CHAPTER#2

Dated: _____

★ Extended Boolean Model => It is an extension to the Boolean Model with additional functionality such as proximity search, phrase search and wildcard search.

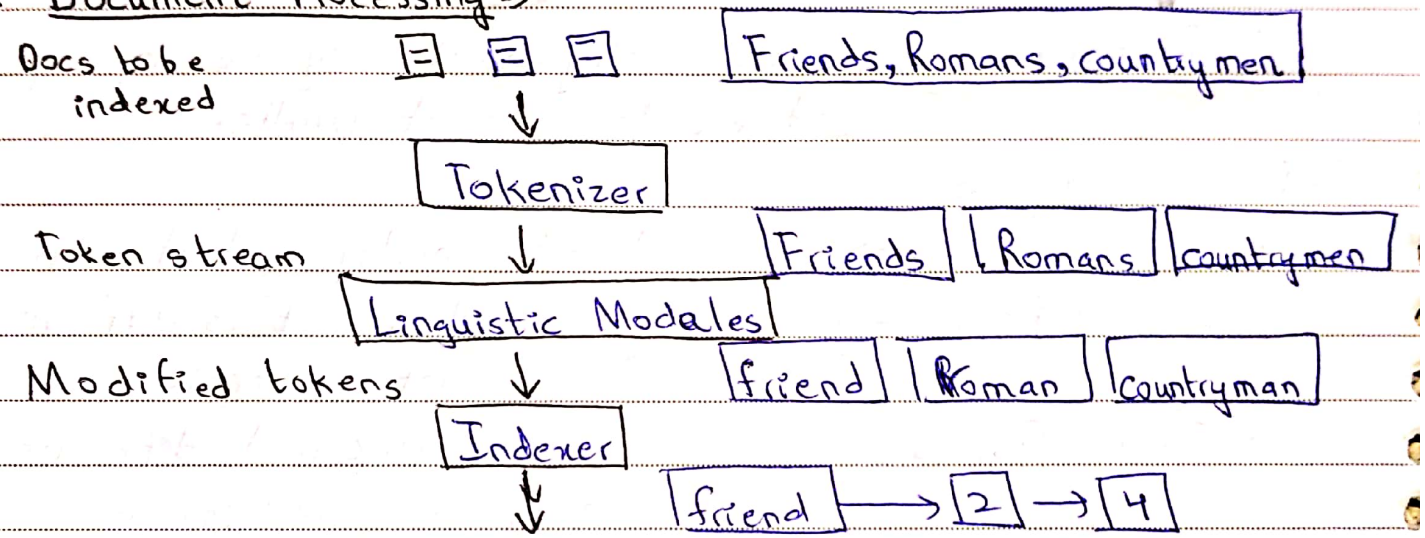
★ Word => A delimited string of characters as it appears in text.

★ Term => A "normalized" word (case, morphology, spelling) equi of word.

★ Token => An instance of word or a term occuring in a document.

★ Type => same as term usually, equivalence of token.

★ Document Processing =>



★ Challenges =>

↳ format => pdf / word / excel / html

↳ language

↳ encoding

★ Tokenization => A process through with documents are parsed and a sequence of characters separated, as a feature for document processing.

↳ Issues =>

↳ Multi-words => Hewlett-Packard, co-education

↳ Numbers => 3/20/91 , B-52

↳ Languages => dialects → pt , us , >,/> , pain

* Stop words => With a stop words list, we can exclude the most common words entirely. [the, a, is]
  ↳ Trend is away from ignoring them
  ↳ Good compression techniques => space for including stopwords in a system is very well.
  ↳ Good query optimization => pay little query time for including stop words.
  ↳ Why they are important?
  ↳ Phrase queries => "King of Denmark"
  ↳ Song titles => "Let it be", "To be or not to be"
  ↳ Relational queries => "flights to London"
* Normalization => Normalize words in indexed text as well as query words into the same basic form.
  ↳ Some strategies
  ↳ Deleting periods => U.S.A → USA
  ↳ Deleting hyphens => co-education → coeducation
  ↳ Accents => résumé → resume
* Case folding => Reduce all letters to lowercase.
  ↳ General Motors → general motors
* Asymmetric Expansion => Expand the search query to include related terms. It increases recall but reduces precision.
  ↳ window → window, windows
  ↳ windows → Windows, windows, window
* Equivalence Classing => Divide the search space into a smaller number of equivalent classes, each of which represent a group of related terms.
  ↳ {run, running, ran, runner} → single class

Dated: _____

★ Thesauri => A lexical resource that provides a collection of related words, including synonyms, antonyms and related terms.

