LECTURE NOTES

ON

INFORMATION RETRIEVAL SYSTEMS

IV B.TECH I SEMESTER (JNTUH-R15)

Dr I Surya Prabha
Professor

INFORMATION TECHNOLOGY
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD-500 043

UNIT - I

RETRIEVAL STRATEGIES

INTRODUCTION:

- Information Retrieval System is a system it is a capable of storing, maintaining from a system and retrieving of information. This information may take any of the form that is audio, video, text.
- Information Retrieval System is mainly focus electronic searching and retrieving of documents.
- Information Retrieval is a activity of obtaining relevant documents based on user needs from collection of retrieved documents.

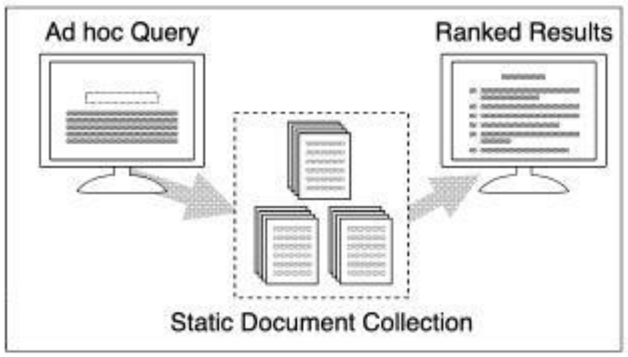


Fig: shows basic information retrieval system

- A static, or relatively static, document collection is indexed prior to any user query.
- A query is issued and a set of documents that are deemed relevant to the query are ranked based on their computed similarity to the query and presented to the user query.
- Information Retrieval (IR) is devoted to finding *relevant* documents, not finding simple matches to patterns.
- A related problem is that of document routing or filtering. Here, the queries are static and the document collection constantly changes. An environment where corporate e-mail is routed based on predefined queries to different parts of the organization (i.e., e-mail about sales is routed to the sales department, marketing e-mail goes to marketing, etc.) is an example of an application of document routing. Figure illustrates document routing

•

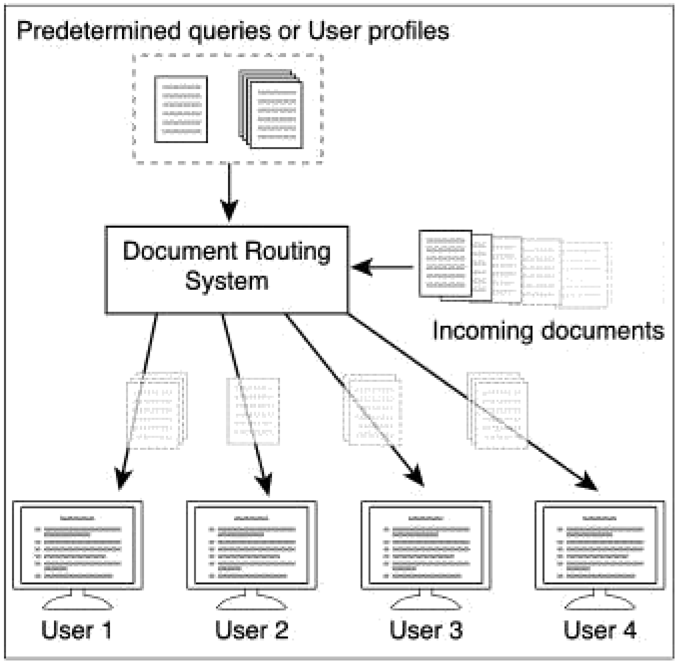


Fig: Document routing algorithms

PRECISION AND RECALL:

In Figure we illustrate the critical document categories that correspond to any issued query. Namely, in the collection there are documents which are retrieved, and there are those documents that are relevant. In a perfect system, these two sets would be equivalent; we would only retrieve relevant documents. In reality, systems retrieve many non-relevant documents. To measure effectiveness, two ratios are used: *precision* and *recall.* Precision is the ratio of the number of relevant documents retrieved to the total number retrieved. Precision provides an indication of the quality of the answer set. However, this does not consider the total number of relevant documents. A system might have good precision by retrieving ten documents and finding that nine are relevant (0.9 precision), but the total number of relevant documents also matters. If there were only nine relevant documents, the system would be a huge success.however if millions of documents were relevant and desired, this would not be a good result set.

Recall considers the total number of relevant documents; it is the ratio of the number of relevant documents retrieved to the total number of documents in the collection that are believed to be relevant. Computing the total number of relevant documents is non-trivial.

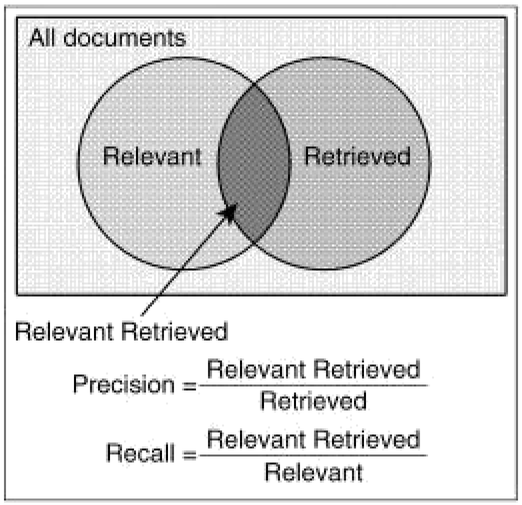

All documents
Relevant
Retrieved
Relevant Retrieved
Precision = Relevant Retrieved / Retrieved
Recall = Relevant Retrieved / Relevant

Fig: PRECISION AND RECALL

1. RETRIEVAL STRATEGIES:

Retrieval strategies assign a measure of similarity between a query and a document. These strategies are based on the common notion that the more often terms are found in both the document and the query, the more "relevant" the document is deemed to be to the query. Some of these strategies employ counter measures to alleviate problems that occur due to the ambiguities inherent in language-the reality that the same concept can often be described withmany different terms.
A retrieval strategy is an algorithm that takes a query Q and a set of documents *D1 , D2 , ... , Dn* and identifies the Similarity Coefficient *SC(Q,Di)* for each of the documents 1 :s: i :s: *n*

The retrieval strategies identified are:

1.1 Vector Space Model

Both the query and each document are represented as vectors in the term space. A measure of the similarity between the two vectors is computed. The vector space model computes a measure of similarity by defining a vector that represents each document, and a vector that represents the query The model is based on the idea that, in some rough sense, the meaning of a document is conveyed by the words used. If one can represent the words in the document by a vector, it is possible to compare documents with queries to determine how similar their content is. If a query is considered to be like a document, a similarity coefficient (SC) that measures the similarity between a document and a query can be computed. Documents whose content, as measured by the terms in the document, correspond most closely to the content of the query are judged to be the most relevant.

Figure illustrates the basic notion of the vector space model in which vectors that represent a query and three documents are illustrated.

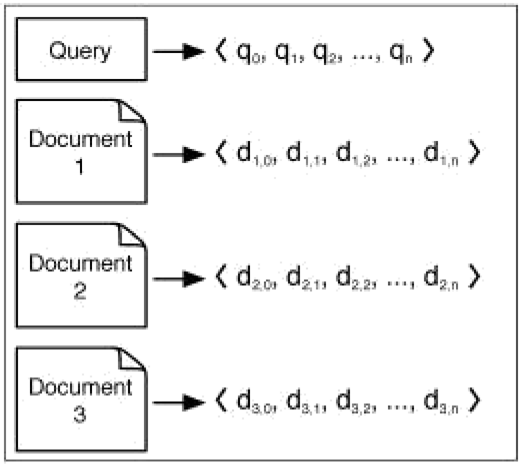

Query
⟨ q0, q1, q2, ..., qn ⟩
Document 1
⟨ d1,0, d1,1, d1,2, ..., d1,n ⟩
Document 2
⟨ d2,0, d2,1, d2,2, ..., d2,n ⟩
Document 3
⟨ d3,0, d3,1, d3,2, ..., d3,n ⟩

Fig: vector space model

The simplest means of constructing a vector is to place a one in the corresponding vector component if the term appears, and a zero if the term does not appear. Consider a document, *D1*, that contains two occurrences of term *CY* and zero occurrences of term (3. The vector < 1,0 > represents this document using a binary representation. This binary representation can be used to produce a similarity coefficient, but it does not take into account the frequency of a term within a document. By extending the representation to include a count of the number of occurrences of the terms in each component, the frequency of the terms can be considered. In this example, the vector would now appear as < 2,0 >.

