AIW & Information Retrieval (UE18CS322)
Unit 1

Aronya Baksy

January 2021

1 Introduction

- Information Retrieval can be defined as finding material (documents) of an unstructured nature (usually text), that satisfies a given information need from within large collections (often stored on computers)
- Scope of IR:
  - Most text on the internet is semi-structured, meaning that even though the English language itself has no overt structure (other than the inherent semantics of the language), the vast majority of text is organized into documents, headings, sub-headings, paragraphs etc.
  - Clustering documents together based on the similarity of their contents
  - Classification based on topic, or other classes (such as age range, other information needs)
- IR systems operate at various scales:
  - **Web search**: Involving billions of documents on millions of machines, efficient and fast indexing, exploiting hyperlinks and avoiding manipulation of page content by sites to boost hits
  - **Personal IR**: Programs such as email search, or search utilities in consumer OSes like MacOS Spotlight Search and Windows Cortana Search, must be efficient and lightweight to be able to run on a single PC.
  - **Enterprise, Institutional, domain-based search**: restricted domain search, larger than personal but smaller than global web search.

2 Boolean Information Retrieval

- A Boolean IR Model is one that can answer any query that involves a set of terms combined by Boolean operators (*AND*, *OR*, *NOT*)
- One basic strategy for handling of Boolean queries is to use a pattern-matching program (eg: `grep` available on *nix systems) to iteratively scan through each document to see which documents satisfy the given query.
- The disadvantages of this approach are:
  - For large-scale corpora like the set of all documents on the WWW, this is a time consuming task
  - It does not allow for non-Boolean queries, or they may be impractical to do (eg: get all documents where Caesar and Calpurnia are at most 5 words apart)
  - It does not allow for ranked retrieval, i.e. to decide which of the retrieved results is the "best" based on some measure of goodness.
- An *information need* is a topic or some information that the user wishes to know more about. This information need is expressed to the computer (the IR system) in the form of a *query.*

- A result of a query is considered to be *relevant* if the user perceives it as containing value with respect to their information need.
- The effectiveness of an IR system is measured in terms of Precision and Recall

2.1 Basic Index construction

- The most basic form of term-document indexing is called the incidence matrix C. The rows of the matrix correspond to the terms, and the columns correspond to the documents in the collection.
- For a term t_i and a document d_j

$$C(t_i, d_j) = \begin{cases} 1 \text{ if term } t_i \in \text{ document } d_j \\ 0 \text{ if term } t_i \notin \text{ document } d_j \end{cases}$$

- Any Boolean query can then be answered by taking the incidence vectors for each term (i.e. the corresponding row of the matrix C) and then applying the corresponding Boolean operations in a bit-wise fashion.
- The term-document matrix is a sparse matrix (i.e. the majority of entries in it are 0) hence storing the entire matrix in memory is not feasible
- This model ignores information about term frequency within a document and positional information of terms within a document.