↳ How to handle synonyms?

↳ Hand constructed equivalence classes → car = automobile

↳ rewrite to form equivalence classes terms

↳ if a doc has automobile, index it under car too.

↳ expand a query

↳ if a query has automobile, search under car too.

★ Soundex => A phonetic algorithm that is used to index words by their sound. The goal is to group similar sound words. [homonyms]

↳ Why soundex?

↳ It can handle spelling mistakes since it just checks how words sound.

★ Lemmatization => Implies doing "proper" reduction of inflectional / variant forms to dictionary headward base forms.

↳ Examples

↳ am, are, is ⟶ be

↳ car, cars, car's, cars' ⟶ car

↳ Definition

↳ algorithmic process of determing lemma from a given word.

↳ It may involve

↳ understanding context and determining the parts of speech.

↳ 'the walk' → also written as {walk, walked, walks}

↳ the base form walk is called lemma.

↳ Hard to implement

↳ incase of new languages

Dated: _____

★ Stemming ⇒ Stemming operates on a single word without knowledge of context.

↳ Easier to implement
  ↳ run faster but with reduced accuracy.

↳ Porter Stemmer
  ↳ word is cleaned up in the initialization phase, one prefix trimming phase, then five suffix trims.

  ↳ Initialization
    ↳ convert to lowercase and keep only digits & letters.
      ↳ F-16 → f16

  ↳ Remove prefixes
    ↳ {kilobyte, megabyte, nanobyte} → byte

  ↳ Suffix trim 1
    ↳ remove "es" from words ending in "sses" or "ies"
      ↳ passes → pass, cries → cri
    ↳ remove "s" from words where second last letter is not "s"
      ↳ runs → run, fuss → fuss
    ↳ remove "ed" if word has a vowel and ends with "eed".
      ↳ agreed → agre, freed → freed
    ↳ replace trailing "y" with "i" if word has a vowel
      ↳ satisfy → satisfi, fly → fli

  ↳ Suffix trim 2
    ↳ replace any suffix from left list to right list.
      ↳ tional → tion, ization → ize, fulness → ful

  ↳ Suffix trim 3
    ↳ replace any suffix from left to right list.
      ↳ icate → ic, ative → –

  ↳ Suffix trim 4
    ↳ remove al, ance, ence, etc..

  ↳ Suffix trim 5 [hinge → hing]
    ↳ remove trailing 'e' if word does not end in a vowel

Dated: _____

Stemming V/S	Lemmatization
1) Heuristic, rule based approach, generally fast, single term.	1) Rigor process that uses a dictionary and context to determine lemma, a slow approach.
2) generates unreadable tokens.	2) generates readable lexeme.
3) too aggressive; sacrificing precision for recall.	3) sacrifices recall for precision.

★ Morphology => Field of linguistics focused on the study of forms and formations of words in a language.

↳ Inflectional => adding a suffix to a word that does not change its grammatical category.

↳ work → worked, working

↳ Derivational => adding a suffix to a word that changes its grammatical category.

↳ nation (noun) => national (adj) => nationalize (verb)

↳ Implementation issues in ~~%~~ Inverted Index

↳ Lists => Lists can suffer from slow performance when dealing with complex morphological forms due to linear search. The memory reqs for storing frequency information for each term can become high when dealing with morphologically rich terms.

↳ Hash maps => The hash function must be able to deal with morphisms without causing collisions. The scalability of the hash function is also a concern due to increasing data.

↳ Trees => Maintaining a balanced tree can be challenging and requires a deep understanding of algorithms and data structures.