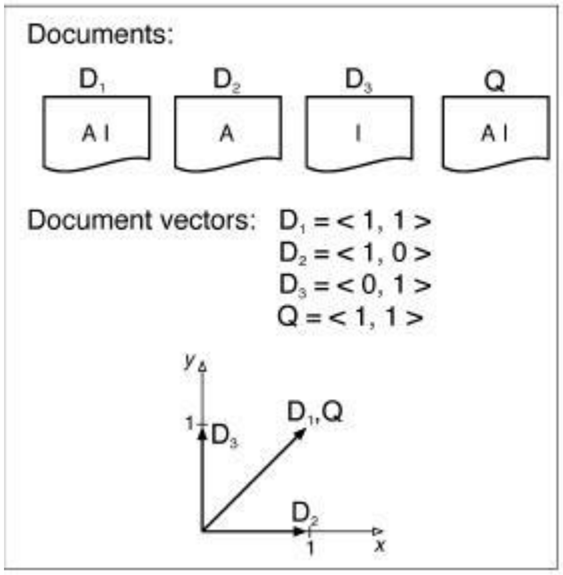


Fig: Documents and Query

This more formal definition, and slightly larger example, illustrates the use of weights based on the collection frequency. Weight is computed using the *Inverse Document Frequency (IDF)* corresponding to a given term. To construct a vector that corresponds to each document, consider the following definitions.

t = number of distinct terms in the document collection

tfij : number of occurrences of term tj in document *Di* .
This is referred to as the *term frequency.*

dfj = number of documents which contain *tj*. This is the *document frequency.*

***Idfr*= log(d/ *dfj*)** where *d* is the total number of documents. This is the *inverse document frequency.*

The vector for each document has *n* components and contains an entry for each distinct term in the entire document collection. The components in the vector are filled with weights computed for each term in the document collection. The terms in each document are automatically assigned weights based on how frequently they occur in the entire document collection and how often a term appears in a particular document. The weight of a term in a document increases the more often the term appears in one document and decreases the more often it appears in all other documents. A weight computed for a term in a document vector is non-zero only if the term appears in the document. For a large document collection consisting of numerous small documents, the document vectors are likely to contain mostly zeros. For example, a document collection with 10,000 distinct terms results in a 10,000-dimensional vector for each document. A given document that has only 100 distinct terms will have a document vector that contains 9,900 zero-valued components .The weighting factor for a term in a document is defined as a combination of term frequency, and inverse document frequency. That is, to compute the value of the *jth* entry in the vector corresponding to document i, the following equation is used:
SC

$$d_{ij} = tf_{ij} \times idf_j$$

Consider a document collection that contains a document, *D l* , with ten occurrences of the term *green* and a document, *D2,* with only five occurrences ofthe term *green.* If *green* is the only term found **in** the query, then document *Dl*is ranked higher than *D2* . When a document retrieval system is used to query a collection of documents with t distinct collection-wide terms, the system computes a vector D *(di1 , di2 , ... , dit)* of size t for each document. The vectors are filled with term weights as described above. Similarly, a vector Q *(Wql, Wq2, ... , Wqt)* is constructed for the terms found **in** the query. A simple similarity coefficient (SC) between a query Q and a document *Di* is defined by the dot product of two vectors. Since a query vector is similar **in** length to a document vector, this same measure is often used to compute the similarity between two documents. We discuss this application of an SC as it applies to document clustering.

$$SC(Q,D)=\sum w_{qj} \text{ X } d_{ij}$$

Example of Similarity Coefficient

Consider a case insensitive query and document collection with a query Q and a ocument collection consisting of the following three documents:

Q: "gold silver truck"
D l : "Shipment of gold damaged **in** a fire"
D2 : "Delivery of silver arrived **in** a silver truck"
D3: "Shipment of gold arrived **in** a truck"

In this collection, there are three documents, so *d* = 3. If a term appears **in** only one of the three documents, its *idfis* log d~j = logf = 0.477. Similarly, if a term appears **in** two of the three documents its *idfis* log ~ = 0.176, and a term which appears **in** all three documents has an *idf* of log ~ = o.The *idf* for the terms **in** the three documents is given below:
Idfa = 0
Idfarrived = 0.176
Idfdamaged = 0.477
Idfdelivery = 0.477
IdfJire = 0.477
Idfin = 0
Idfof = 0
Idfsilver = 0.477
Idfshipment = 0.176
Idftruck = 0.176
Idfgold = 0.176

Document vectors can now be constructed. Since eleven terms appear in the document collection, an eleven-dimensional document vector is constructed. The alphabetical ordering given above is used to construct the document vector so that h corresponds to term number one which is *a* and *t2* is *arrived,* etc. The weight for term i in vector j is computed as the *idfi* x *t fij.* The document

Similarly,
SC (Q, D2) = (0.954)(0.477) + (0.176)2 R:i 0.486
SC (Q, D3) = (0.176)2 + (0.176)2 R: i 0.062
Hence, the ranking would be D2, D3, D1.

Implementations of the vector space model and other retrieval strategies typically use an inverted index to avoid a lengthy sequential scan through every document to find the terms in the query.

Instead, an inverted index is generated prior to the user issuing any queries. Figure illustrates the structure of the inverted index. An entry for each of the *n* terms is stored in a structure called the index. For each term, a pointer references a logical linked list called the posting list. The posting list contains an entry for each unique document that contains the term. In the figure below, the posting list contains both a document identifier and the term frequency. The posting list in the figure indicates that term *tl* appears once in document one and twice in document ten. An entry for an arbitrary term *ti* indicates that it occurs *t f* times in document j. Details of inverted index construction and use are provided in Chapter 5, but it is useful to know that inverted indexes are commonly used to improve run-time performance of various retrieval strategies.

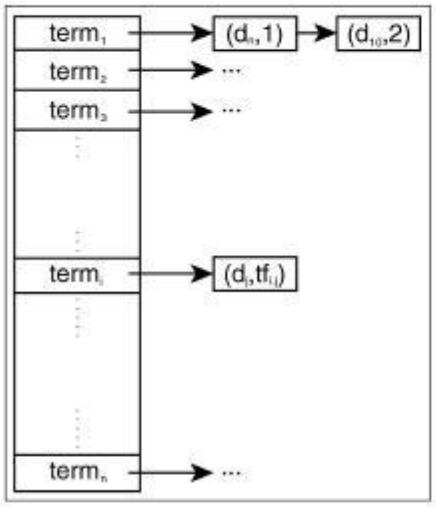


Fig: inverted index

The measure is important as it is used by a retrieval system to identify which documents aredisplayed to the user. Typically, the user requests the top *n* documents, and these are displayed ranked according to the similarity coefficient. Subsequently, work on term weighting was done to improve on the basic combination of *tf-idf* weights . Many variations were studied, and the following weight for term j in document i was identifiedas a good performer:

The motivation for this weight is that a single matching term with a high term frequency can skew the effect of remaining matches between a query and a given document. To avoid this, the *log(tf)* + 1 is used reduce the range of term frequencies. A variation on the basic theme is to use weight terms in the query differently than terms in the document. One term weighting scheme, referred to as *Inc. ltc,* was effective. It uses a document weight of (1 + *log(tf)) (idf)* and query weight of (1 + *log(tf)).* The *labellnc.ltc* is of the form: *qqq.ddd* where *qqq* refers to query weights and *ddd* refers to document weights. The three letters: *qqq* or *ddd* are of the form *xyz.* The first letter, *x,* is either *n, l,* or *a. n* indicates the "natural" term frequency or just *t f* is used. *l* indicates that the logarithm is used to scale down the weight so 1 + *log(tf)* is used. *a* indicates that an augmented weight was used where the weight is 0.5 + 0.5 x *t/f* . The second letter, *y,* indi2; tes whether or not the *idf* was used. A value of *n* indicates that no *idf* was used while a value of *t* indicates that the *idf* was used. The third letter, *z,* indicates whether or not document length normalization was used. By normalizing for document length, we are trying to reduce the impact document length might have on retrieval (see Equation 2.1). A value of *n* indicates no normalization was used, a value of c indicates the standard cosine normalization was used, and a value of *u* indicates pivoted length normalization.

1.2. Probabilistic Retrieval Strategies:

The probabilistic model computes the similarity coefficient (SC) between a query and a document as the probability that the document will be relevant to the query. This reduces the relevance ranking problem to an application of probability theory. Probability theory can be used to compute a measure of relevance between a query and a document.

1. Simple Term Weights.
2. Non binary independent model.
3. Language model.

1.2.1. Simple Term Weights:

The use of term weights is based on the Probability Ranking Principle (PRP), which assumes that optimal effectiveness occurs when documents are ranked based on an estimate of the probability of their relevance to a query The key is to assign probabilities to components of the query and then use each of these as evidence in computing the final probability that a document is relevant to the query. The terms in the query are assigned weights which correspond to the probability that a particular term, in a match with a given query, will retrieve a relevant document. The weights for each term in the query are combined to obtain a final measure of relevance. Most of the papers in this area incorporate probability theory and describe the validity of independence assumptions, so a brief review of probability theory is in order. Suppose we are trying to predict whether or not a softball team called the Salamanders will win one of its games. We might observe, based on past experience, that they usually win on sunny days when their best shortstop plays. This means that two pieces of evidence, outdoor-conditions and presence of good-shortstop, might be used. For any given game, there is a seventy five percent chance that the team will win if the weather is sunny and a sixty percent chance that the team will win if the shortstop plays.
Therefore, we write:
P (win I sunny) = 0.75
P (win I good-shortstop) = 0.6

The conditional probability that the team will win given both situations is writtenas p(win I sunny, good-shortstop). This is read "the probability that theteam will win given that there is a sunny day and the good-shortstop plays."We have two pieces of evidence indicating that the Salamanders will win. Intuition says that together the two pieces should be stronger than either alone.This method of combining them is to "look at the odds." A seventy-five percent chance of winning is a twenty-five percent chance of losing, and a sixty percent chance of winning is a forty percent chance of losing. Let us assumethe independence of the pieces of evidence. P(win I sunny, good-shortstop) = a
P (win I sunny) = (3
P (win I good-shortstop) = *r*

Note the combined effect of both sunny weather and the good-shortstop results in a higher probability of success than either individual condition. The key is the independence assumptions. The likelihood of the weather being nice and the good-shortstop showing up are completely independent. The chance the shortstop will show up is not changed by the weather. Similarly, he weather is not affected by the presence or absence of the good-shortstop. If the independence assumptions are violated suppose the shortstop prefer sunny weather - special consideration for the dependencies is required. The independence assumptions also require that the weather and the appearance of the good-shortstop are independent given either a win or a loss .For an information retrieval query, the terms in the query can be viewed as indicators that a given document is relevant. The presence or absence of query term A can be used to predict whether or not a document is relevant. Hence, after a period of observation, it is found that when term A is in both the query and the document, there is an *x* percent chance the document is relevant. We then assign a probability to term A. Assuming independence of terms this can be done for each of the terms in the query. Ultimately, the product of all the weights can be used to compute the probability of relevance. We know that independence assumptions are really not a good model of reality. Some research has investigated why systems with these assumptions For example, a relevant document that has the term *apple* in response to a query for *apple pie* probably has a better chance of having the term *pie* than some other randomly selected term. Hence, the key independence assumption is violated.

Most work in the probabilistic model assumes independence of terms because handle independencies involves substantial computation. It is unclear whether or not effectiveness is improved when dependencies are considered. We note that relatively little work has been done implementing these approaches. They are computationally expensive, but more importantly, they are difficult to estimate. It is necessary to obtain sufficient training data about term co occurrence in both relevant and non-relevant documents. Typically, it is very difficult to obtain sufficient training data to estimate these parameters. In the need for training data with most probabilistic models A query with two terms, *ql* and *q2*, is executed. Five documents are returned and an assessment is made that documents two and four are relevant. From this assessment, the probability that a document is relevant (or non-relevant) given that it contains term *ql* is computed. Likewise, the same probabilities are computed for term *q2*. Clearly, these probabilities are estimates based on training data. The idea is that sufficient training data can be obtained so that when a user issues a query, a good estimate of which documents are relevant to the query can be obtained. Consider a document, *di*, consisting of t terms *(WI, W2, ... , Wt)*, where *Wi* is the estimate that term i will result in this document being relevant. The weight or "odds" that document *di* is relevant is based on the probability of relevance for each term in the document. For a given term in a document, its contribution to the estimate of relevance for the entire document is computed as

$$(P(w_i|red))/((P(w_i|nonred))$$

The question is then: How do we combine the odds of relevance for each term into an estimate for the entire document? Given our independence assumptions, we can multiply the odds for each term in a document to obtain the odd is that the document is relevant. Taking the log of the product yields:

We note that these values are computed based on the assumption that terms will occur independently in relevant and non-relevant documents. The assumption is also made that if one term appears in a document, then it has no impact on whether or not another term will appear in the same document.

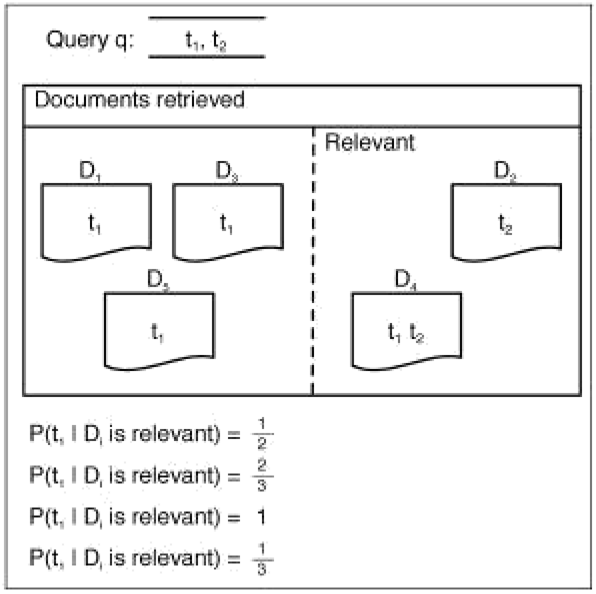

Query q: t1, t2
Documents retrieved
Relevant
D1
D3
D2
D5
D4
t1
t1
t2
t1
t1 t2
P(t, | D, is relevant) = 1/2
P(t, | D, is relevant) = 2/3
P(t, | D, is relevant) = 1
P(t, | D, is relevant) = 1/3

Fig: Documents relevant and retrieved

Now that we have described how the individual term estimates can be combined into a total estimate of relevance for the document, it is necessary to describe a means of estimating the individual term weights. Several different means of computing the probability of relevance and non-relevance for a given term were studied since the introduction of the probabilistic retrieval model.

