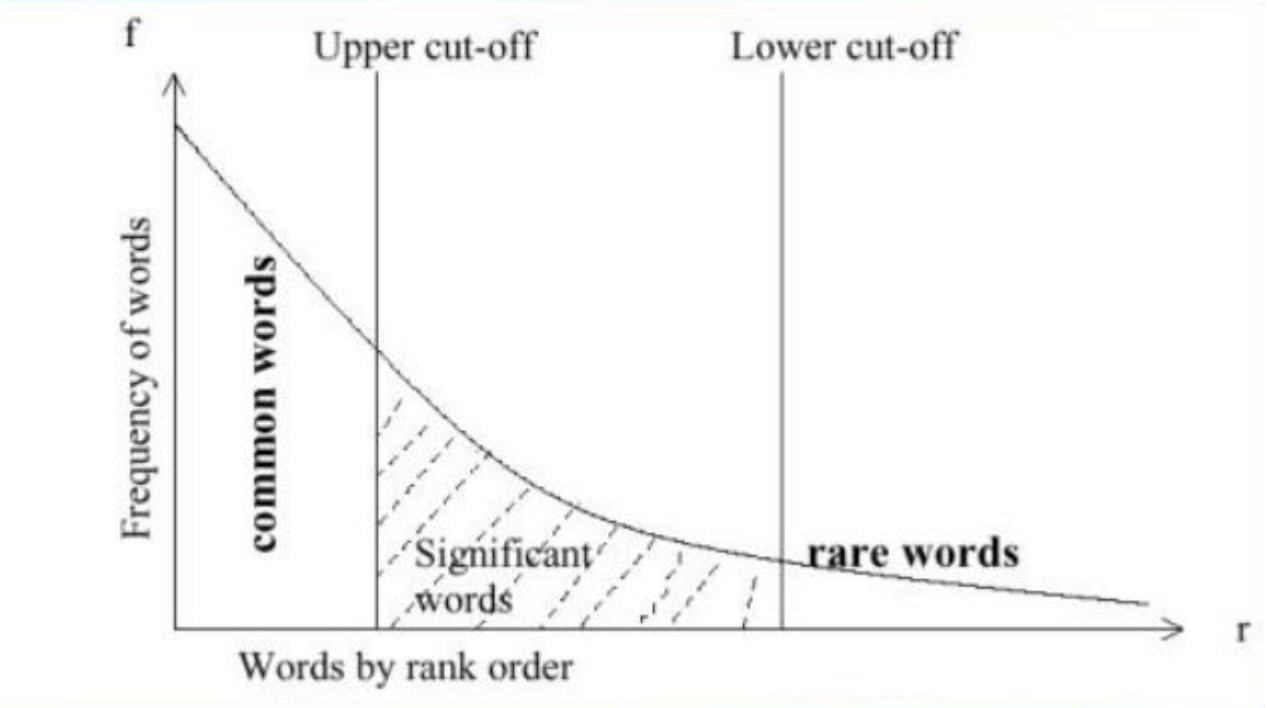
Text Processing

- Text processing for information retrieval (IR) involves transforming and analyzing textual data to retrieve relevant information in response to user queries.

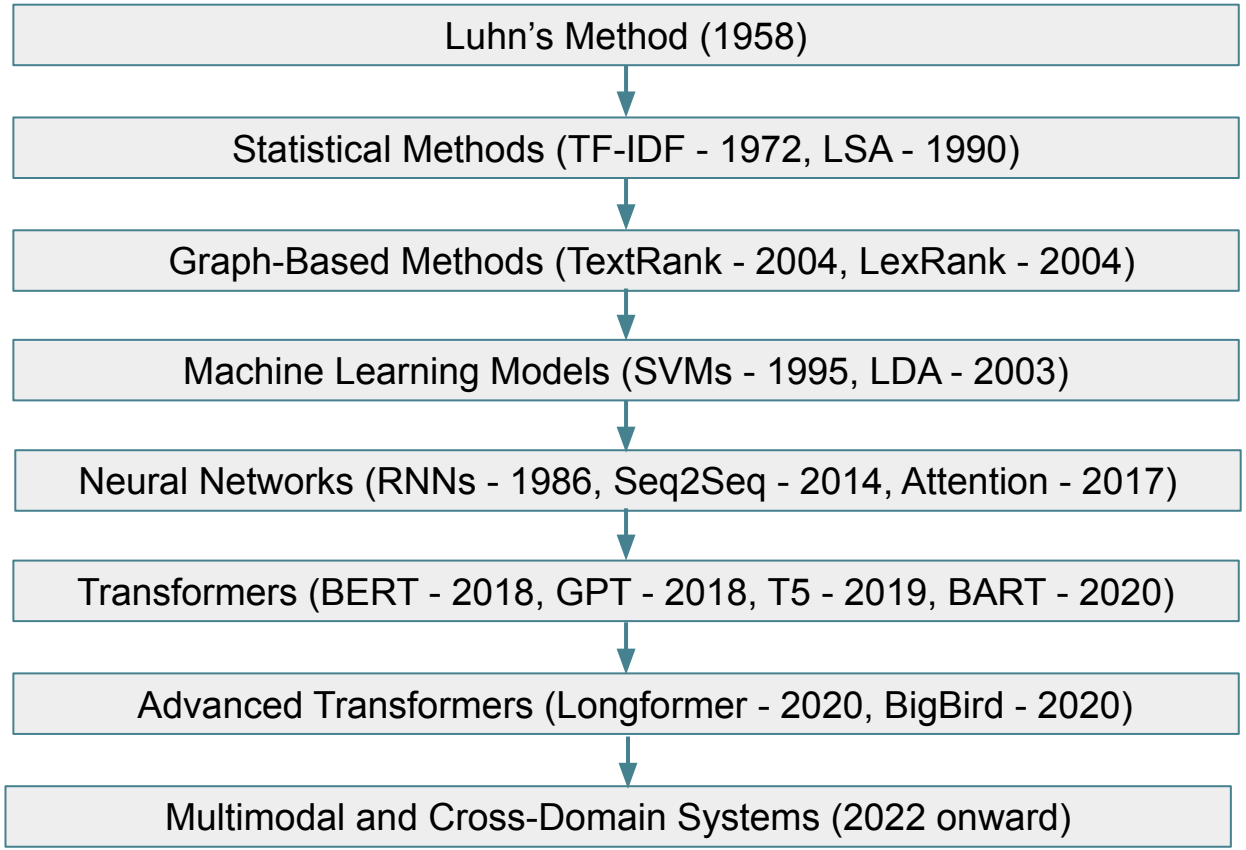
- It is a critical aspect of search engines, recommendation systems, and document retrieval applications.

