Introduction to Search Engines

Made by: Ali Ibrahim

Supervisor: MR. Ali Jnaide

Class: 12

Introduction:

As recently as the 1990s, studies showed that most people preferred getting information from other people rather than from information retrieval systems.

Of course, in that time period, most people also used human travel agents to book their travel. However, during the last decade, relentless optimization of information retrieval effectiveness has driven web search engines to new quality levels where most people are satisfied most of the time, and web search has become a standard and often preferred source of information finding. For example, the 2004 Pew Internet Survey (Fallows 2004) found that “92% of Internet users say the Internet is a good place to go for getting everyday information.”

What are search engines? How do they work?

And finally but not last how much flexible are they?

Information Retrieval:

A brief history:

The idea of using computers to search for relevant pieces of information was popularized in the article As we may think by Vannevar Bush in 1945.[1] It would appear that Bush was inspired by patents for a 'statistical machine' - filed by Emanuel Goldberg in the 1920s and '30s - that searched for documents stored on film.[2] The first description of a computer searching for information was described by Holmstrom in 1948,[3] detailing an early mention of the Univac computer. Automated information retrieval systems were introduced in the 1950s: one even featured in the 1957 romantic comedy, Desk Set. In the 1960s, the first large information retrieval research group was formed by Gerard Salton at Cornell. By the 1970s several different retrieval techniques had been shown to perform well on small text corpora such as the Cranfield collection (several thousand documents).[1] Large-scale retrieval systems, such as the Lockheed Dialog system, came into use early in the 1970s.

In 1992, the US Department of Defense along with the National Institute of Standards and Technology (NIST), cosponsored the Text Retrieval Conference (TREC) as part of the TIPSTER text program. The aim of this was to look into the information retrieval community by supplying the infrastructure that was needed for evaluation of text retrieval methodologies on a very large text collection. This catalyzed research on methods that scale to huge corpora. The introduction of web search engines has boosted the need for very large scale retrieval systems even further.

IR Definitions:

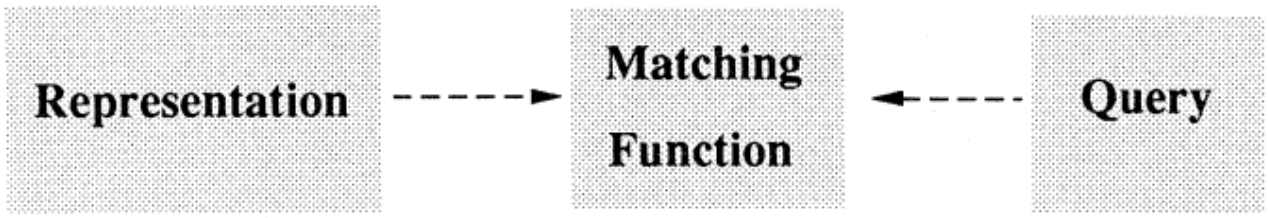
Since the 1940s the problem of information storage and retrieval has attracted increasing attention. It is simply stated: we have vast amounts of information to which accurate and speedy access is becoming ever more difficult. One effect of this is that relevant information gets ignored since it is never uncovered, which in turn leads to much duplication of work and effort. With the advent of computers, a great deal of thought has been given to using them to provide rapid and intelligent retrieval systems. In libraries, many of which certainly have an information storage and retrieval problem, some of the more mundane tasks, such as cataloguing and general administration, have successfully been taken over by computers. However, the problem of effective retrieval remains largely unsolved.

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).

As defined in this way, information retrieval used to be an activity that only a few people engaged in: reference librarians, paralegals, and similar professional searchers. Now the world has changed, and hundreds of millions of people engage

in information retrieval every day when they use a web search engine or search their email.

As such, IR is the core research in information science (Javerlinc ad vajjari). The objective is to study and understand IR processes in order to design, build and test retrieval systems that may facilitate the effective communication of desired information between human generator and human user.



(1)

Traditionally, information takes the form of text, implying that IR is synonymous with document or text retrieval, regardless of whether we are talking about full-text, administrative, directory, numeric or bibliographic information. In recent years the IR landscape has been extended to multi-media environments concerned with storage and retrieval of images, graphics, sound, software components, office document, etc.