2.2 Inverted Index construction

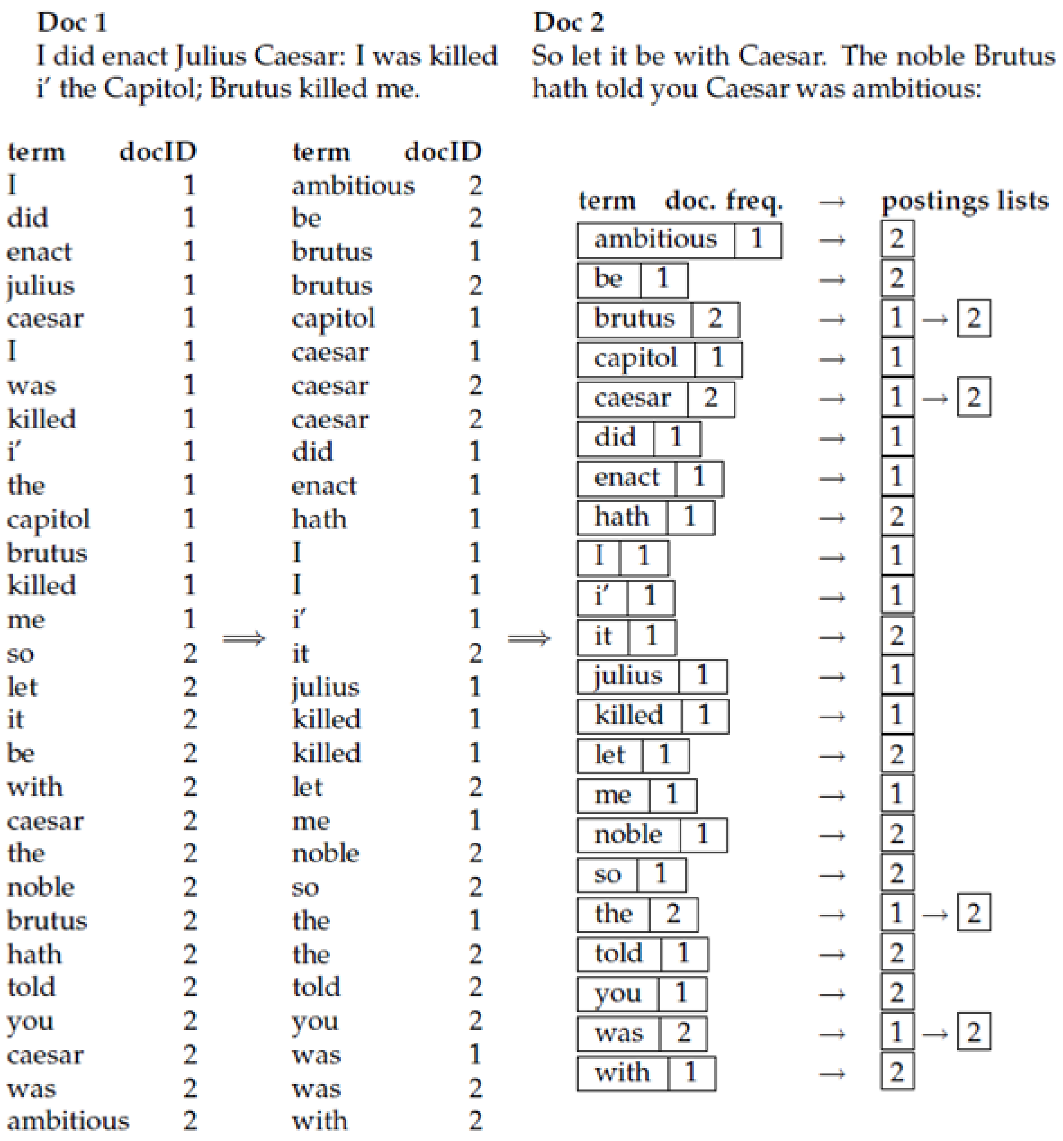
- The inverted index is called so because the search query asks for documents, but the index consists of terms that point to documents (or document IDs), i.e. the index is build by term and not directly by document.
- The inverted index consists of 2 parts: a *dictionary* of terms and a list of *postings*
- Vocabulary is defined as the **set** of all unique terms across the corpus
- The dictionary is a data structure that stores each term of the vocabulary, along with a pointer to its posting list (which is stored on disk) and additional info about that term (e.g. the number of documents it appears in, called the *document frequency*)
- A posting list for a particular term *t* is a list of document IDs that the term *t* appears in. For later convenience the posting list is maintained in sorted order by document ID.
- Steps involved in Inverted index construction are as follows:
  1. Tokenization of documents into tokens
  2. Normalization of tokens. This involves processes like conversion to a single uniform case (typically lowercase), lemmatization and stemming which are carried out by NLP modules (like `nltk` in Python, `stringr` in R).
  3. Generate a sequence of ⟨ term, docID ⟩ pairs for each normalized term in each document. Sort this sequence, first in lexical order of term, and then by docID.
  4. Merge multiple entries for a single term into a single posting list. The length of this posting list is the document frequency (number of documents in which that term appears).
  5. Separate the dictionary component (term, document frequency and pointer to posting list. stored in memory) and posting list component (stored on disk).
- To resolve Boolean queries of the form $x\ AND\ y$ we use the *intersect* algorithm to find the intersection of the posting lists of terms x and y.
- The intersect algorithm works in linear time $O(|x| + |y|)$ where $|x|$ and $|y|$ are the lengths of the two posting lists.