- Before indexing or searching, raw text is cleaned and standardized through several preprocessing steps.

Luhn's idea


f
Upper cut-off
Lower cut-off
Frequency of words
common words
Significant words
rare words
r
Words by rank order

timeline of key developments after Luhn’s work



...

Tokenization: Splitting text into individual words, phrases, or sentences.

Lowercasing: Converting all text to lowercase to avoid case sensitivity issues. (might not needed for all languages)

Stopword Removal: Eliminating common words like "and," "the," and "is," which carry little informational value.

Stemming/Lemmatization: Reducing words to their base or root form (e.g., "running" becomes "run").

Punctuation and Special Character Removal: Cleaning irrelevant symbols from text.

1. Tokenization

- Tokenization is the process of splitting text into smaller pieces, typically words or sentences.
- In our case, we'll focus on word tokenization, which breaks a sentence into individual words.
- Tokenization is essential for indexing and searching in systems that store and retrieve information efficiently based on words or phrases.

Tokenization: Concept

- Breaking text into tokens (words, subwords, sentences)
- First step in every NLP pipeline

English Example:

Text: "Information Retrieval systems use indexing."

→ ['Information', 'Retrieval', 'systems', 'use', 'indexing']

Nepali Example:

पाठ: "नेपालको मौसम सुन्दर छ।" → ["नेपालको", "मौसम", "सुन्दर", "छ"]

a. Word Tokenization

text = "Text preprocessing includes tasks like tokenization, stemming, and lemmatization."

Tokens: ['text', 'preprocessing', 'includes', 'tasks', 'like', 'tokenization', ',', 'stemming', ',', 'and', 'lemmatization', '.']

In this step, the sentence is split into individual words, including punctuation marks.

b. Sentence tokenization

Input: "Tokenization is an important NLP task. It helps break down text into smaller units."

Output: ["Tokenization is an important NLP task.", "It helps break down text into smaller units."]

b. Character tokenization

Input: "Tokenization"

Output: ["T", "o", "k", "e", "n", "i", "z", "a", "t", "i", "o", "n"]

2. Stopword Removal

- Stopwords are common words like "and," "is," and "the," which usually don't carry significant meaning.
- Removing them helps reduce noise in text processing.

Tokens: ['text', 'preprocessing', 'includes', 'tasks', 'like', 'tokenization', ',', 'stemming', ',', 'and', 'lemmatization', '.']

Filtered Tokens: ['text', 'preprocessing', 'includes', 'tasks', 'like', 'tokenization', ',', 'stemming', ',', 'lemmatization', '.']

3. Stemming

- Stemming reduces words to their root forms by chopping off prefixes and suffixes.
- It’s a crude method that doesn’t always produce valid words (e.g., "running" becomes "run").

Filtered Tokens: ['text', 'preprocessing', 'includes', 'tasks', 'like', 'tokenization', ',', 'stemming', ',', 'lemmatization', '.']

Stemmed Tokens: ['text', 'preprocess', 'includ', 'task', 'like', 'token', ',', 'stem', ',', 'lemmat', '.']

4. Lemmatization

- Lemmatization is a more sophisticated approach than stemming.
- It reduces words to their base form (lemma) while ensuring that the resulting word is meaningful.
- Lemmatization uses the context (e.g., part of speech) to return correct forms like "better" to "good."

Stemmed Tokens: ['text', 'preprocess', 'includ', 'task', 'like', 'token', ',', 'stem', ',', 'lemmat', '.']

Lemmatized Tokens: ['text', 'preprocessing', 'include', 'task', 'like', 'tokenization', ',', 'stemming', ',', 'lemmatization', '.']

More on

https://www.geeksforgeeks.org/python-lemmatization-approaches-with-examples/

Indexing

Indexing is a critical component of Information Retrieval (IR) systems.

It allows efficient searching, retrieval, and organization of large amounts of textual data.

Let's explore some popular indexing structures,

- including the inverted index and others like B-trees, k-d trees, R-trees, and suffix trees.

Forward Index

In contrast to inverted indexing, forward indexing is not a common term used in database systems. It refers to the straightforward organization of documents or records in a sequential manner. Each document or record is stored in a separate location, and the index (if any) simply points to the location of each document. It allows for direct retrieval of documents based on unique IDs or keys.

Inverted Index

Maps terms to the documents they appear in. It is widely used in modern search engines like Elasticsearch for efficient full-text search and retrieval.

Suppose you have a document management system that stores files, and each file is given a unique document ID. The files are stored as individual documents with their content. The system maintains a forward index, which is a mapping of document IDs to the locations where the files are stored on disk.

Example Forward Index:

Document ID: 1001, Location: /data/files/file_1001.txt
Document ID: 1002, Location: /data/files/file_1002.txt
Document ID: 1003, Location: /data/files/file_1003.txt

Inverted index

- most common indexing structure used in text search engines like Google.

- It maps terms (words) to their locations (documents or positions within a document).

- An inverted index is essentially a dictionary where each term points to a list of documents that contain the term.

Structure: An inverted index typically has two main components:

- **Vocabulary:** A list of all unique terms in the corpus.
- **Posting List:** For each term in the vocabulary, a list of documents or positions where the term appears.

...

- Is a fundamental data structure used to efficiently store and retrieve information from a large collection of documents
- Backbone of elasticsearch powerful and fast full-text search capabilities.
- Instead of scanning through each document one by one, which can be time consuming for large datasets,
    - Inverted index allows elasticsearch to perform searches much more efficiently.
- Elasticsearch can quickly look up the term in the inverted index and retrieve documents id’s which correspond to the documents containing the term.

Example: For a collection of documents:

Doc1: "Information retrieval and indexing are essential."

Doc2: "Indexing techniques improve retrieval performance."

Term	Posting List
information	[Doc1]
retrieval	[Doc1, Doc2]
indexing	[Doc1, Doc2]
are	[Doc1]
essential	[Doc1]
techniques	[Doc2]
improve	[Doc2]
performance	[Doc2]

The posting list tells us which documents contain each term, enabling efficient searching.

...

There are two types of inverted indexes:

Record-Level Inverted Index: Record Level Inverted Index contains a list of references to documents for each word.

Word-Level Inverted Index: Word Level Inverted Index additionally contains the positions of each word within a document.