Waraj you are pagal! 😊
Dated:

↳ Implementation issues with skip lists
  ↳ Handling complex morphs ⇒ Morphologically rich data is tough to sort. Inflections and derivations can make the order tough to determine, thus lexicographical sort is harder.
  ↳ Scalability ⇒ They are designed to be scalable but the size of the data can increase overtime making maintainence tough.
  ↳ Memory management ⇒ They require a large amount of memory to store each frequency for morphologically rich terms since the memory requirements for such terms becomes prohibitively high.

★ Phrase queries ⇒ The multiword compounds or phrase queries make sense together e.g: "Stanford University". Such words cannot be broken down separately and queries due to false positives retrieved. So, the following solutions are implemented:
  ↳ Bi-word Indexes ⇒ The first approach is to index every consecutive pair of terms in the text as a phrase. For example, the text "Friends, Romans, Countrymen" will generate the biwords "friends romans", "romans countrymen". The two word phrase query-processing is now immediate.
  ↳ Longer Phrase queries ⇒ In cases of longer phrases like "stanford university palo alto", the following query will be broken into "stanford university", "university palo", "palo alto". Without the actual documents, we cannot verify if the actual phrase was retrieved, hence there will be a lot of false positives.

Dated: _____

phrase index

↳ Extended bi-words => We firstly parse the indexed text and part-of-speech tagging (POST). [a process that involves assigning a part of speech to each word]. Then, group the words into nouns (N) and function words (articles/prepositions) (X). Now, each string of the form NX^*N is an extended bi-word. For example: Coins are in pocket, the

N X X N

lookup and dictionary word for the text will be "coin pocket".

↳ Issues =>

↳ False +ves.

↳ Index blowup due to extremely huge dictionary.

↳ Positional Indexes => In the posting lists, store the position of appearence of each term for a document as well.

<term, number of docs; docNumber: position1, p2; >

For phrase queries, we would have to use a merge algorithm recursively while dealing with term proximity as well.

The following algorithm uses positional indexes to find phrases within proximity.

```
POSITIONAL INTERSECT (p1, p2, k):
1   answer ← ()
2   while p1 and p2:                    matching IDs
3       do if docID(p1) = docID(p2)
4           l ← ()
5           pp1 ← positions(p1)  ]  get term
6           pp2 ← positions(p2)  ]  positions
```

to be 1234 5678 2 ← 5,6 4 ① ← to

Dated: _____

```
7                while pp1:
8                    do while pp2:                    proximity
9                        do if |pos(pp1) - pos(pp2)| ≤ k   distance
10                           add (2, pos(pp2))
11                       else if pos(pp2) > pos(pp1)
12                           break   since sorted, no need to go ahead
13                       pp2 ← next(pp2)   ← get next position
14       some sort   [ while l ≠ () and |2[0] - pos(pp1)| > k
15       of recheking    DELETE (2[0])
16                   for each ps ∈ 2
17                       add (answer, <docID(p1), pos(pp1), ps>)
18                   pp1 ← next(pp1)
19               p1 ← next(p1)   ]  get next
20               p2 ← next(p2)   ]  posting list
21           else if docID(p1) < docID(p2)
22               p1 ← next(p1)
23           else
24               p2 ← next(p2)
25   return answer
```

★ **Proximity Queries** => They also use the above intersection algorithm to search words within proximity of \k words. eg: STATUE /3 FEDERAL /2 TORT. Clearly positional indexes are required for such queries.

★ **Positional Index Size** => Although the positional index expands the posting storage substantially, it is still standardized due to its power of proximity query searching.

↳ Rules of thumb => {holds for all english like languages}

↳ A positional index is 2-4 as large as non-positional index.

↳ It is 35-50% of the volume of original text.

Dated: _____

★ Combination Schemes => If users query commonly on particular phrases, such as "Micheal Jackson" it is inefficient to keep merging positional posting lists.

A combination strategy uses biword index for certain queries and positional index for the rest. Good queries are to be involved as phrase indexes such that the bi-words together are known to be common based on percent querying behavior.

↳ Williams Strategy => He combined the two schemes such that when a query is made, the phrase indexes are used to identify potential matches then the positional index is used to rank those matches based on closeness. With this strategy, he acheived the precision of positional indexing while being able to handle phrases. It required 26% more space but executed a simple web query in 1/4 time of positional index.