Algorithm 1 Intersect Algorithm

```
procedure INTERSECT($p_1$, $p_2$)
    $answer \leftarrow \langle\rangle$
    while $p_1 \neq NULL$ and $p_2 \neq NULL$ do
        if docID($p_1$) = docID($p_2$) then
            ADD(answer, docID($p_1$))
            $p_1 \leftarrow$ next($p_1$)
            $p_2 \leftarrow$ next($p_2$)
        else if docID($p_1$) < docID($p_2$) then
            $p_1 \leftarrow$ next($p_1$)
        else
            $p_2 \leftarrow$ next($p_2$)
    return answer
```

Algorithm 2 Union Algorithm

```
procedure UNION($p_1$, $p_2$)
    $answer \leftarrow \langle\rangle$
    while $p_1 \neq NULL$ and $p_2 \neq NULL$ do
        if docID($p_1$) = docID($p_2$) then
            ADD(answer, docID($p_1$))
            $p_1 \leftarrow$ next($p_1$)
            $p_2 \leftarrow$ next($p_2$)
        else if docID($p_1$) < docID($p_2$) then
            ADD(answer, docID($p_1$))
            $p_1 \leftarrow$ next($p_1$)
        else
            ADD(answer, docID($p_2$))
            $p_2 \leftarrow$ next($p_2$)
    if $p_1 < len(p_1)$ then                    ▷ Add remaining entries from list 1 if any
        while $p_1 < len(p_1)$ do
            ADD(answer, docID($p_1$))
            $p_1 \leftarrow$ next($p_1$)
    if $p_2 < len(p_2)$ then                    ▷ Add remaining entries from list 2 if any
        while $p_2 < len(p_2)$ do
            ADD(answer, docID($p_2$))
            $p_2 \leftarrow$ next($p_2$)
    return answer
```



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

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

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

$\Longrightarrow$

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

$\Longrightarrow$

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

Figure 1: Example for Inverted Index construction

- A slight modification of the intersect algorithm results in the union algorithm that works in linear time, to produce the union of two posting lists

- The NOT operation may or may not be completed in linear time.
- The strengths of a Boolean query model are that it is precise and it is simple to implement.
- The weaknesses of a Boolean query model are:
  - Not tolerant to spelling mistakes
  - Cannot handle phrase search as it does not store any position information for each term within a document.
  - Does not give importance to term frequency within a document
  - Does not retrieve documents in ranked order of relevance

2.3 Case Study on Boolean Retrieval: Westlaw

- Westlaw is an IR system for legal research that supports enhanced Boolean querying over tens of TBs of legal data stored as text

- Apart from the above discussed Boolean operators, Westlaw also implements other operators such as the **proximity operator** (a method of specifying that 2 terms must appear near each other in a document, nearness is defined as either number of words in between or in terms of grammar structures like paragraphs or sentences)
- Some operators used in the Westlaw IR system are:
  - **!**: A wildcard operator, represents zero or more characters (eg: access! represents all words starting with the prefix ”access” such as accessed, accessing, accessory etc.
  - **/p**: Represents that two terms must appear in the same paragraph
  - **/s**: Represents that two terms must appear in the same sentence.
  - Whitespace: Represents the OR operator (disjunction)
  - **/n** where n is an integer: represents that two terms must appear within n words of one another
- e.g.: **disabl! /p access! /s work-place work-site (employment /3 place)** means the words with prefix ”disabl” appearing in the same paragraph as words with prefix ”access” appearing the same sentence as either the word ”work-place” or the word ”work-site” or the phrase having ”employment” at 3 words distance from ”place”