- Offers more functionality but needs more processing power and space to be created.

Example…

Documents			Terms			Index	
DocId	Docs		TermId	Term		Terms	Docs
1	To be or not to be		1	be		1	1:2:[2,6], 2:1:[2], 3:1:[3]
2	To be right		2	left		2	3:1:[4]
3	Not to be left		3	not		3	1:1:[4], 3:1:[1]
			4	or		4	1:1:[3]
			5	right		5	2:1:[3]
			6	to		6	1:2:[1,5], 2:1:[1], 3:1:[2]

The first element of this list is 1:2:[2,6],

the term occurs in document 1, with frequency 2, at positions 2 and 6.

How Is a Search Query Executed?

When a search query is executed, the query is first tokenized, and the individual terms are looked up in the inverted index.

For each term, the index returns a list of documents that contain the term, along with information about the term's frequency and position within each document.

These lists are then combined and ranked according to a relevance score,

- which considers factors such as term frequency, document length, and the proximity of the terms within the document.

Proximity is the search technique used to find two words next to, near, or within a specified distance of each other within a **document**.

The highest-ranked documents are then returned as search results:

Original Query	To do right
Query terms	to, do, right

Term	TermId	Docs from index
do	na	na
to	1	1:2:[2,6], 2:1:[2], 3:1:[3]
right	5	2:1:[3]

Results:
DocId=1; To be or not to be
DocId=2; To be right
DocId=3; Not to be left

Search Time Complexity:

Best Case: O(1) (for direct access to the posting list of a term).

Worst Case: O(N) (if searching across multiple documents for uncommon terms).

Space Complexity:

O(T + D + P):

T: Number of unique terms.

D: Number of documents.

P: Number of postings (term-document pairs).

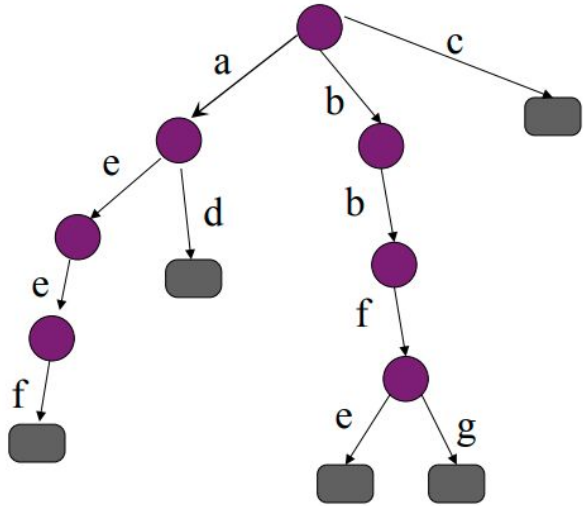
- The inverted index maintains a dictionary where each term maps to a list of documents (posting list).
- Searching for a term is generally O(1) if you know the term directly.
- However, the space complexity grows with the number of unique terms, documents, and postings.

Suffix Trees and Suffix array

Trie

A tree representing a set of strings.

{aeef, ad, bbfe, bbfg, c}

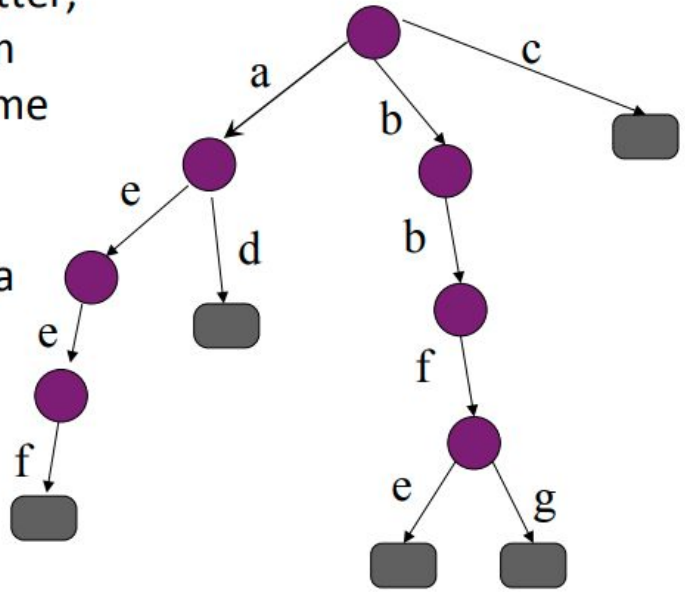


Trie

Assume no string is a prefix of another

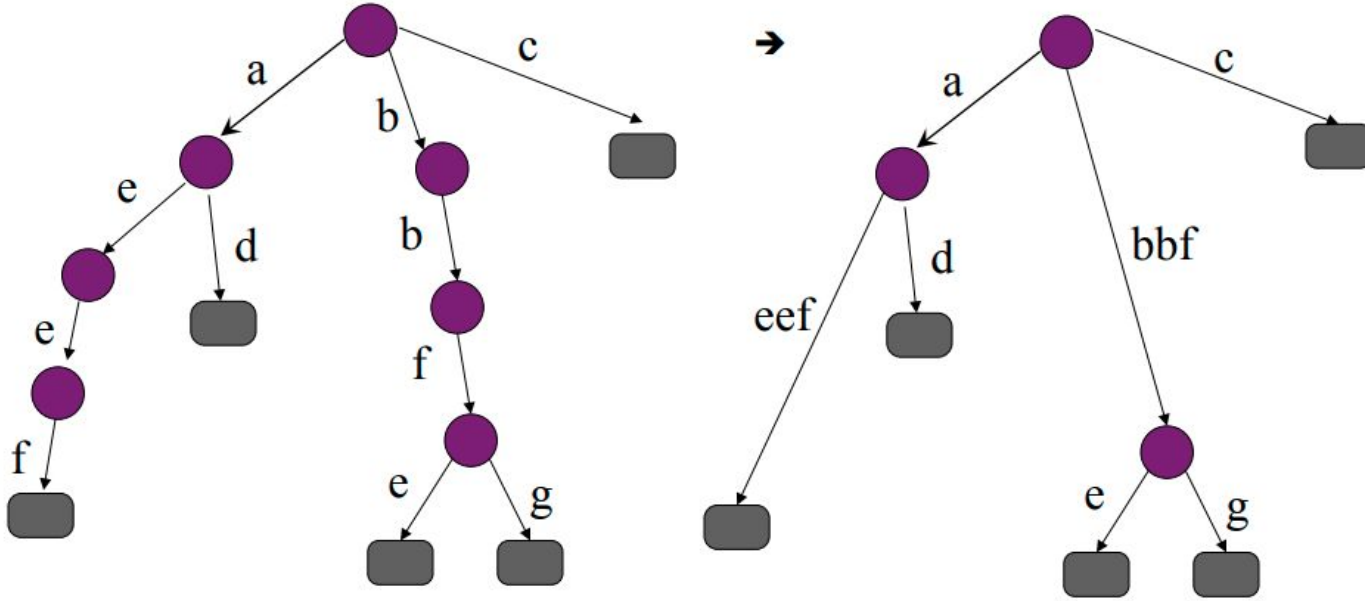
Each edge is labeled by a letter, no two edges outgoing from the same node have the same label.