The simplest model for IR is shown in figure (1) To the left potential information is represented, for example by itself, index terms, graphical structures or category code as well as formal data. To the right a requirement for information is represented by a query, in natural language or in artificial query language. In the center a matching function compares representations with query and retrievers text entities, e.g. documents or parts of documents, that provide the information that the user seeks.

Essentially, the problem is to find that *information* in the form of text(s) and other media which optimally satisfies the user's state of uncertainty and problem space. Hence, some texts are more relevant to a specific requirement for information than other texts, and a specific text may have different significance to several information requirements.[4]

Search Engines:

Types of search engines:

We will talk about three types of search engines:

1- Crawler-based search engines, such as Google, All The Web and AltaVista, create their listings automatically by using a piece of software to “crawl” or “spider” the web and then index what it finds to build the search base. Web page changes can be dynamically caught by crawler-based search engines and will affect how these web pages get listed in the search results.
Crawler-based search engines are good when you have a specific search topic in mind and can be very efficient in finding relevant information in this situation. However, when the search topic is general, crawler-base search engines may return hundreds of thousands of irrelevant responses to simple search requests, including lengthy documents in which your keyword appears only once.

2- Human-powered directories, such as the Yahoo directory, Open Directory and Look Smart, depend on human editors to create their listings. Typically, webmasters submit a short description to the directory for their websites, or editors write one for the sites they review, and these manually edited descriptions will form the search base. Therefore, changes made to individual web pages will have no effect on how these pages get listed in the search results.
Human-powered directories are good when you are interested in a general topic of search. In this situation, a directory can guide and help you narrow your search and get refined results. Therefore, search results found in a human-powered directory are usually more relevant to the search topic and more accurate. However, this is not an efficient way to find information when a specific search topic is in mind.

3- **meta search engine,** A **metasearch engine** (or aggregator) is a search tool that uses another search engine's data to produce their own results from the Internet.[5] Metasearch engines take input from a user and simultaneously send out queries to third party search engines for results. Sufficient data is gathered, formatted by their ranks and presented to the users.

A metasearch engine accepts a single search request from the user. This search request is then passed on to another search engine’s database. A metasearch engine does not create a database of webpages but generates a virtual database to integrate data from multiple sources.[6]

Since every search engine is unique and has different algorithms for generating ranked data, duplicates will therefore also be generated. To remove duplicates, a metasearch engine processes this data and applies its own algorithm. A revised list is produced as an output for the user.[7, 8] When a metasearch engine contacts other search engines, these search engines will respond in three ways:

- They will both cooperate and provide complete access to interface for the metasearch engine, including private access to the index database, and will inform the metasearch engine of any changes made upon the index database;

- Search engines can behave in a non-cooperative manner whereby they will not deny or provide any access to interfaces;
- The search engine can be completely hostile and refuse the metasearch engine total access to their database and in serious circumstances, by seeking legal methods.[9]