Exclusive independence assumptions:

11: The distribution of terms in relevant documents is independent and their distribution in all documents is independent.

12: The distribution of terms in relevant documents is independent and their distribution in non-relevant documents is independent.

1: Probable relevance is based only on the presence of search terms in the documents.

2: Probable relevance is based on both the presence of search terms in documents and their absence from documents.

11 indicates that terms occur randomly within a document-that is, the presence of one term in a document in no way impacts the presence of another term in the same document. This is analogous to our example in which the presence of the good-shortstop had no impact on the weather given a win. This also states that the distribution of terms across all documents is independent un conditionally for all documents-that is, the presence of one term in a document tin no way impacts the presence of the same term in other documents. This is analogous to saying that the presence of a good-shortstop in one game has no impact on whether or not a good-shortstop will play in any other game. Similarly, the presence of good-shortstop in one game has no impact on the weather for any other game.

12 indicate that terms in relevant documents are independent-that is, they satisfy 11 and terms in non-relevant documents also satisfy 11. Returning to our example, this is analogous to saying that the independence of a good-shortstop and sunny weather holds regardless of whether the team wins or loses.01 indicates that documents should be highly ranked only if they contain matching terms in the query (i.e., the only evidence used is which query terms are actually present in the document). We note that this ordering assumption is not commonly held today because it is also important to consider when query terms are not found in the document. This is inconvenient in practice. Most systems use an inverted index that identifies for each term, all occurrences of that term in a given document. If absence from a document is required, the index would have to identify all terms *not* in a document To avoid the need to track the absence of a term in a document, the estimate makes the zero point correspond to the probability of relevance of a document lacking all the query terms-as opposed to the probability of relevance of a random document. The zero point does not mean that we do not know anything: it simply means that we have some evidence for non-relevance. This has the effect of converting the 02 based weights to presence-only weights.02 takes 01 a little further and says that we should consider both the *presence* and the *absence* of search terms in the query. Hence, for a query that asks for term *tl* and term t2-a document with just one of these terms should be ranked lower than a document with both terms

Four weights are then derived based on different combinations of these ordering principles and independence assumptions. Given a term, *t*, consider the following quantities:

N =number of documents in the collection

R= number of relevant documents for a given query *q*

n = number of documents that contain term *t*

r = number of relevant documents that contain term *t*

1.2.2 Non-Binary Independence Model:

The non-binary independence model term frequency and document length, somewhat naturally, into the calculation of term weights . Once the term weights are computed, the vector space model is used to compute an inner product for obtaining a final similarity coefficient. The simple term weight approach estimates a term's weight based on whether or not the term appears in a relevant document. Instead of estimating the probability that a given term will identify a relevant document, the probability that a *term which appears if times* will appear in a relevant document is estimated.

For example, consider a ten document collection in which document one contains the term *blue* once and document two contains ten occurrences of the term *blue.* Assume both documents one and two are relevant, and the eight other documents are not relevant. With the simple term weight model, we would compute the P(Rel I *blue)* = 0.2 because *blue* occurs in two out of ten relevant documents.
With the non-binary independence model, we calculate a separate probability for each term frequency. Hence, we compute the probability that *blue* will occur one time P(l I R) = 0.1, because it did occur one time in document one. The probability that *blue* will occur ten times is P(lO I R) = 0.1, because it did occur ten times in one out of ten documents. To incorporate document length, weights are normalized based on the size of the document. Hence, if document one contains five terms and document two contains ten terms, we recomputed the probability that *blue* occurs only once in a relevant document to the probability that *blue* occurs 0.5 times in a relevant document. The probability that a term will result in a non-relevant document is also used. The final weight is computed as the ratio of the probability that a term will occur *if* times in relevant documents to the probability that the term will occur *if* times in non-relevant documents.

More formally

where P(*di* I *R)* is the probability that a relevant document will contain *di* occurrences of the *i!h* term, and P(*di* I *N)* is the probability that a non-relevant document has *di* occurrences of the *i!h* term.

1.3. Language Models.

A statistical language model is a probabilistic mechanism for "generating" a piece of text. It thus defines a distribution over all the possible word sequences. The simplest language model is the unigram language model, which is essentially a word distribution. More complex language models might use more context information (e.g., word history) in predicting the next word if the speaker were to utter the words in a document, what is the likelihood they would then say the words in the query.
Formally, the similarity coefficient is simply:

$$SC(Q_i,Di)=P(Q|M_d)$$

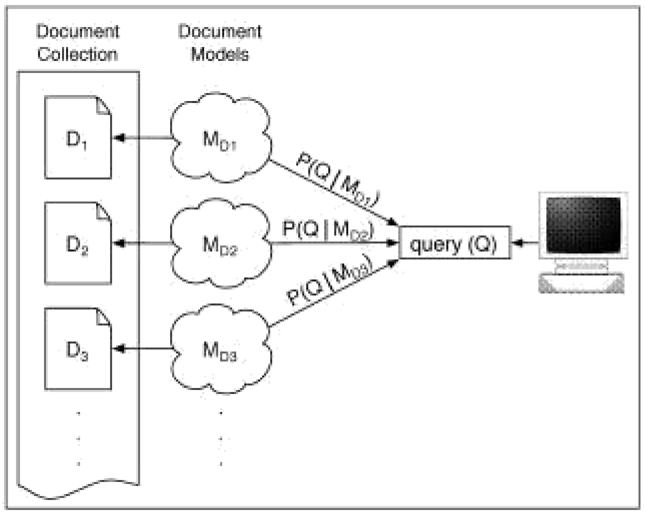
where M_d is the language model implicit in document *Di.*

There is a need to precisely define what we mean exactly by "generating" a query. That is, we need a probabilistic model for queries. One approach in is to model the presence or absence of any term as an independent Bernoulli event and view the generation of the whole query as a joint event of observing all the query terms and not observing any terms that are not present in the query. **In** this case, the probability of the query is calculated as the product of probabilities for both the terms in the query and terms absent. That is, The model p(tj| M_{di})) can be estimated in many different ways. A straightforward method is:

$$p(\ tj\ |\ M_{di}))=p_{ml}(t_j|M_{di})$$

where PmZ(tj |M_{DJ} is the maximum likelihood estimate of the term distribution (i.e., the relative term frequency), and is given by:

Fig: Documents collection



The basic idea is illustrated in Figure. The similarity measure will work, but it has a big problem. If a term in the query does not occur in a document, the whole similarity measure becomes zero Consider our small running example of a query and three documents:

Q : "gold silver truck"
D1: "Shipment of gold damaged in a fire"
D2 : "Delivery of silver arrived in a silver truck"
D3: "Shipment of gold arrived in a truck"

The term *silver* does not appear in document *D1*. Likewise, *silver* does not appear in document *D3* and *gold* does not appear in document *D2* • Hence, this would result in a similarity coefficient of zero for all three sample documents and this sample query. Hence, the maximum likelihood estimate for

1.3.1 Smoothing:

To avoid the problem caused by terms in the query that are not present in a document, various *smoothing* approaches exist which estimate non-zero values for these terms. One approach assumes that the query term could occur in this model, but simply at no higher a rate than the chance of it occurring in any other document. The ratio cft/cs was initially proposed where *eft* is the number of occurrences of term *t* in the collection, and cs is the number of terms in the entire collection. In our example, the estimate for *silver* would be 2/22 = .091. An additional adjustment is made to account for the reality that these document models are based solely on individual documents. These are relatively small sample sizes from which to build a model. To use a larger sample (the entire collection) the following estimate is proposed where *df* t is the document frequency of term *t*, which is also used in computing the idf as To improve the effectiveness of the estimates for term weights it is possible to minimize the risk involved in our estimate. We first define ft as the mean term frequency of term *t* in the document. This can be computed as ft = Pavg(t) x dld. The risk can be obtained using a geometric distribution as:

The first similarity measure described for using language models in information retrieval uses the smoothing ratio cft/cs fo r terms that do not occur in the query and the risk function as a mixing parameter when estimating the values for *w* based on small document models. The term weight is now estimated as:

UNIT-II

RETRIEVAL UTILITIES

Utilities improve the results of a retrieval strategy. Most utilities add or remove terms from the initial query in an attempt to refine the query. Others simply refine the focus of the query by using subdocuments or passages instead of whole documents. The key is that each of these utilities (although rarely presented as such) are plug-and-play utilities that operate with any arbitrary retrieval strategy.