Each string corresponds to a leaf.



Compressed Trie

Compress nodes with single outgoing edge, label edges by strings



Suffix tree

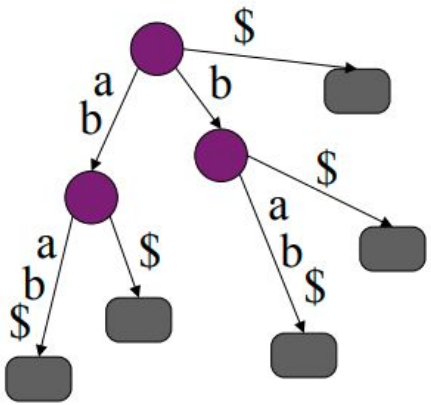
Given a string T, a suffix tree of T is a compressed trie of all suffixes of T

To make these suffixes prefix-free we add a special character, say $, at the end of T

Example

Suffix tree for the string T=abab$

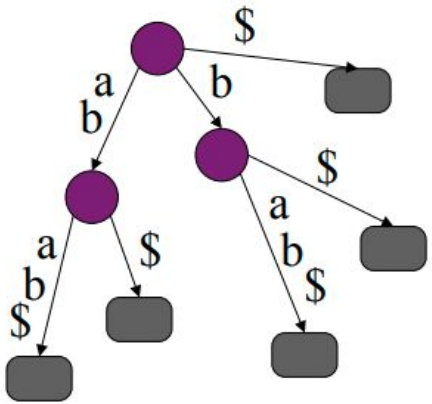
{$, b$, ab$,bab$, abab$}



Suffix tree

A suffix tree for an m-character string T:

- A rooted directed tree with exactly m leaves numbered from 1 to m.
- Each internal node, other than root, has at least two children and each edge is labeled with a non-empty substring of T.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of T that starts at position i, that is, spells out T[i,…,m].



What can we do with it?

Exact string matching:

Given a Text T, |T| = m, and a string s, |s| = n, does s occur in T?

Naïve solution: search for occurrence of s at every position in T.

Running time?

Using suffix trees: Construct suffix tree for T. How do we now check if s occurs in T?

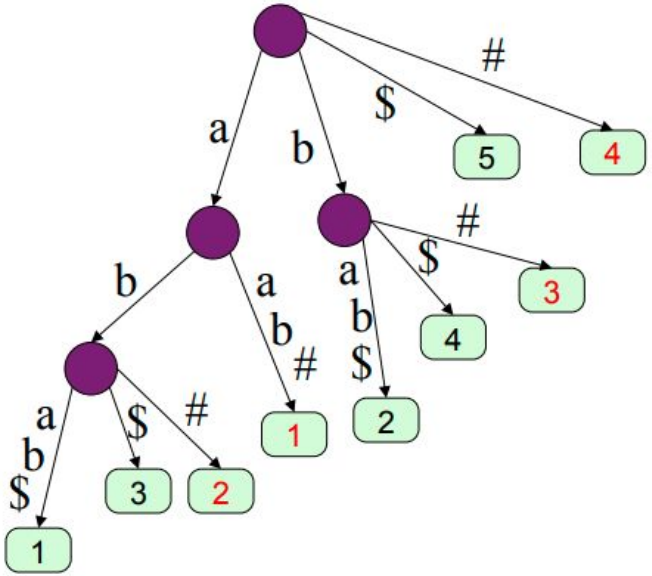
Generalized suffix tree

A generalized suffix tree for t_1=abab and t_2=aab

{

$	#
b$	b#
ab$	ab#
bab$	aab#
abab$	

}



...

Search Time Complexity:

O(m) (where m is the length of the pattern being searched).

Space Complexity:

O(n): (where n is the length of the input string).

Suffix trees are used for pattern matching, where each node represents a substring.

- The tree structure allows substring searches in linear time relative to the pattern length.
- However, it is space-intensive since it stores every possible suffix of the input string.

Suffix arrays

A suffix array is a sorted array of all suffixes of a given string.

Use less space, but not as fast.

Let T = abab.

Sort the suffixes lexicographically:

ab, abab, b, bab

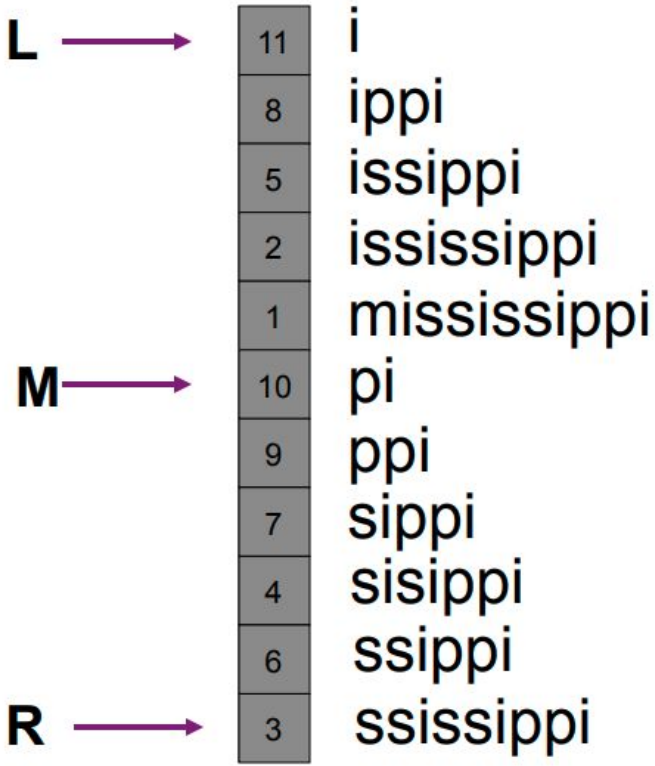
The suffix array gives the indices of the suffixes in sorted order.