2.4 Boolean Query Optimization

- For queries that consist of conjunction (AND) of n terms, the queries can be processed in increasing order of size.
- The AND operation that yields the *smallest possible set* as its answer is done first (as the size of the result is then constrained to this size). For this reason the document frequency (i.e. length of individual posting lists) is stored in the dictionary
- For queries involving OR of 2 terms, the formula

  $$|A \cup B| = |A| + |B| - |A \cap B|$$

  can be used. Since the value of $|A \cap B|$ is not known, a conservative estimate of the union size is

  $$|A \cup B| \approx |A| + |B|$$

- Using this estimate, the union queries can be processed in increasing order of size (smallest first).

3 Words, Tokens and Terms

3.1 Tokenization

- A token is defined as an instance of a sequence of characters in a particular document that are grouped together as a single semantic useful unit for processing.
- A type is the class of all tokens containing the same sequence of characters. A token is the (maybe normalized) type that is included in the dictionary of the IR system.
- Tokenization is the process of splitting up a block of text into tokens.

3.1.1 Issues with Tokenization

- Handling words with **apostrophes** in them. e.g.: ”Finland’s Capital” could be tokenized as either ⟨ Finland, s, Capital ⟩ or ⟨ Finlands, Capital ⟩. This issue also occurs in languages like French.
- **Domain-specific** tokens. e.g.: Aircraft names like B-52, A-320 or F-16, programming languages like C++ and C#, TV shows like F.R.I.E.N.D.S, IP addresses, web URLs, package tracking numbers. Such tokens are normally removed from the vocabulary to keep the dictionary size reasonable, but certain meta-data items like creation dates etc. can be indexed separately.

- **Hyphenation** is used in English for various purposes (separating vowels lke co-education, joining nouns like Hewlett-Packard, or word groupings like three-hundred-year-old trees). Splitting such words into tokens is either handled as a classification problem, or other heuristic rules.
- Splitting tokens with **whitespaces** in them (e.g.: Los Angeles which is a single token but white space split would make it into two). Splitting on whitespace can cause bad results as well (e.g.: a query for York University might yield documents containing New York University).
- **Language specific** issues. For example, compound nouns in German are not split by space, e.g. lebensversicherungsgesellschaftsangestellter means life insurance company employee. Or the fact that Chinese text is never split by white space, or that Arabic/Hebrew are written from right to left except sequences like numbers that are read left to right.

3.2 Stop word Removal

- Stop words are extremely common tokens which do not contain any semantic information that enhances the meaning of a sentence/paragraph.
- Examples of stop words are a, an, and, are, is, the, for, from
- Stop words can be identified by sorting terms by their collection frequency (number of times a term appears in the entire document collection), picking the k most frequently occurring terms and then hand-filtering those terms by their semantic content. The list of terms with little or no semantic content can be dropped from the index.
- Removal of stop words generally does no harm to performance of IR systems while reducing memory requirements, but phrase search will be affected by stop word removal (e.g.: a query for the phrase ”to be or not to be”)
- General trend in IR systems is reduction in size of stop lists (currently very few IR systems use stop lists, web search engines do not use them).

3.3 Normalization (equivalence classing of terms)

- Normalization is the processes of converting all tokens into a canonical form such that two or more sequences can match despite having superficial differences between one another.
- Most common method of normalization is implicitly creating equivalence classes using mapping rules (e.g.: deleting hyphens or deleting periods) to match tokens to a common form (e.g.: anti-discriminatory and antidiscriminatory can be mapped to the same class, so do U.S.A and USA).
- Relations between un-normalized tokens can be maintained using an index of un-normalized tokens each of which lead to an expansion list of vocabulary terms to consider for that given query term (hence a query term is an AND of the posting lists for all the corresponding synonym index terms)
- The above can also be done during the construction of the index. (e.g.: if ”automobile” is encountered in a document that docID can be added to both ”automobile” and ”car” posting lists)