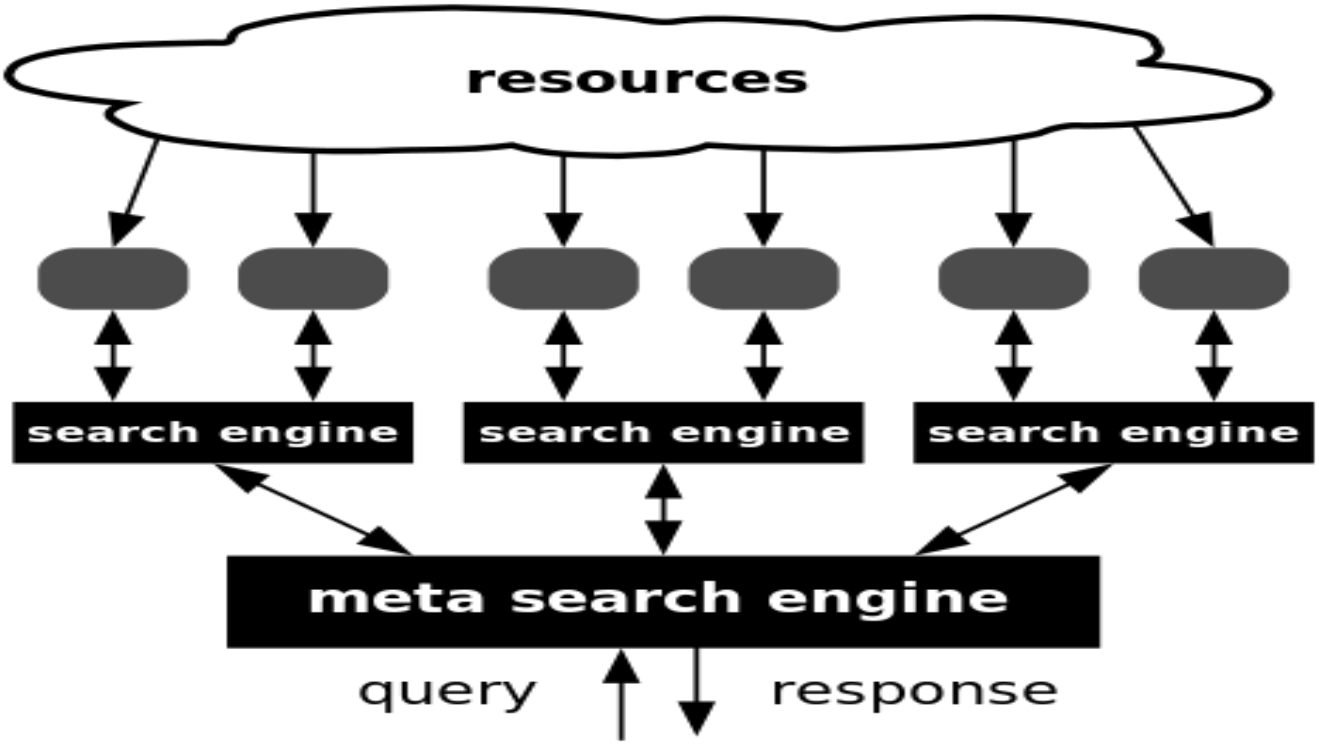
Architecture of ranking:

Webpages that are highly ranked on many search engines are likely to be more relevant in providing useful information. However, all search engines have different ranking scores for each website and most of the time these scores are not the same. This is because search engines priorities different criteria and methods for scoring, hence a website might appear highly ranked on one search engine and lowly ranked on another. This is a problem because Metasearch engines rely heavily on the consistency of this data to generate reliable accounts.[10]

Fusion:

A metasearch engine uses the process of Fusion to filter data for more efficient results. The two main fusion methods used are: Collection Fusion and Data Fusion.

- Collection Fusion: also known as distributed retrieval, deals specifically with search engines that index unrelated data. To determine how valuable these sources are, Collection Fusion looks at the content and then ranks the data on how likely it is to provide relevant information in relation to the query. From what is generated, Collection Fusion is able to pick out the best resources from the rank. These chosen resources are then merged into a list.
- Data Fusion: deals with information retrieved from search engines that indexes common data sets. The process is very similar. The initial rank scores of data are merged into a single list, after which the original ranks of each of these documents are analyzed. Data with high scores indicate a high level of relevancy to a particular query and are therefore selected. To produce a list, the scores must be normalized using algorithms such as Comb Sum. This is because search engines adopt different policies of algorithms resulting in the score produced being incomparable.



(2)

How do Search Engines work?

Hardware basics:

When building an information retrieval (IR) system, many decisions are based on the characteristics of the computer hardware on which the system runs. Here is a list of hardware basics that we need to motivate IR system design follows:

-Consequently, we want to keep as much data as possible in memory, especially those data that we need to access frequently. We call the technique of keeping frequently used disk data CACHING in main memory *caching*.