3	1	4	2

Example

T = mississippi

s = issa



Searching a suffix array

If s occurs in T then all its occurrences are consecutive in the suffix array.

Do a binary search on the suffix array.

Takes O(n log m) time.

Searching a suffix array

If s occurs in T then all its occurrences are consecutive in the suffix array.

Do a binary search on the suffix array.

Takes O(n log m) time.

Constructing a suffix array

Build a suffix tree. Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array: O(m) time.

But if our objective was to save space, do it directly: O(mr log m) time, r = longest repeat length.

Other indexing structures

Link: https://nlp.stanford.edu/IR-book/html/htmledition/index-construction-1.html

Static properties of terms

- The Rule of 30
- **Heap's Law** - describes vocabulary growth.
- **Zipf's Law -** describes term frequency distribution.

What we need to compress?

Dictionary = Terms, pointers -> Reduce memory footprint for term lookup

Postings List = DocIDs, term frequencies, positions -> Major source of space saving

Documents (optional) = Cached text/snippets -> For storage and caching efficiency.

Static properties of terms

- The Rule of 30
  - The 30 most common words account for 30% of the tokens in the written text.
  - For eg:
    - a, an, the, …

- An empirical Finding (from experiments on several datasets)

$$M = kT^b$$

- ***M*** is the size of the Vocabulary
- ***T*** is the number of tokens in the collection
- ***k*** and ***b*** are the constants that are derived using empirical methods.
- usually k and b for Nepali language = 20 - 60 and 0.4 - 0.6

Reuters-RCV1

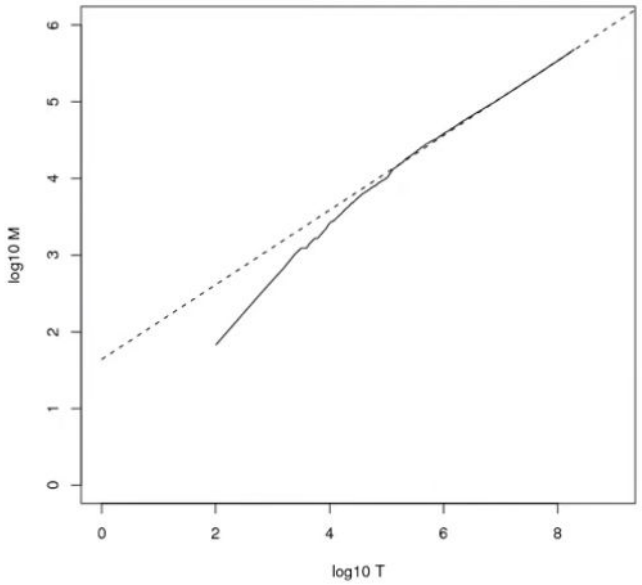
Table: Collection statistics for Reuters-RCV1. Values are rounded for the computations in this book. The unrounded values are: 806,791 documents, 222 tokens per document, 391,523 (distinct) terms, 6.04 bytes per token with spaces and punctuation, 4.5 bytes per token without spaces and punctuation, 7.5 bytes per term, and 96,969,056 tokens. The numbers in this table correspond to the third line (``case folding'') in icompresstb5.

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
T	tokens	100,000,000

Source: https://nlp.stanford.edu/IR-book/html/htmledition/blocked-sort-based-indexing-1.html

Heap's Law

- $\log_{10} M = 0.49 \log_{10} T + 1.64$ is the best least squares fit for RCV1.
- So $k = 10^{1.64} \approx 44$ and $b = 0.49$.
- For first 1,000,020 tokens, law predicts 38,323 terms.
- Actually, we have 38,365 terms.



- From above chat we can see that:
  - Dictionary Size grows with collection Size.
  - Size of dictionary can get really large for large collections.
    - So, dictionary compression is important for an effective information retrieval system.

Zipf's law: Modeling the distribution of terms

- We also want to understand how terms are distributed across documents.
- This helps us to characterize the properties of the algorithms for compressing postings lists

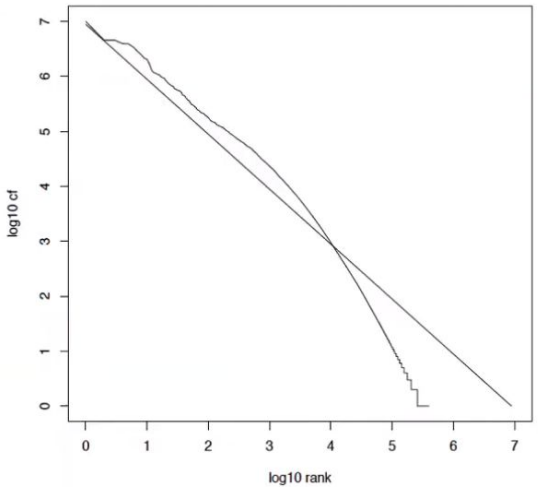
It states that, if $\mathbf{t_1}$ is the most common term in the collection, $\mathbf{t_2}$ is the next most common, and so on,

- then the collection frequency $\mathbf{cf_i}$ of the **ith** most common term is proportional to ***1/i***:

$$Cf_i \propto 1/i$$

- This also means that rank and Frequency are inversely proportional to each other.

Zipf’s law for Reuters RCV1



Why Zipf’s Law Matters for IR

1. **Index compression:**
    - Because most words are rare, posting lists are highly skewed.
    - You can store frequent words separately and apply variable-length encoding efficiently.
2. **Stopword removal:**
    - Extremely frequent words (“the”, “is”, “छ”) carry little information and are often dropped from the index.
3. **Query optimization:**
    - Knowing term frequency helps prioritize rare (more discriminative) terms.
4. **Language modeling:**
    - Zipf’s pattern is used in smoothing, text generation, and token probability estimation.

Why is compression needed in IR

1. Reduce Storage Space
   - Inverted indexes for large document collections can be massive
2. Improve Query Speed
   - Smaller indexes fit in memory → fewer disk reads.
3. Reduce I/O and Network Cost
   - Faster data transfer between memory and disk.
4. Enable Scalability
   - Efficient compression allows handling billions of documents.

Compressing the inverted index:

1. Lexicon:

- Contains a list of unique terms (words) that appear in the documents.
- Each term is associated with its frequency of occurrence (*occ) in the entire corpus.