★ Query Processing Skip Lists => We can use skip lists for faster query processing.

↳ Tradeoffs

↳ More skips → shorter skip spans => more likely to skip but lots of comparison to skip pointers.

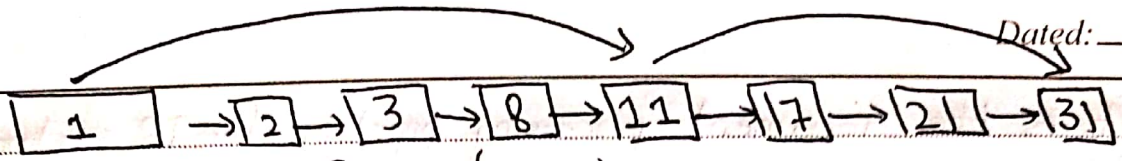
↳ Fewer skips → long skip spans => few successful skips

↳ Placing skips

↳ for postings of length L, use $\sqrt{L}$ evenly-spaced skip pointers.

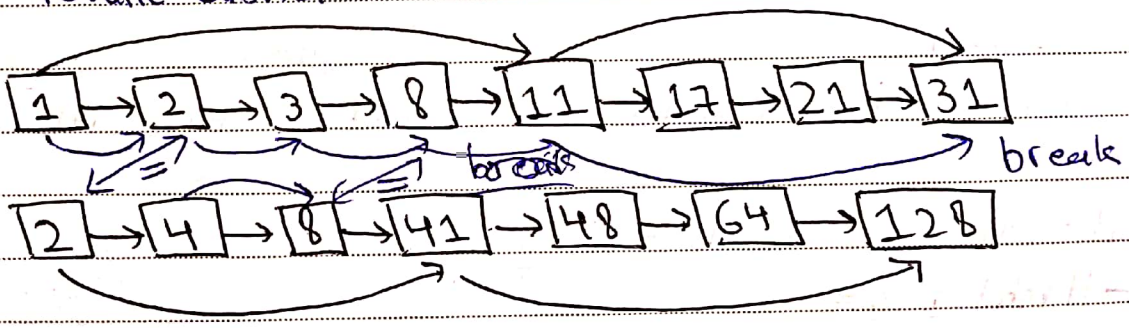
↳ Hard to maintain if L is dynamic.

Dated: _____



INTERSECT WITH SKIPS (p1, p2):

```
1   answer ← ()
2   while p1 and p2:
3       if docID(p1) = docID(p2):
4           add(answer, docID(p1))          ┐ if equal,
5           p1 ← next(p1)                   │ match found
6           p2 ← next(p2)                   ┘
7       else if docID(p1) < docID(p2):
8   while skip  { if hasSkip(p1) and docID(skip(p1)) ≤ docID(p2)
9   is possible,    while hasSkip(p1) and docID(skip(p1)) ≤ docID(p2)
10  keep skipping       p1 ← skip(p1)
11              else
12  else go one by one  p1 ← next(p1)
13              else if hasSkip(p2) and docID(skip(p2)) ≤ docID(p1)
14  same            while hasSkip(p2) and docID(skip(p2)) ≤ docID(p1)
15  as                  p2 ← skip(p2)
16  above       else
17                  p2 ← next(p2)
18  return answer
```



Dated: _____

→ What is the document processing pipeline for IR system?

It refers to a series of computational steps that are performed on a set of textual documents to transform them into a format that is suitable for indexing and searching.

The stages involve:

1) Document acquisition => system collects raw text docs.
2) Text preprocessing => cleaning and normalizing docs by removing irrelevant documents such as HTML tags, punctuation, stop words. It also involves stemming and lemmatization.
3) Indexing => The preprocessed text is converted into a form that can be efficiently stored or searched.
4) Query Processing => This stage involves processing user queries, typically in the form of a set of keywords and transforming them into a form that can be matched against the index. This may involve similar preprocessing and indexing steps for the query.
5) Ranking & Retrieval => The search engine retrieves a ranked list of relevant documents based on the query and similarity to the query.