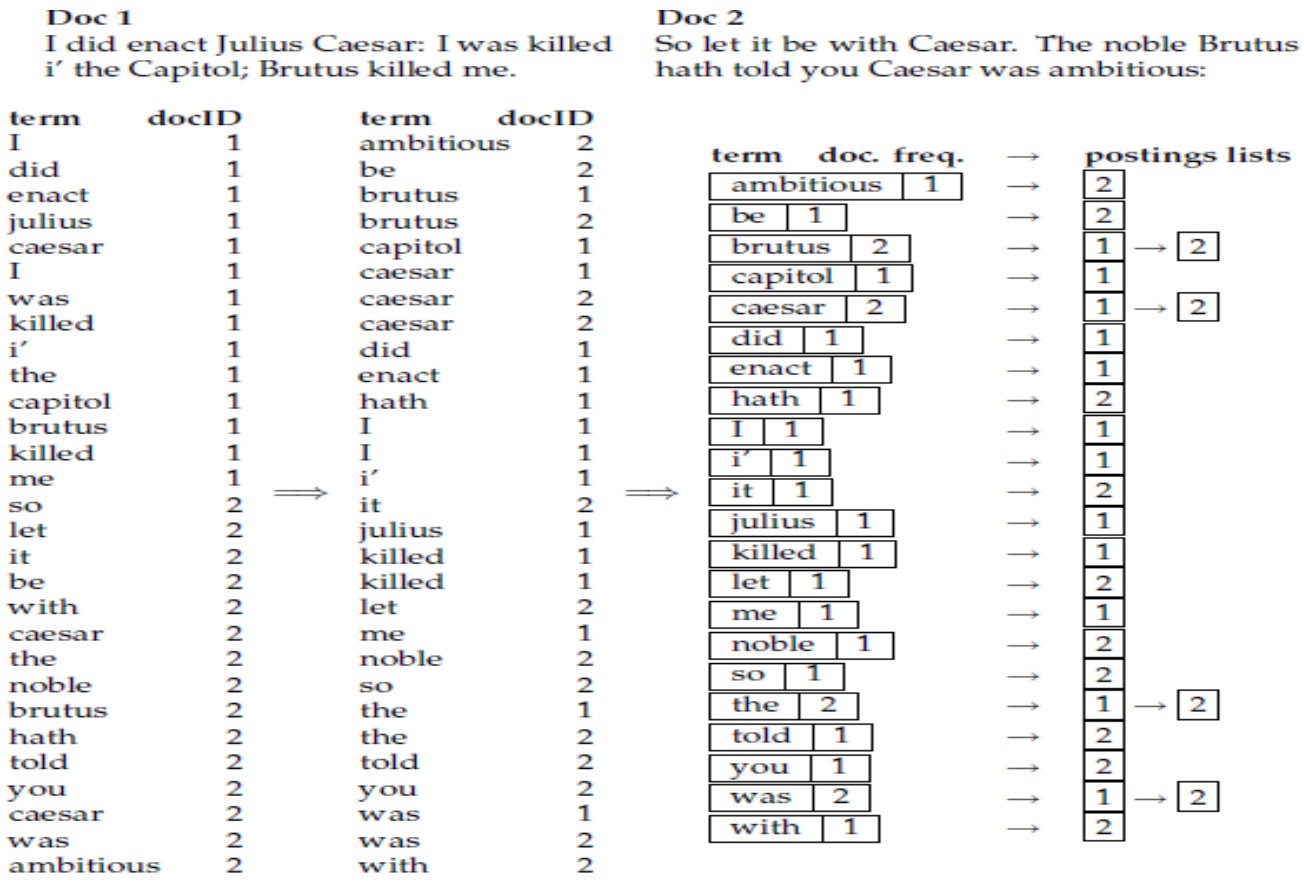
-Servers used in IR systems typically have several gigabytes (GB) of main memory, sometimes tens of GB. Available disk space is several orders of magnitude larger.

Indexing:

Blocked sort-based indexing:

The basic steps in constructing a nonpositional index are depicted in Figure 3. We first make a pass through the collection assembling all term–docID pairs. We then sort the pairs with the term as the dominant key and docID as the secondary key. Finally, we organize the docIDs for each term into a postings list and compute statistics like term and document frequency. For small collections, all this can be done in memory. In this chapter, we describe methods for large collections that require the use of secondary storage.

To make index construction more efficient, we represent terms as termIDs (instead of strings as we did in Figure 3), where each *termID* is a unique serial number. We can build the mapping from terms to termIDs on the fly while we are processing the collection; or, in a two-pass approach, we compile the vocabulary in the first pass and construct the inverted index in the second pass.



(3)

Dynamic indexing:

Thus far, we have assumed that the document collection is static. This is fine for collections that change infrequently or never (e.g., the Bible or Shakespeare).
But most collections are modified frequently with documents being added, deleted, and updated. This means that new terms need to be added to the dictionary, and postings lists need to be updated for existing terms.

The simplest way to achieve this is to periodically reconstruct the index from scratch. This is a good solution if the number of changes over time is small and a delay in making new documents searchable is acceptable – and if enough resources are available to construct a new index while the old one is still available for querying.
If there is a requirement that new documents be included quickly, one solution is to maintain two indexes: a large main index and AUXILIARY INDEX a small *auxiliary index* that stores new documents. The auxiliary index is kept in memory. Searches are run across both indexes and results merged. Deletions are stored in an invalidation bit vector. We can then filter out deleted documents before returning the search result. Documents are updated by deleting and reinserting them.