2. Occurrences:

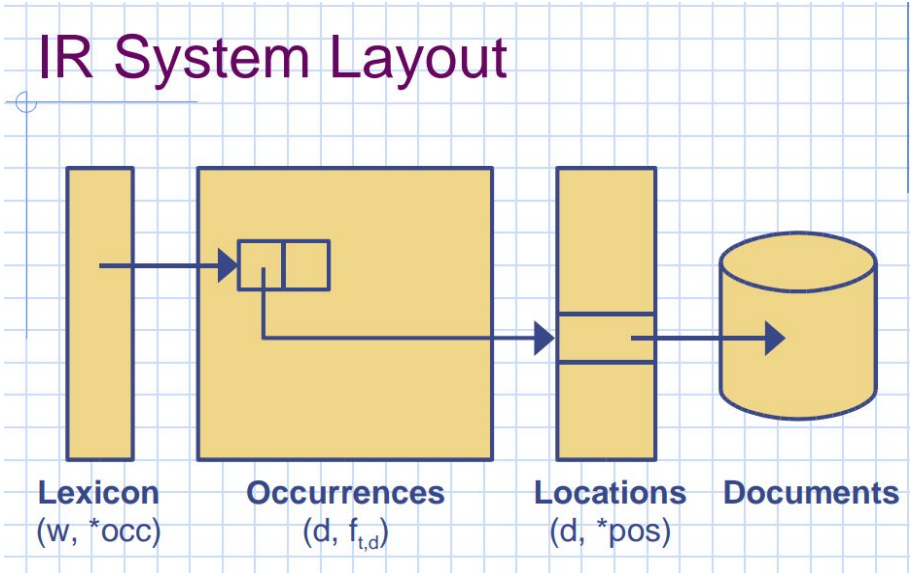
- Stores information about which documents contain a particular term and how often it appears in each document.
- The format is (document ID, frequency of term in document).

3. Locations:

- Records the positions of a term within each document.
- The format is (document ID, list of positions of the term).

4. Documents:

- Contains the actual text content of each document.



Actually, the components of inverted index can be seen as:

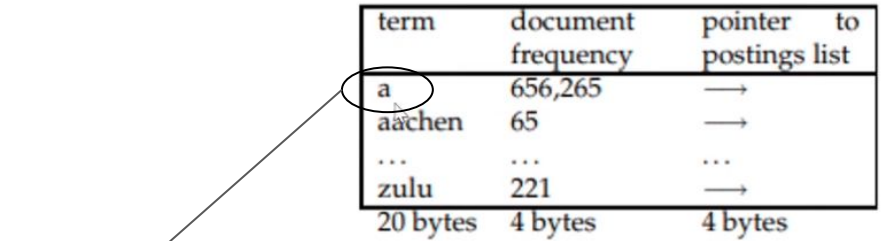
Component	Description	Example
Dictionary	**Stores each unique term once**	`cat`, `dog`, `banana`
Postings Lists	**For each term, stores list of docIDs and term positions**	`cat → [D1:2:(1,3), D2:1:(4)]`

Compression techniques

Type	Targets	Example Techniques
Dictionary Compression	**Term list**	**Sorted Array, Sorted String, Front Coding, Blocked Storage**
Postings Compression	**DocIDs & Positions**	**Gap Encoding, Variable Byte, Gamma Codes, Delta Codes**

Dictionary as a Sorted Array

How can we store dictionary in such a way that we save some space.

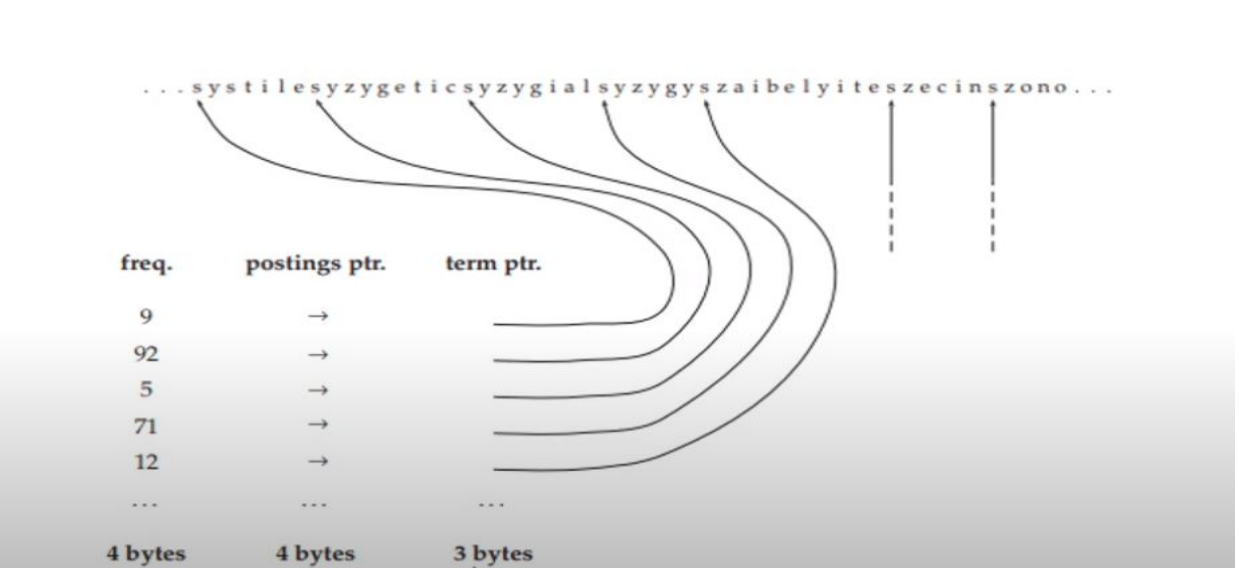


This **a** does not need 20 bytes.

For Reuters-RCV1, we need M x (2x20 + 4 + 4) = 400,000 x 48 = **19.2 MB for dictionary + offset table (each char of 2 byte)**

Also, M x (20 + 4 + 4) = 400,000 x 28 = **11.2 MB if 1 byte** for storing the dictionary in this scheme.

One way is: **Dictionary as a String**



Can we do better??

For Reuters-RCV1, we need M x () = 400,000 x (8+4+4+3+8) = **10.8 MB for vocab + table**
M x (4+4+3+8) = 400,000 x 19 = **7.6 MB for table**
for storing the dictionary in this scheme.