3.3.1 Techniques for Normalization

- **Accents and diacritics** are generally removed due to issues with rendering them and the fact that they are less frequently used in queries (even though some words carry different meanings with and without a single accent)
- **Case-folding**, i.e. the conversion of all characters to a uniform case , most often lower case. This allows resolution of more queries but causes problems between common nouns and proper nouns (eg: General Motors)
- **True casing**: Using machine learning sequence models, or simple heuristic rules to normalize the case. (rules may be like: all uppercase words in a title or at the start of a sentence can be made lowercase, but all other uppercase words retain their original case).
- Using heuristics for normalizing words in foreign languages that may have different spellings and conventions. Most often used is the **Soundex algorithm** that classes terms with similar phonetic sounds together. Also used heuristic is the **edit distance**.

3.4 Lemmatization and Stemming

- Reducing all inflectional and variant forms of a word to a common base form is the goal of both stemming and lemmatization.
- Stemming uses a crude heuristic that involves chopping off the ends of words in the hope of achieving this normalization (direct removal of derivative suffixes)
- Lemmatization refers to the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the lemma (base form of the word).
- Most common stemming algorithm is the **Porter Stemmer**. It is a **fast rule based approach** that uses 5 sequential steps: the first step involves a rule-based direct replacement, and the following steps use the *measure* of the word (# of syllables) to detect whether a suffix is derivative or is actually a part of the word
- Other stemming algorithms are **Snowball Stemmer** and the **Paice-Husk Stemmer**.

4 Inverted Index Optimization

4.1 Intersection Algorithm with Skip pointers

Algorithm 3 Intersect Algorithm with Skip Pointers

```
procedure INTERSECT_SKIP($p_1$, $p_2$)
    $answer \leftarrow \langle\rangle$
    while $p_1 \neq NULL$ and $p_2 \neq NULL$ do
        if docID($p_1$) = docID($p_2$) then
            ADD(answer, docID($p_1$))
            $p_1 \leftarrow$ next($p_1$)
            $p_2 \leftarrow$ next($p_2$)
        else if docID($p_1$) < docID($p_2$) then
            if hasSkip($p_1$) and (docID(skip($p_1$)) $\leq$ docID($p_2$)) then
                while hasSkip($p_1$) and (docID(skip($p_1$)) $\leq$ docID(p2)) do
                    $p_1 \leftarrow$ skip($p_1$)
                $p_1 \leftarrow$ next($p_1$)
            else if hasSkip($p_2$) and (docID(skip($p_2$)) $\leq$ docID($p_1$)) then
                while hasSkip($p_2$) and (docID(skip($p_2$)) $\leq$ docID($p_1$)) do
                    $p_2 \leftarrow$ skip($p_2$)
                $p_2 \leftarrow$ next($p_2$)
    return answer
```

- The presence of skip pointers (i.e. pointers that allow one to skip multiple entries in a list rather than traverse in the linked order only) allows for faster traversal and eliminates docIDs that will not feature in the final result faster.
- The placing of skips involves a trade-off between large skip length (fewer pointer comparisons but also fewer chances to skip) and large number of skip pointers with short length (more likely to skip but more space needed to store pointers and also more pointer comparisons)
- A simple heuristic for pointer placement is, in a list of length P, there should be $\sqrt{P}$ evenly-spaced skip pointers.
- Skip pointers are useful for static document collections, but do not work with dynamic document collections.

4.2 Positional Indices

- Different approaches are needed to support phrase queries (i.e. queries composed of more than one term such as "Stanford university")