Each time the auxiliary index becomes too large, we merge it into the main index. The cost of this merging operation depends on how we store the index in the file system. If we store each postings list as a separate file, then the merge simply consists of extending each postings list of the main index by the corresponding postings list of the auxiliary index. In this scheme, the reason for keeping the auxiliary index is to reduce the number of disk seeks required over time. Updating

each document separately requires up to M_{ave} disk seeks, where M_{ave} is the average size of the vocabulary of documents in the collection. With an auxiliary index, we only put additional load on the disk when we merge auxiliary and main indexes.

Unfortunately, the one-file-per-postings-list scheme is infeasible because most file systems cannot efficiently handle very large numbers of files. The simplest alternative is to store the index as one large file, that is, as a concatenation of all postings lists.[4]

Determining the vocabulary of terms:

Tokenization:

Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation. These tokens are often loosely referred to as terms or words, but it is sometimes important to make a type/token distinction. A *token* is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. A *type* is the class of all tokens containing the same character sequence. A *term* is a (perhaps normalized) type that is included in the IR system's dictionary. The set of index terms could be entirely distinct from the tokens, for instance, they could be semantic identifiers in a taxonomy, but in practice in modern IR systems they are strongly related to the tokens in the document. However, rather than being exactly the tokens that appear in the document, they are usually derived from them by various normalization processes. For example, if the document to be indexed is to sleep perchance to dream, then there are5 tokens, but only 4 types (since there are2 instances of to). However, if to is omitted from the index (as a stop word) then there will be only 3 terms: sleep, per chance, and dream. The major question of the tokenization phase is what are the correct tokens to use? In this example, it looks fairly trivial: you chop on whitespace and throw away punctuation characters. This is a starting point, but even for English there are a number of tricky cases. For example, what do you do about the various uses of the apostrophe for possession and contractions? Mr. O'Neill thinks that the boys' stories about Chile's capital aren't amusing. For O'Neill, which of the following is the desired tokenization?

neil	oneill	o'neill	o'	neill

o	neill

And for aren't, is it:

aren't	arent	are	n't

aren	t

A simple strategy is to just split on all non-alphanumeric characters, but while (o) (neill) looks okay, (aren) (t) looks intuitively bad. For all of them, the choices determine which Boolean queries will match. A query of neill AND capital will match in three cases but not the other two. In how many cases would a query of o'neill AND capital match? If no preprocessing of a query is done, then it would match in only one of the five cases.

For either Boolean or free text queries, you always want to do the exact same tokenization of document and query words, generally by processing queries with the same tokenizer. This guarantees that a sequence of characters in a text will always match the same sequence typed in a query.3 These issues of tokenization are language-specific. It thus requires the language of the document to be known. Language identification based on classifiers that use short character subsequences as features is highly effective; most languages have distinctive signature patterns.

Normalization:

Having broken up our documents (and also our query) into tokens, the easy case is if tokens in the query just match tokens in the token list of the document. However, there are many cases when two character sequences are not quite the same but you would like a match to occur. For instance, if you search for USA, you might hope to also match documents containing U.S.A. Token normalization is the process of canonicalizing tokens so that matches occur despite superficial differences in the character sequences of the tokens. The most standard way to normalize is to implicitly create equivalence classes, which are normally named after one member of the set. For instance, if the tokens anti-discriminatory and ant discriminatory are both mapped onto the term ant discriminatory, in both the document text and queries, then searches for one term will retrieve documents that contain either. The advantage of just using mapping rules that remove characters like hyphens is that the equivalence classing to be done is implicit, rather than being fully calculated in advance: the terms that happen to become identical as the result of these rules are the equivalence classes. It is only easy to write rules of this sort that remove characters. Since the equivalence classes are implicit, it is not obvious when you might want to add characters. For instance, it would be hard to know to turn ant discriminatory into anti-discriminatory.