The utilities identified are:

Relevance Feedback-The top documents found by an initial query are identified as relevant. These documents are then examined. They may be deemed relevant either by manual intervention or by an assumption that the top n documents are relevant. Various techniques are used to rank the terms. The top t terms from these documents are then added back to the original query.

Clustering-Documents or terms are clustered into groups either automatically or manually. The query is only matched against clusters that are deemed to contain relevant information. This limits the search space. The goal is to avoid non-relevant documents before the search even begins

N-grams-The query is partitioned into n-grams (overlapping or non-overlapping sequences of n characters). These are used to match queries with the document. The goal is to obtain a "fuzzier" match that would be resilient to misspellings or optical character recognition (OCR) errors. Also, n-grams are language independent.

Thesauri-Thesauri are automatically generated from text or by manual methods. The key is not only to generate the thesaurus, but to use it to expand either queries or documents to improve retrieval.

Regression Analysis- Statistical techniques are used to identify parameters that describe characteristics of a match to a relevant document. These can then be used with a regression analysis to identify the exact parameters that refine the similarity measure.

2.1 Relevance Feedback

A popular information retrieval utility is relevance feedback. The basic premise is to implement retrieval in multiple passes. The user refines the query in each pass based on results of previous queries. Typically, the user indicates which of the documents presented in response to an initial query are relevant, and new terms are added to the query based on this selection. Additionally, existing terms in the query can be re-weighted based on user feedback. This process is illustrated in Figure.

An alternative is to avoid asking the user anything at all and to simply assume the top ranked documents are relevant. Using either manual (where the user is asked) or automatic (where it is assumed the top documents are relevant) feedback, the initial query is modified, and the new query is re-executed.

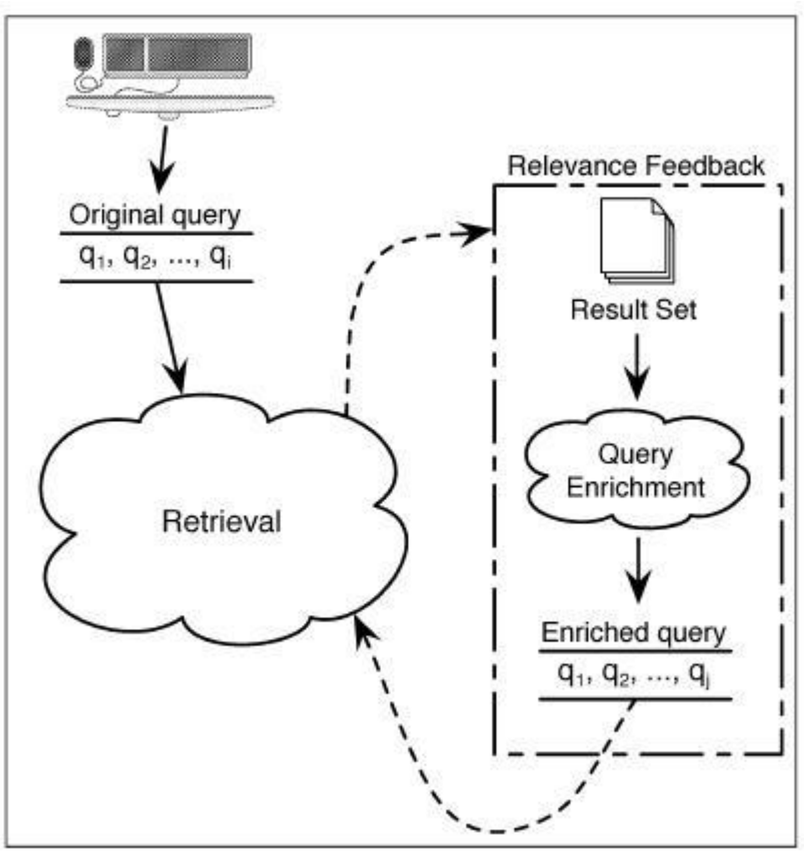


Fig: Relevance feedback process

2.1.1 Relevance Feedback in the Vector Space Model

Rocchio, in his initial paper, started the discussion of relevance feedback . Interestingly, his basic approach has remained fundamentally unchanged. Rocchio's approach used the vector space model to rank documents. The query is represented by a vector Q, each document is represented by a vector Di, and a measure of relevance between the query and the document vector is computed as SC(Q, Di), where SC is the similarity coefficient. As discussed the SC is computed as an inner product of the document and query vector or the cosine of the angle between the two vectors. The basic assumption is that the user has issued a query Q and retrieved a set of documents. The user is then asked whether or not the documents are relevant. After the user responds, the set R contains the nl relevant document vectors, and the set S contains the n2 non-relevant document vectors. Rocchio builds the new query Q' from the old query Q using the equation given below:

$$Q^{1=}Q+(1/n_1)\sum R_i-(1/n_2)\sum S_i$$

Ri and Si are individual components of R and S, respectively.

The document vectors from the relevant documents are added to the initial query vector, and the vectors from the non-relevant documents are subtracted. If all documents are relevant, the third term does not appear. To ensure that the new information does not completely override the original query, all vector modifications are normalized by the number of relevant and non-relevant documents. The process can be repeated such that Qi+1 is derived from Qi for as many iterations as desired. The idea is that the relevant documents have terms matching those in the original query. The weights corresponding to these terms are increased by adding the relevant document vector. Terms in the query that are in the nonrelevant documents have their weights decreased. Also, terms that are not in the original query (had an initial component value of zero) are now added to the original query. In addition to using values n1 and n2, it is possible to use arbitrary weights.

Not all of the relevant or non-relevant documents must be used. Adding thresholds na and nb to indicate the thresholds for relevant and non-relevant vectors results in:

The weights a, ,8, and, are referred to as Rocchio weights and are frequently mentioned in the annual proceedings of TREe. The optimal values were experimentally obtained, but it is considered common today to drop the use of nonrelevant documents (assign zero to ,) and only use the relevant documents. This basic theme was used by Ide in follow-up research to Rocchio where the following equation was defined:

Another intresting case when q retrieves only non-relevant documents then an arbitrary weight should be added to most frequently occurring term.This increases weight of term .By increasing weight of term it yields some relevant documents.This approach is applied only in manual relevance feedback and not in automatic relevance feedback.

2.1.2 Relevance Feedback in the Probabilistic Model

In probabilistic model the terms in the document are treated as evidence that a document is relevant to a query. Given the assumption of term independence, the probability that a document is relevant is computed as a product of the probabilities of each term in the document matching a term in the query. The probabilistic model is well suited for relevance feedback because it is necessary to know how many relevant documents exist for a query to compute the term weights.

Typically, the native probabilistic model requires some training data for which relevance information is known. Once the term weights are computed, they are applied to another collection. Relevance feedback does not require training data. Viewed as simply a utility instead of a retrieval strategy, probabilistic relevance feedback "plugs in" to any existing retrieval strategy. The initial query is executed using an arbitrary retrieval strategy and then the relevance information obtained during the feedback stage is incorporated.