4.2.1 Bi-Word Indices

- In a bi-word index, each pair of consecutive terms is considered as a single dictionary entry. Such pairs are referred to as bi-words
- e.g.: The phrase ”friends, Romans, countrymen” yields the bi-words ”friends Romans” and ”Romans countrymen”.
- Multi-word queries (more than 2 words) can be broken up into bi-word pairs and then the intersection of all the intermediate results gives the answer.
- This can result in *false positives* when the result contains all the pairs but not consecutively. These have to be resolved using hand-filtering.
- Queries that consists of linking grammar structures such as ”abolition of slavery” and ”cost overruns on a power plant” can be resolved using Parts-Of-Speech Tagging (POS Tagging)
- Phrases of the form N x* N where N represents a noun and x represents any other word can be considered as **extended bi-words**. The extended bi-words can also be added to the index in order to resolve queries of the above form.
- Drawbacks of bi-word indexes are
  - Tendency to result in false positives
  - Size of the index blows up due to larger vocabulary size
- However some bi-word queries (such as ”Narendra Modi” or ”Joe Biden”) are very popular in web IR systems hence they can be included as part of the index.

4.2.2 Positional Indices

- In a positional index, a posting list for a term consists not only of the docID where it appears, but also a list of positions in that document where that term appears
- e.g.: $\langle$ term, doc freq $\rangle$: $\langle \langle doc_1\text{: } p_1, p_2, p_3 \rangle, \langle doc_2\text{: } p_1, p_2, p_3 \rangle \rangle$
- To evaluate queries of the form ”University of Minnesota”:
  - The posting lists of ”University” and ”Minnesota” are retrieved
  - The common doc IDs are found using the intersect algorithm
  - For each doc ID in the intersection set, the condition is checked whether Minnesota appears at a gap of 1 word after University. The doc IDs that satisfy the condition are returned.
- The above step can be extended to any length of gap between 2 terms in a phrase query (such as /n proximity queries in Westlaw IR)
- Positional indices are substantially larger than normal inverted indices, but the extra storage is offset by the larger flexibility and usefulness.
- In a **combination scheme**, frequent bi-words are indexed normally as vocabulary terms, but the remaining terms are stored as positional indices.
- The choice of which bi-words to store is based on how often these bi-words are queried for, and what is the time saving if they were directly stored in a bi-word index.

4.3 Posting List Implementation

- Fixed size array posting list leads to large amounts of wasted space. Variable sized arrays address this issue.
- Arrays work well for static corpora (where the action on the list is mostly read, hence due to caching based on locality of reference, accesses become faster)
- Dynamic indices are best implemented as linked lists, despite the extra space needed to store pointers, as insertion and deletion in linked lists can be done in constant time in linked list.
- For large posting lists, using linked list is not a good idea as it leads to large number of disk seeks during traversal (large list does not fit in memory). Arrays make more sense in this case.

4.4 Dictionary Implementation

- Most often used data structures are hash table and Binary Search Tree (or its balanced and more general version, the B-Tree).

4.4.1 Hash Table

- In a hash table, the term is converted to an integer by the *hash function.* This integer is used to access the relevant term information (the doc frequency and the pointer to posting list) from the table
- At query time: first find the index using hash function, then resolve collision (i.e. multiple terms having same hash value) and then locate the correct entry.
- **Pros**: Hash table has lookup time complexity of $O(1)$ (constant time)
- **Cons**:
  - Minor variants of a single query term may not be close to each other in the hash table
  - No prefix search
  - For dynamic corpora, rehashing is needed to avoid large number of collisions

4.4.2 Binary Search Tree

- BST is a type of tree wherein all keys in the left sub-tree of a node are smaller and all keys in the right sub-tree are larger in value than the key at that node.
- **Pros**:
  - Allow prefix queries
  - Terms are always stored in sorted order
- **Cons**:
  - Insertion takes more time than hash table($O(logN)$ time complexity only for balanced trees, this degrades to $O(N)$ for unbalanced trees)
  - Rebalancing a BST to keep the height optimal is an expensive operation

4.4.3 B Trees

- Every internal node has number of children in the range $[a, b]$ where $a < b$ and $a, b \in \mathbb{Z}^+$
- B-trees allow for prefix search like BST but also solve the issue of balancing
- A large b results in a fatter and shorter tree