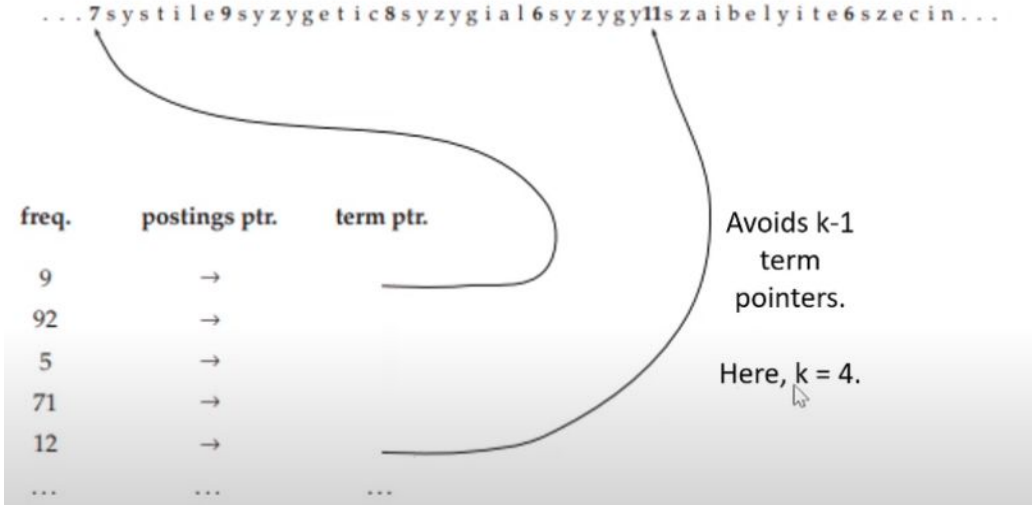
- Dictionary-as-a-string storage.
- Pointers mark the end of the preceding term and the beginning of the next.
- For example, the first three terms in this example are systile, syzygetic, and syzygial.

Blocked Storage

- further compress the dictionary by grouping terms in the string into of size ***k*** and keeping a term pointer only for the first term of each block

By increasing the block size **k**, we get better compression. For k=4, we save (k-1) x 3 = 9 bytes for term pointers,

- but need an additional k=4 bytes for term lengths.



So, the total space requirements for the dictionary of Reuters-RCV1 are reduced by 5 bytes per four-term block,

- or a total of 400,000 x ¼ x 5 =0.5

bringing us down to 10.3 and 7.1 MB.

Front coding

Term	Common prefix with previous	Stored as
compute	—	`7compute`
computer	“compute” (7)	`7* r`
computers	“computer” (8)	`8* s`
computing	“comput” (6)	`6* ing`

Thus, the front-coded string looks like:

`7compute7*r8*s6*ing`

Here:

- The `*` symbol separates the prefix length from the suffix.
- You only need to store the **first word fully**.

Huffman Coding for dictionary compression??

Learn yourself

Assignments

Also, try to encode and decode the following sentence:

“the study area is divided in mutually exclusive, bounded parts, with all locations in one part having the same field value.”

Use character and word based huffman coding and also with canonical huffman code.

...

Exercises.

Estimate the space usage of the Reuters-RCV1 dictionary with blocks of size k=8 and k=16 in blocked dictionary storage.

Table 5.2: Dictionary compression for Reuters-RCV1.

data structure	size in MB
dictionary, fixed-width	19.211.2
dictionary, term pointers into string	10.8 7.6
~, with blocking, $k = 4$	10.3 7.1
~, with blocking & front coding	7.9 5.9

Integer Compression

- We can compress integers using simple encodings to reduce the space required.
- There are various integer encoding techniques, such as Elias gamma coding, variable-byte encoding, and delta encoding.

Choosing the Best Encoding:

- The best encoding method depends on the distribution of values in the postings list.
- For example, if the values are mostly small, a variable-byte encoding might be more efficient than Elias gamma coding.

Document Gaps

- Representing d document numbers:
  - Keep d in ascending order
  - Store as sequence of gaps

Original Postings List:

- 3, 2
- 5, 1
- 20, 3
- 21, 2
- 23, 1
- 76, 2
- 77, 1
- 78, 2

Postings List with Document Gaps:

- 3, 2
- 2, 1
- 15, 3
- 1, 2
- 2, 1
- 53, 2
- 1, 1
- 1, 2

Gaps can be efficiently compressed
- frequent terms have short gaps
- rare terms have large gaps

Always use document id’s with gaps for encoding.

Integer Coding

- Non parametric codes
  - Binary
  - Unary
  - Elias gamma, delta
  - Variable-byte
- Parametric codes
  - Golomb
  - Local Golomb (rice coding)

Binary encoding

Postings List with Document Gaps:

- 3, 2
- 2, 1
- 15, 3
- 1, 2
- 2, 1
- 53, 2
- 1, 1
- 1, 2

DocIDs and TermIDs are integers, and instead of storing them as regular integers,

- we can encode them using binary encoding.
- in the list for the term "be" (3,2,15):

Eg.: 3 = 11
2 = 10
16 = 10000

- Used rarely by itself;
- Prone to long integers,
- usually in combination (e.g., Elias delta).

Unary code

- Encode integer n as (n-1) 1’s followed by a 0.
  - 1=0
  - 3=110
  - 9=1111111110
- Simple and fast
- Smallest numbers have very short codes, but codeword length grows quickly. (Inefficient for large values)

- Unary encoding can be used to represent the frequency of terms.
- For instance, in the list for the term "be," the frequency of 2 (which occurs in document 1) can be encoded as 11 (two 1s followed by a 0).

Similarly, a frequency of 1 can be represented as 10 in unary form.

Elias Gamma (γ) code

A **self-delimiting** universal code — efficient for small to medium integers.

Encoding Steps:

1. Write the number `n` in binary (without leading zeros).
2. Count the number of bits in that binary (say `L`).
3. Prefix `(L-1)` zeros to the binary.

n	Binary	L	Code
1	1	1	1
2	10	2	010
3	11	2	011
4	100	3	00100
5	101	3	00101
9	1001	4	0001001

Example:
Postings gaps `[3, 1, 4]`
→ **Gamma codes =** `011 1 00100`
→ **Combined:** `011100100`

...

the probability $\Pr(x)$ of encountering an integer x when using Elias Gamma encoding is given by:

$$\Pr(x) \approx \frac{1}{2x^2}$$

This implies that the likelihood of encountering a larger integer is much smaller than encountering a smaller integer, reflecting the fact that Elias Gamma encoding is optimized for small integers. The encoding becomes less efficient for larger integers, and their probability of occurrence is reduced accordingly.