For example, the basic weight used in the probabilistic retrieval strategy is:

$$w_i = \log \frac{\frac{r_i}{R-r_i}}{\frac{n_i-r_i}{(N-n_i)-(R-r_i)}}$$

where:

W_i -weight of term i in a particular query R -number

of documents that are relevant to the query

N -number of documents in the collection

r_I - number of relevant documents that contain term

i n_i -number of documents that contain term i

R and r cannot be known at the time of the initial query unless training data with relevance information is available

2.1.2.1 Initial Estimates

The initial estimates for the use of relevance feedback using the probabilistic model have varied widely. Some approaches simply sum the idf as an initial first estimate. Wu and Salton proposed an interesting extension which requires the use of training data. For a given term t, it is necessary to know how many documents are relevant to term t for other queries. The following equation estimates the value of r i prior to doing a retrieval:

$$ri = a + b\log f$$

Where f is the frequency of the term across the entire document collection.

After obtaining a few sample points, values for a and b can be obtained by a least squares curve fitting process. Once this is done, the value for ri can be estimated given a value of f, and using the value of ri, an estimate for an initial weight (IW) is obtained. The initial weights are then combined to compute a similarity coefficient. In the paper [Wu and Salton, 1981] it was concluded (using very small collections) that idf was far less computationally expensive, and that the IW resulted in slightly worse precision and recall.

2.1.2.2 Computing New Query Weights

For,query Q,Document D and t terms in D,D_i is binary.If the term is present then place 1 otherwise place 0.

Using relevance feedback, a query is initially submitted and some relevant documents might be found in the initial answer set. The top documents are now examined by the user and values for r i and R can be more accurately estimated (the values for ni and N are known prior to any retrieval). Once this is done, new weights are computed and the query is executed again. Wu and Salton tested four variations of composing the new query:

1. Generate the new query using weights computed after the first retrieval.
2. Generate the new query, but combine the old weights with the new. Wu suggested that the weights could be combined as:

Where

Q-old weights

T-weights computed during first pass

β-scaling factor that inducates importance of initial weights

The ratio of relevant documents retrieved to relevant documents available collection-wide is used for this value

A query that retrieves many relevant documents should use the new weights more heavily than a query that retrieves only a few relevant documents.

3. Expand the query by combining all the terms in the original query with all the terms found in the relevant documents. The weights for the new query are used as in step one for all of the old terms (those that existed in the original query and in the relevant documents). For terms that occurred in the original query, but not in any documents retrieved in the initial phase, their weights are not changed. This is a fundamental difference from the work done by

4. Expand the query using a combination of the initial weight and the new weight. This is similar to variation number two above. Assuming ql to qm are the weights found in the m components of the original query, and m - n new

Here the key element of the idf is used as the adjustment factor instead of the crude 0.5 assumption.

2.1.2.3 Partial Query Expansion

The initial work done by Wu and Salton in 1981 either used the original query and reweighted it or added all of the terms in the initial result set to the query and computed the weights for them. The idea of using only a selection of the terms found in the top documents was presented. Here the top ten documents were retrieved. Some of these documents were manually identified as relevant. The question then arises as to which terms from these documents should be used to expand the initial query. Harman sorted the terms based on six different sort orders and, once the terms were sorted, chose the top twenty terms. The sort order had a large impact on effectiveness. Six different sort orders were tested on the small Cranfield collection.

In many of the sort orders a noise measure, n, is used. This measure, for the k_{th} term is computed as:

$$n_k = \sum_{i=1} N \frac{tf_{ik}}{f_k} \log_2 f_k t f_{ik}$$

t $_{fik}$ -number of occurrences of term i in document k

f_k -number of occurrences of term k in the collection

N -number of terms in the collection

This noise value increases for terms that occur infrequently in many documents, but frequently across the collection. A small value for noise occurs if a term occurs frequently in the collection. It is similar to the idf, but the frequency within individual documents is incorporated.

Additional variables used for sort orders are:

P_k number of documents in the relevant set that contain term k

rt f_k number of occurrences of term k in the relevant set

A modified noise measure, rn_k. is defined as the noise within the relevant set.

This is computed as:

$$rn_k = \sum_{i=1} p_k \frac{tf_{ik}}{f_k} \log_2 f_k t f_{ik}$$

Various combinations of rnk, nk. and Pk were used to sort the top terms. The six sort orders tested were:

- n_k
- P_k
- r_{nk}
- n_k x rtf_k
- n_k x f_k x P_k
- n_k x f_k

Six additional sort orders were tested.

RTj - total number of documents retrieved for query j,

df_i - document frequency or number of documents in the collection that contain term

i, N - number of documents in the collection.

This gives additional weight to terms that appear in multiple documents of the initial answer set.

•

r_{ij} - number of retrieved relevant documents for query j that have term i.

R_j-number of retrieved relevant documents for query j.

This gives additional weight to terms that occur in many relevant documents and which occur infrequently across the entire document collection.

W_{ij} - term weight for term i in query j.

P_{ij}-The probability that term i is assigned within the set of relevant documents to query j

q_{ij} -The probability that term i is assigned within the set of non-relevant documents for query j is.

These are computed as:

$$p_{ij} = \frac{r_{ij} + 0.5}{R_j + 1.0} \qquad q_{ij} = \frac{df_i - r_i + 0.5}{N - R_j + 1.0}$$

where the theoretical foundation is based on the presumption that the term i's importance is computed as the amount that it will increase the difference between the average score of a relevant document and the average score of a nonrelevant document. The means of identifying a term weight are not specified in this work, so for this sort order, idf_j is used.

$$W_{ij}(p_{ij}-q_{ij})$$

RTF_i is the number of occurrences of term i in the retrieved relevant documents. Essentially, sort three was found to be superior to sorts four, five, and six, but there was little difference in the use of the various sort techniques. Sorts one and two were not as effective.

2.1.2.4 Number of Feedback Iterations

The number of iterations needed for successful relevance feedback was initially tested in 1971 by Salton. His 1990 work with 72 variations on relevance feedback assumed that only one iteration of relevance feedback was used. Harman investigated the effect of using multiple iterations of relevance feedback . The top ten documents were initially retrieved. A count of the number of relevant documents was obtained, and a new set of ten documents was then retrieved. The process continued for six iterations. Searching terminates if no relevant documents are found in a given iteration. Three variations of updating term weights across iterations were used based on whether or not the counting of relevant documents found was static or cumulative. Each iteration used the basic strategy of retrieving the top ten documents, identifying the top 20 terms, and reweighting the terms.

The three variations tested were:

• Cumulative count-counts relevant documents and term frequencies within relevant documents. It accumulates across iterations

• Reset count-resets the number of relevant documents and term frequencies within relevant documents are reset after each iteration

• Reset count, single iteration term---counts are reset and the query is reset such that it only contains terms from the current iteration

In each case, the number of new relevant documents found increased with each iteration. However, most relevant documents were found in the first two iterations.On average, iterations 3, 4, 5, and 6 routinely found less than one new relevant document per query.

2.1.2.5 User Interaction

The initial work in relevance feedback assumed the user would be asked to determine which documents were relevant to the query. Subsequent work assumes the top n documents are relevant and simply uses these documents. An interesting user study, done by Spink, looked at the question of using the top documents to suggest terms for query expansion, but giving the user the ability to pick and choose which terms to add . Users were also studied to determine how much relevance feedback is used to add terms as compared to other sources. The alternative sources for query terms were:

• Original written query

• User interaction-discussions with an expert research user or "intermediary" prior to the search to identify good terms for the query

• Intermediary-suggestion by expert users during the search

• Thesaurus

• Relevance feedback-selection of terms could be selected by either the user or the expert intermediary

Users chose forty-eight terms (eleven percent) of their search terms (over forty queries) from relevance feedback. Of these, the end-user chose fifteen and the expert chose thirty-three. This indicates a more advanced user is more likely to take advantage of the opportunity to use relevance feedback.