5 Tolerant Retrieval

5.1 Wildcard Queries

- The wildcard character * represents any sequence of 0 or more characters. Any query with * in it can be considered as a wildcard query
- Queries where the * appears at the end are called prefix queries (e.g.: ”mon*” signifies all words that start with the prefix ”mon”). As seen before a B-Tree based index easily handles prefix queries
- Queries with * at the beginning are called suffix queries (e.g.: ”*mon” gives all words ending with mon). To service such queries, another inverted index of **reversed terms** is stored. The reverse index is accessed for such queries.
- Queries of the form X*Y where X and Y are sequences of characters are resolved as follows:
  - Retrieve terms that contain X* from the normal B Tree

  - Retrieve terms that contain *Y from the reversed B Tree
  - Intersect these two term lists.
  - Find the OR (disjunction) of all the posting lists of the terms in this intersection.
- For more complicated queries, the process involves multiple uses of the intersect algorithm which is an expensive operation. The **permuterm index** is used in such a scenario

5.2 Permuterm Index

- For every term in the standard inverted index, add a special character $ to the end, and rotate the term about the $ character. The $ indicates the end of a string
- All the possible rotations of a word are indexed to that word in an index called the permuterm index.
- e.g.: For hello$ the rotations are elloh, llohe, lohel, ohell. All these rotations are indexed to the same word ”hello” in the permuterm index.
- For wildcard queries of the form m*n$, it can be transformed into n$m*. From the permuterm index, the terms that match this query can be extracted and the standard inverted index can be queried to get the documents.
- For wildcard queries of the form x*y*z$, look up z$x* first from the permuterm index. This gives us a list of terms that satisfy the query x*z. Now from these, filter out the terms that contain y in them.
- Permuterm index is an efficient method for allowing wildcard queries, but its dictionary takes up a large amount of space due the number of rotations of a term that need to be stored for all terms.

5.2.1 K-gram Index

- The special $ character is used here to denote the beginning or the end of a term.
- In a k-gram index, the dictionary contains all k-grams that appear in the corpus. Each k-gram points to a posting list of terms that contain that k-gram.
- These lists are sorted in lexicographical order so as to allow the intersect algorithm to run.
- e.g.: for the term $castle$, the possible 3-grams are ca, cas, stl, tle, le. Hence the term ”castle” would appear in the posting lists for all five of these 3-grams.
- Assuming a 3-gram index, given a query like re*ve, the 3-grams $re and ve$ are searched, and the intersection of the two term lists is found.
- Given the same 3-gram index and a query like red*, the 3-grams to be searched are $re and red. This leads to false positives like the term ”retired” which satisfies both $re and red but does not satisfy the original query.
- To avoid these false positives, the intersection list is filtered by hand to check whether the terms match the original query. This is done using a simple string match.
- Most search engines and other IR systems hide the wildcard query functionality behind some ”Advanced Search” option, as it is an expensive operation to process such queries, and also because such queries are rarely used in practice.

5.3 Spelling Correction

- Spelling correction can either be used to correct misspelled words within the corpus, or to correct misspelled words in the query.
- Document error correction is used for documents retrieved using OCR using tuned algorithms and domain knowledge.

- For query correction, either the original query is itself modified to the correct one and documents are retrieved, or the system suggests corrections that the user can choose from, which can be used by the system.
- In **isolated** spelling correction, each word is corrected on its own, without considering the neighbouring words in the phrase. This approach does not catch typos caused by correctly spelled words (e.g.: I ducking hate this course)
- In **context-sensitive** spelling correction, the decision to correct an error is taken based on the surrounding words.