Elias’ Delta (δ) code

Elias Delta Encoding is an extension of Elias Gamma encoding.

It is used for efficiently encoding integers, particularly when dealing with a large range of values.

The idea behind Delta coding is to apply Elias Gamma encoding to the length of the number, followed by the binary representation of the number without its most significant bit (MSB).

...

Improves on Gamma for larger numbers.

Encoding Steps:

1. Find binary of `n` (say length `L`).
2. Encode `L` with gamma code.
3. Append the remaining `(L-1)` bits of binary `n`.

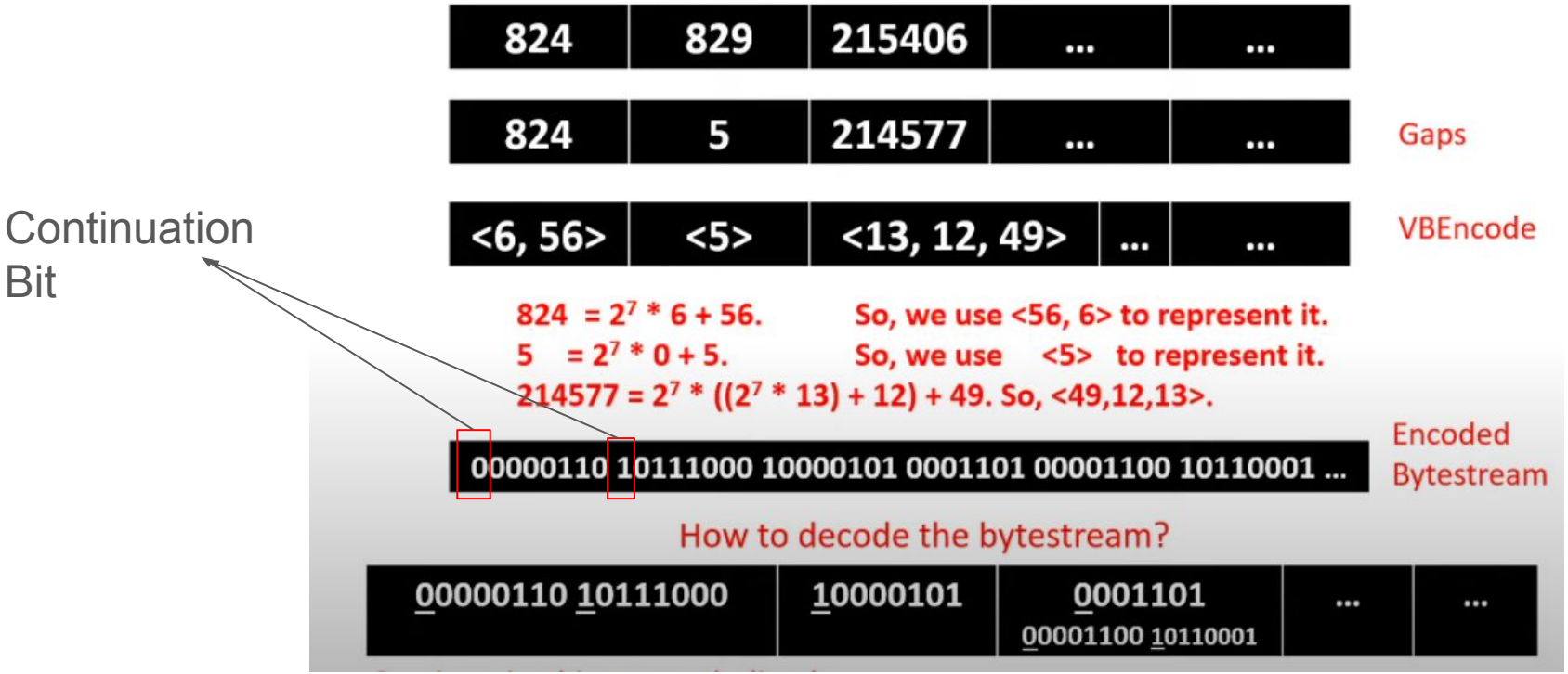
n	Binary	L	γ(L)	Code
3	11	2	010	0101
9	1001	4	00100	001001001

Example:
Encode gaps `[9, 3]`
→ Delta codes = `001001001 0101`
→ Compact and scalable.

Variable-Byte Code

- Binary, but use minimum number of bytes
- 7 bits to store value, 1 bit to flag if there is another byte
  - $0 < x < 128$: 1 byte
  - $128 < x < 16384$: 2 bytes
  - $16384 < x < 2097152$: 3 bytes
  - Integral byte sizes for easy coding
- Very effective for medium-sized numbers
- A little wasteful for very small numbers

Variable Byte encoding



...

Another Dig at 214577

- What is the VB encoding for 214577?

n	$\lfloor n/128 \rfloor$	n % 128
214577	1676	49
1676	13	12
13	0	13

- So, 214577 = <13, 12, 49> which means ((13*128)+12)*128 + 49.
- Thus we have, 214577 represented as 0001101 00001100 10110001 in VB encoded bytestream.

Exercise 1:

1. Compute the variable byte codes for the postings list
   a. (100, 200, 400, 800)
   b.

Original Postings List:

- 3, 2
- 5, 1
- 20, 3
- 21, 2
- 23, 1
- 76, 2
- 77, 1
- 78, 2

Why Parametric Compression in IR?

- Posting lists store **gaps** between document IDs.
- Gaps often follow a **geometric distribution** (small gaps are frequent).
- Golomb coding

Golomb Coding

Golomb coding encodes a positive integer x using a **divisor parameter** b, typically chosen so that it fits the geometric distribution of the gaps.

$$b = \lceil \frac{\ln(2)}{p} \rceil$$

or empirically chosen near the **average gap**.

Steps

1. Compute:
   - $q = \lfloor (x-1)/b \rfloor$
   - $r = (x-1) \mod b$
2. Encode q in **unary** (sequence of 1's followed by 0).
3. Encode r in **truncated binary** (fixed-length bits).
4. Concatenate: Unary(q) + Binary(r)

Local Golomb Coding

Simplified Golomb code where b is a power of 2 (b = 2^k).

Thus, the remainder can be represented with exactly k bits.

Term	DocIDs	Gaps
ai	[1, 3, 10, 15, 21]	[1, 2, 7, 5, 6]

Assignments

- Golomb Coding
- Local Golomb coding (Rice Coding)

Any Queries???