Additionally, the study identified which section of documents users found terms for relevance feedback. Some eighty-five percent of the relevance feedback terms came from the title or the descriptor fields in the documents, and only two terms came from the abstract of the document. This study concluded that new systems should focus on using only the title and descriptor elements of documents for sources of terms during the relevance feedback stages.

2. 2 Clustering

Document clustering attempts to group documents by content to reduce the search space required to respond to a query. For example, a document collection that contains both medical and legal documents might be clustered such that all medical documents are placed into one cluster, and all legal documents are assigned to a legal cluster. A query over legal material might then be directed (either automatically or manually) to the legal document cluster.

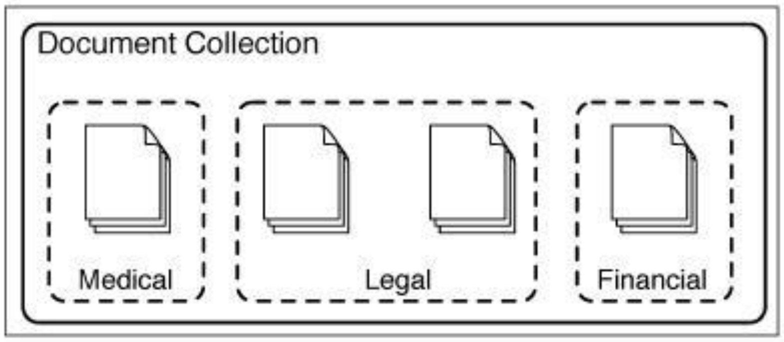


Fig : Document clustering

Several clustering algorithms have been proposed. In many cases, the evaluation of clustering algorithms has been challenging because it is difficult to automatically point a query at a

document cluster. Viewing document clustering as a utility to assist in ad hoc document retrieval, we now focus on clustering algorithms and examine the potential uses of these algorithms in improving precision and recall of ad hoc and manual query processing. Another factor that limits the widespread use of clustering algorithms is their computational complexity. Many algorithms begin with a matrix that contains the similarity of each document with every other document. For a 1,000,000 document collection, this matrix has $(1,000,000 /2)^2$different elements. Each of these pair-wise similarity calculations is computationally expensive due to the same factors found in the traditional retrieval problem. Initial work on a Digital Array Processor (DAP) was done to improve run-time performance of clustering algorithms by using parallel processing subsequently , these algorithms were implemented on a parallel machine with a torus interconnection network. Clusters are formed with either a top-down or bottom-up process. In a top-down approach, the entire collection is viewed as a single cluster and is partitioned into smaller and smaller clusters. The bottom-up approach starts with each document being placed into a separate cluster of size one and these clusters are then glued to one another to form larger and larger clusters. The bottom up approach is referred to as hierarchical agglomerative because the result of the clustering is a hierarchy (as clusters are pieced together, a hierarchy emerges). Other clustering algorithms, such as the popular K-Means algorithm, use an iterative process that begins with random cluster centroids and iteratively adjusts them until some termination condition is met. Some studies have found that hierarchical algorithms, particularly those that use group-average cluster merging schemes, produce better clusters because of their complete
Document-to-document comparisons. More recent work has indicated that this may not be true across all metrics and that some combination of hierarchical and iterative algorithms yields improved effectiveness .As these studies use a variety of different experiments, employ different metrics and (often very small) document collections, it is difficult to conclude which clustering method is definitively superior.

2.2.1 Result Set Clustering

Clustering was used as a utility to assist relevance feedback.In those cases only the results of a query were clustered (a much smaller document set), and in the relevance feedback process, by only new terms from large clusters were selected.Recently, Web search results were clustered based on significant phrases in the result set . First, documents in a result set are parsed, and two term phrases are identified. Characteristics about these phrases are then used as input to a model built by various learning algorithms (e.g.; linear regression, logistic regression, and support vector regression are used in this work). Once the most significant phrases are identified they are used to build clusters. A cluster is initially identified as the set of documents that contains one of the most significant phrases. For example, if a significant phrase contained the phrase "New York", all documents that contain this phrase would be initially placed into a cluster. Finally, these initial clusters are merged based on document-document similarity.

2.2.2 Hierarchical Agglomerative Clustering

First the N x N document similarity matrix is formed. Each document is placed into its own cluster. The following two steps are repeated until only one cluster exists.

• The two clusters that have the highest similarity are found.

• These two clusters are combined, and the similarity between the newly formed cluster and the remaining clusters recomputed.

As the larger cluster is formed, the clusters that merged together are tracked and form a hierarchy.

Assume documents A, B, C, D, and E exist and a document-document similarity matrix exists.
At this point, each document is in a cluster by itself:

{{A} {B} {C} {D} {E}}

We now assume the highest similarity is between document A and document B. So the contents of the clusters become:

{{A,B} {C} {D} {E}}

After repeated iterations of this algorithm, eventually there will only be a single cluster that consists of {A,B,C,D,E}. However, the history of the formation of this cluster will be known.
The node {AB} will be a parent of nodes {A} and {B} in the hierarchy that is formed by clustering since both A and B were merged to form the cluster {AB}.

Hierarchical agglomerative algorithms differ based on how {A} is combined with {B} in the first step. Once it is combined, a new similarity measure is computed that indicates the similarity of a document to the newly formed cluster {AB}

2.2.2.1 Single Link Clustering

The similarity between two clusters is computed as the maximum similarity between any two documents in the two clusters, each initially from a separate cluster. Hence, if eight documents are in cluster A and ten are in cluster B, we compute the similarity of A to B as the maximum similarity between any of the eight documents in A and the ten documents in B.

2.2.2.2 Complete Linkage

Inter-cluster similarity is computed as the minimum of the similarity between any documents in the two clusters such that one document is from each cluster.

2.2.2.3 Group Average

Each cluster member has a greater average similarity to the remaining members of that cluster than to any other cluster. As a node is considered for a cluster its average similarity to all nodes in that cluster is computed. It is placed in the cluster as long as its average similarity is higher than its average similarity for any other cluster.

2.2.2.4 Ward's Method

Clusters are joined so that their merger minimizes the increase in the sum of the distances from each individual document to the centroid of the cluster containing it. The centroid is defined as the average vector in the vector space. If a vector represents the ith document,Di =< tl, t2, ... , tn >, the centroid C is written as C =< CI, C2, ... , Cn >.The jth element of the centroid vector is computed as the average of all of the jth elements of the document vectors:

$$c_j = \frac{\sum_{i=1}^{n} t_{ij}}{n}$$

Hence, if cluster A merged with either cluster B or cluster C, the centroids for the potential cluster AB and AC are computed as well as the maximum distance of any document to the centroid. The cluster with the lowest maximum is used.

2.2.2.5 Analysis of Hierarchical Clustering Algorithms

Ward's method typically took the longest to implement these algorithms, with single link and complete linkage being somewhat similar in run-time .A summary of several different studies on clustering is given in . Clusters in single link clustering tend to be fairly broad in nature and provide lower effectiveness. Choosing the best cluster as the source of relevant documents resulted in very close effectiveness results for complete link, Ward's, and group average clustering. A consistent drop in effectiveness for single link clustering was noted.

2.2.3 Clustering Without a Precomputed Matrix

Other approaches exist in which the N x N similarity matrix indicates that the similarity between each document and every other document is not required.These approaches are dependent upon the order in which the input text is received, and do not produce the same result for the same set of input files.