5.3.1 Edit Distance

- For isolated spelling correction, a notion of similarity between 2 words is needed. The edit distance is one such measure
- The edit distance between two strings s and d is the least number of transformations (insert character anywhere, delete any character, change a single character) needed to transform s into d.
- Dynamic programming can be used as the edit distance computation has the overlapping subproblems property. Assume a memoization matrix D that stores the edit distance. Arrange the letters of source s along the vertical and letters of destination d along the horiziontal
- Then the matrix is filled using:

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

- Then the bottom right entry of the matrix is the minimum edit distance
- Maintaining back-pointers as the optimal operation is chosen at each step allows the optimum path to also be known
- The weighted edit distance captures the weight of an operation (insert, delete, update) as per the characters involved in that operation (e.g.: m is more likely to be mistyped as n than as q, hence m-n distance is smaller)
- The modified recurrence relation is now:

$$D(i,j) = min \begin{cases} D(i-1,j)+del(s(i)) \\ D(i,j-1)+ins(d(j)) \\ D(i-1,j-1) + \begin{cases} subs(s(i),d(j)) \text{ if s(i)} \neq \text{d(j)} \\ 0 \text{ if s(i) = d(j)} \end{cases} \end{cases}$$

- The edit distance can be used for query correction in the following manner:
  - **Approach 1**: Generate all character sequences that are at edit distance d from the query. Intersect this list with the correct words set in the dictionary. Show the terms to the user as suggestions for the user to act
  - **Approach 2**: IR system looks up all possible corrections and returns the relevant documents. This can be refined by considering a single most likely correction (using the collection frequency of the corrected terms).

5.3.2 k-gram Index

- Using the k-gram index (refer 5.2.1) spelling correction can be refined further.
- Given the set of all k-grams in the query term, the term lists for each k-gram can be retrieved from the k-gram index.
- The naive approach is to compute the set of all terms that contain atleast n of the k-grams that are present in the query. (e.g.: for a query ”bord” the 2-grams are bo, or, rd. The terms that contain at least 2 of the 3 k-grams may be ”boardroom”, ”aboard”, ”border”)
- This naive approach fails to give appropriate corrections (”boardroom” for ”bord”). Hence a more nuanced measure of similarity is needed.
- The **Jaccard similarity** between 2 sets X and Y is defined as

$$sim_J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{1}$$

- A single linear scan through the entire vocabulary is done. For each vocabulary term t the Jaccard similarity is computed with the query term q.
- If $sim_J(q,t) > sim_{min}$ where sim_{min} is some minimum threshold for the similarity, then t is added to the list of possible corrections.
- note: computation of Jaccard similarity only needs lengths of 2 strings, hence it is efficient to do on the fly.

5.3.3 Context-Sensitive Spelling Correction

- Given a query with n words, take one word at a time, find its possible corrections and substitute them in the query.
- Run each substituted query by the IR system and see which query gives the largest number of results.
- This approach is costly if each term contains a large number of corrections possible. Several heuristics are used to trim this space, sich as hit-based refinement
- Hit-based refinement is used to optimize this to a certain extent:
  - Hits in corpus: number of times the suggested query appears in corpus
  - Hits in query logs: number of times a suggested query appears in past queries done by users.
- Bi-word based spelling correction is a technique where the query is first split into list of bi-words, then the bi-words that are common (again based on the hit-based refinement) are chosen to be expanded.

5.3.4 Phonetic Spelling Correction

- The Soundex algorithm (invented in 1918 for the US Census) is used for this purpose.
- Each term in the vocabulary is converted into a 4-character long code. Convert the query into a code and then retrieve all terms with the same code.
- The algorithm is summarized as follows:
  - Retain the first letter of the term
  - Replace all occurrences of A, E, I, O, U, H W, Y with 0
  - Substitute letters with numbers according to the rule:
    - B, F, P, V $\rightarrow$ 1
    - C, G, J, K, Q, S, X, Z $\rightarrow$ 2
    - D, T $\rightarrow$ 3

        * L → 4
        * M, N → 5
        * R → 6
    - Replace all pairs of consecutive digits with one digit
    - Remove all 0s from the string. Pad the remaining places with 0 and return the first 4 characters of the result. It should be of the form letter followed by 3 digits.
    - e.g.: HERMANN becomes H655, HERMAN becomes H655 too
- Soundex is implemented in many RDBMS (MS SQL Server, Oracle, PostgreSQL, etc.). It is not very useful for terms that are not names, and it has been shown to be biased towards certain nationalities.
- Soundex is suited for high recall tasks but may not offer good precision.