Stemming and lemmatization:

For grammatical reasons, documents are going to use different forms of a word, such as organize, organizes, and organizing. Additionally, there are families of derivationally related words with similar meanings, such as democracy, democratic, and democratization. In many situations, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set. The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. For

instance: (am, are, is⇒ be) (car, cars, car's, cars'⇒ car). The result of this mapping of text will be something like: (the boy's cars are different colors⇒ the boy car be differ color). However, the two words differ in their flavor. Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma. If confronted with the token saw, stemming might return just s, whereas lemmatization would attempt to return either see or saw depending on whether the use of the token was as a verb or a noun. The two may also differ in that stemming most commonly collapses derivationally related words, whereas lemmatization commonly only collapses the different inflectional forms of a lemma. Linguistic processing for stemming or lemmatization is often done by an additional plug-in component to the indexing process, and a number of such components exist, both commercial and open-source. The most common algorithm for stemming English, and one that has repeatedly been shown to be empirically very effective, is Porter's algorithm (Porter 1980). The entire algorithm is too long and intricate to present here, but we will indicate its general nature. Porter's algorithm consists of 5 phases of word reductions, applied sequentially. Within each phase there are various conventions to select rules, such as selecting the rule from each rule group that applies to the longest suffix.[4] In the first phase, this convention is used with the following rule group:

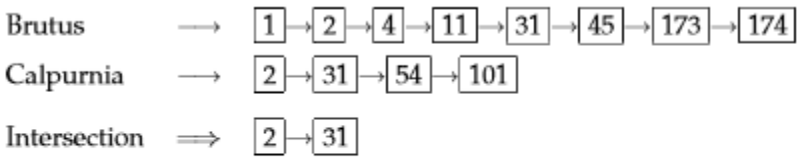
Rule			Example		
SSES	→	SS	caresses	→	caress
IES	→	I	ponies	→	poni
SS	→	SS	caress	→	caress
S	→		cats	→	cat

(4)

Query Process:

inverted index:

How do we process a query using an inverted index and the basic Boolean retrieval model? Consider processing the simple conjunctive query: Brutus AND Calpurnia



Intersecting the postings lists for Brutus and Calpurnia

(5)

We: 1. Locate Brutus in the Dictionary 2. Retrieve its postings 3. Locate Calpurnia in the Dictionary 4. Retrieve its postings 5. Intersect the two postings lists, as shown in Figure above.

The intersection operation is the crucial one: we need to efficiently intersect postings lists so as to be able to quickly find documents that contain both terms. (This operation is sometimes referred to as merging postings lists: this slightly counterintuitive name reflects using the term merge algorithm for a general family of algorithms that combine multiple sorted lists by interleaved advancing of pointers through each; here we are merging the lists with a logical AND operation.) There is a simple and effective method of intersecting postings lists using the merge algorithm: we maintain pointers into both lists and walk through the two postings lists simultaneously, in time linear in the total number of postings entries. At each step, we compare the docID pointed to by both pointers. If they are the same, we put that docID in the results list, and advance both pointers. Otherwise we advance the pointer pointing to the smaller docID. If the lengths of the postings lists are x and y, the intersection takes O (x + y) operations. Formally, the complexity of querying is Θ(N), where N is the number of documents in the collection. Our indexing methods gain us just a constant, not a difference in Θ time complexity compared to a linear scan, but in practice the constant is huge. To use this algorithm, it is crucial that postings be sorted by a single global ordering. Using a numeric sort by docID is one simple way to achieve this.[4] We can extend the intersection operation to process more complicated queries like: (Brutus OR Caesar) AND NOT Calpurnia

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

Algorithm for the intersection of two postings lists p_1 and p_2. (6)

Positional postings and phrase queries:

Many complex or technical concepts and many organization and product names are multiword compounds or phrases. We would like to be able to pose a query such as Stanford University by treating it as a phrase so that a sentence in a document like The inventor Stanford Ovshinsky never went to university. is not a match. Most recent search engines support a double quotes syntax (“Stanford

university”) for phrase queries, which has proven to be very easily understood and successfully used by users.

1. Biword indexes:
   One approach to handling phrases is to consider every pair of consecutive terms in a document as a phrase. For example, the text Friends, Romans, Countrymen would generate the biwords:
   friends romans, romans countrymen.
   This query could be expected to work fairly well in practice, but there can and will be occasional false positives. Without examining the documents, we cannot verify that the documents matching the above Boolean query do actually contain the original 4-word phrase. The concept of a biword index can be extended to longer sequences of words, and if the index includes variable length word sequences, it is generally referred to as a phrase index. Indeed, searches for a single term are not naturally handled in a biword index (you would need to scan the dictionary for all biwords containing the term), and so we also need to have an index of single-word terms. While there is always a chance of false positive matches, the chance of a false positive match on indexed phrases of length 3 or more becomes very small indeed. But on the other hand, storing longer phrases has the potential to greatly expand the vocabulary size. Maintaining exhaustive phrase indexes for phrases of length greater than two is a daunting prospect, and even use of an exhaustive biword dictionary greatly expands the size of the vocabulary. However, towards the end of this section we discuss the utility of the strategy of using a partial phrase index in a compound indexing scheme.
2. Positional indexes:
   For the reasons given, a biword index is not the standard solution. Rather, a positional index is most commonly employed. Here, for each term in the vocabulary, we store postings of the form docID: hposition1, position2, ...i, as shown in Figure (7), where each position is a token index in the document. Each posting will also usually record the term frequency. To process a phrase query, you still need to access the inverted index entries for each distinct term. As before, you would start with the least frequent term and then work to further restrict the list of possible candidates. In the merge operation, the same general technique is used as before, but rather than simply checking that both terms are in a document, you also need to check that their positions of appearance in the document are compatible with the phrase query being evaluated. This requires working out of sets between the words.

```
POSITIONALINTERSECT(p1, p2, k)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then l ← ⟨ ⟩
 5             pp1 ← positions(p1)
 6             pp2 ← positions(p2)
 7             while pp1 ≠ NIL
 8             do while pp2 ≠ NIL
 9                do if |pos(pp1) − pos(pp2)| ≤ k
10                      then ADD(l, pos(pp2))
11                      else if pos(pp2) > pos(pp1)
12                              then break
13                   pp2 ← next(pp2)
14                while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
15                do DELETE(l[0])
16                for each ps ∈ l
17                do ADD(answer, ⟨docID(p1), pos(pp1), ps⟩)
18                pp1 ← next(pp1)
19             p1 ← next(p1)
20             p2 ← next(p2)
21        else if docID(p1) < docID(p2)
22                then p1 ← next(p1)
23                else p2 ← next(p2)
24  return answer
```

An algorithm for proximity intersection of postings lists p_1 and p_2. The algorithm finds places where the two terms appear within k words of each other and returns a list of triples giving docID and the term position in p_1 and p_2.

(7)

Here, /k means “within k words of (on either side)”. Clearly, positional indexes can be used for such queries; biword indexes cannot.[4]

Conclusion:

The usefulness of a search engine depends on the relevance of the results it gives back. While there may be millions of Web pages that include a particular word or phrase, some pages may be more relevant, popular, or authoritative than others.

Most search engines employ methods to rank the results to provide the "best" results first. How a search engine decides which pages are the best matches, and what order the results should be shown in, varies widely from one engine to another. The methods also change over time as Internet usage changes and new techniques evolve. So after doing my research I think human can create search engines which can understand what the user really wants even if he didn’t write the right query.

Suggestions:

- We should learn the basics of Boolean Jebra to understand how search engines works; to make new algorithms to make it easier to the researchers.
- We should support (Shamra) the first Syrian search engine and connect the creators of it to help us.

Contents

Table of figures:

References:

1. Singhal, A., *Modern Information Retrieval: A Brief Overview.* 2001: p. 35–43.
2. Croft, M.S.W.B., *The History of Information Retrieval Research.* 2012: p. 1444–1451.
3. Holmstrom, J., *'Section III. Opening Plenary Session.* 1948: p. 85.
4. Christopher D. Manning, P.R., Hinrich Schütze, *An Introduction to Information Retrieval.* 2009: p. 38.
5. Berger, S., *Great Age Guide to the Internet*. 2005.
6. Selberg, E., Etzioni *The MetaCrawler architecture for resource aggregation on the Web*. 1997.
7. Manoj, M.J., E, *Design and Development of a Programmable Meta Search Engine*. 2013. 6–11.
8. Patel, B.a.S., D. 2014 Oct. 2014]; Available from: http://www.technicaljournalsonline.com/ijaers/VOL%20II/IJAERS%20VOL%20II%20ISSUE%20I%20%20OCTBER%20DECEMBER%202012/231.pdf.
9. Manoj, M.E., Jacob, *nformation retrieval on Internet using Metasearch engines: A review*. 2008.
10. M. and Jacob, E. 2014 27 Oct. 2014]; Available from: http://nopr.niscair.res.in/bitstream/123456789/2243/1/JSIR%2067(10)%20739-746.pdf.