2.2.3.1 One-Pass Clustering

One approach uses a single pass through the document collection. The first document is assumed to be in a cluster of size one. A new document is read as input, and the similarity between the new document and all existing clusters is computed. The similarity is computed as the distance between the new doc ument and the centroid of the existing clusters. The document is then placed into the closest cluster, as long as it exceeds some threshold of closeness. This approach is very dependent on the order of the input. An input sequence of documents 1,2, ... ,10 can result in very different clusters than any other of the (10! - 1) possible orderings.

Since resulting clusters can be too large, it may be necessary to split them into smaller clusters. Also, clusters that are too small may be merged into larger clusters.

2.2.3.2 Rocchio Clustering

Rocchio developed a clustering algorithm, in which all documents are scanned and defined as either clustered or loose. An unclustered document is tested as a potential center of a cluster by examining the density of the document and thereby requiring that n_l documents have a similarity coefficient of at least P_l and at least n_2 documents have a correlation of P_2. The similarity coefficient Rocchio most typically used was the cosine coefficient. If this is the case, the new document is viewed as the center of the cluster and the old documents in the cluster are checked to ensure they are close enough to this new center to stay in the cluster. The new document is then marked as clustered If a document is outside of the threshold, its status may change from clustered to loose. After processing all documents, some remain loose. These are added to the cluster whose centroid the document is closest to (revert to the single pass approach).

Several parameters for this algorithm were described . These included:

• Minimum and maximum documents per cluster

• Lower bound on the correlation between an item and a cluster below which an item will not be placed in the cluster. This is a threshold that would be used in the final cleanup phase of unclustered items.

- Density test parameters (n_l, n_2, P_l, P_2)
- Similarity coefficient

2.2.3.3 K-Means

The popular K-means algorithm is a partitioning algorithm that iteratively moves k centroids until a termination condition is met. Typically, these centroids are initially chosen at random. Documents are assigned to the cluster corresponding to the nearest centroid. Each centroid is then recomputed. Thealgorithm stops when the centroids move so slightly that they fall below a user-defined threshold or a required information gain is achieved for a given iteration.

2.2.3.4 Buckshot Clustering

Buckshot clustering is a clustering algorithm which runs in O(kn) time where k is the number of clusters that are generated and n is the number of documents. For applications where the number of desired clusters is small, the clustering time is close to 0(n) which is a clear improvement over the 0 (n^2) alternatives that require a document -document similarity matrix.

Buckshot clustering works by choosing a random sample of √kn documents.These √kn documents are then clustered by a hierarchical clustering algorithm (anyone will do). Using this approach, k clusters can be identified from the cluster hierarchy. The hierarchical clustering algorithms all require a DOC-DOC similarity matrix, so this step will require $O(\sqrt{kn}^2) = O(kn)$ time. Once the k centers are found, the remaining documents are then scanned and assigned to one of the k centers based on the similarity coefficient between the incoming document and each of the k centers. The entire algorithm requires on the order of 0 (kn) time, as 0 (kn) is required to obtain the centers and O(kn) is required to scan the document collection and assign each document to one of the centers. Note that buckshot clustering can result in different clusters with each running because a different random set of documents can be chosen to find the initial k centers.

2.2.3.5 Non-negative Matrix Factorization

A more recent clustering algorithm uses non-negative matrix factorization (NMF). This provides a latent semantic space where each axis represents the topic of each cluster. Documents are represented as a summation of each axis and are assigned to the cluster associated with the axis for which they have the greatest projection value .

2.2.4 Querying Hierarchically Clustered Collections

Once the hierarchy is generated, it is necessary to determine which portion of the hierarchy should be searched. A top-down search starts at the root of the tree and compares the query vector to the centroid for each subtree. The subtree with the greatest similarity is then searched. The process continues until a leaf is found or the cluster size is smaller than a predetermined threshold. A bottom-up search starts with the leaves and moves upwards. Early work showed that starting with leaves, which contained small clusters, was better than starting with large clusters. Subsequently three different bottom-up procedures were studied :

- Assume a relevant document is available, and start with the cluster that contains that document.
- Assume no relevant document is available. Implement a standard vector space query, and assume the top-ranked document is relevant. Start with the cluster that contains the top-ranked document.
- Start with the bottom level cluster whose centroid is closest to the query.

Once the leaf or bottom-level cluster is identified, all of its parent clusters are added to the answer set until some threshold for the size of the answer set is obtained.

These three bottom-up procedures were compared to a simpler approach in which only the bottom is used. The bottom-level cluster centroids are compared to the query and the answer set is obtained by expanding the top n clusters.

2.2.5 Efficiency Issues

Although the focus of this chapter is on effectiveness, the limited use of clustering algorithms compels us to briefly mention efficiency concerns. Many algorithms begin with a matrix that contains the similarity of each document with every other document. For a 1,000,000 document collection, this matrix has $\frac{1{,}000{,}000^2}{2}$ elements. Algorithms designed to improve the efficiency of clustering are given in , but at present, no TREC participant has clustered the entire document collection.

2.2.5.1 Parallel Document Clustering

Another means of improving run-time performance of clustering algorithms is to implement them on a parallel processor. Initial work on a Digital Array Processor (DAP) was done to improve the run-time of clustering algorithms by using parallel processing. These algorithms were implemented on a parallel machine with a torus interconnection network . A parallel version of the Buckshot clustering algorithm was developed that showed near-linear speedup on a network of sixteen workstations. This enables Buckshot to scale to significantly larger collections and provides a parallel hierarchical agglomerative algorithm There exists some other work specifically focused on parallel hierarchical clustering , but these algorithms often have large computational overhead or have not been evaluated for document clustering. Some work was done in developing parallel algorithms for hierarchical document clustering, however these algorithms were developed for several types of specialized interconnection networks, and it is unclear whether they are applicable to the simple bus connection that is common for many current parallel architectures.

Additional proposals use clustering as a utility to assist relevance feedback . Only the results of a query are clustered (a much smaller document set), and relevance feedback proceeds by only obtaining new terms from large clusters.

2.2.5.2 Clustering with Truncated Document Vectors

The most expensive step in the clustering process occurs when the distance between the new document and all existing clusters is computed. This is typically done by computing the centroid

of each cluster and measuring the cosine of the angle between the new document vector and the centroid of each cluster.
Later, it was shown that clustering can be done with vectors that use only a few representative terms from a document .

One means of reducing the size of the document vector is to use Latent Semantic Indexing to identify the most important components.Another means is to simply truncate the vector by removing those terms with a weight below a given threshold. No significant difference in effectiveness was found for a baseline of no truncation, or using latent semantic indexing with twenty, fifty, and one hundred and fifty terms or simple truncation with fifty terms.

2.4 N-grams

Term-based search techniques typically use an inverted index or a scan of the text. Additionally, queries that are based on exact matches with terms in a document perform poorly against corrupted documents. This occurs regardless of the source of the errors-either OCR (optical character recognition)
errors or those due to misspelling. To provide resilience to noise, n-grams were proposed. The premise is to decompose terms into word fragments of size n, then design matching algorithms that use these fragments to determine whether or not a match exists.

N-grams have also been used for detection and correction of spelling errors and text compression. A survey of automatic correction techniques is found in . Additionally, n-grams were used to determine the authorship of documents.

2.4.1 D' Amore and Mah

Initial information retrieval research focused on n-grams as presented in. The motivation behind their work was the fact that it is difficult to develop mathematical models for terms since the potential for a term that has not been seen before is infinite. With n-grams, only a fixed number of n-grams can exist for a given value of n. A mathematical model was developed to estimate the noise in indexing and to determine appropriate document similarity measures.

D' Amore and Mah's method replaces terms with n-grams in the vector space model. The only remaining issue is computing the weights for each n-gram. Instead of simply using n-gram frequencies, a scaling method is used to normalize the length of the document. D' Amore and Mah's contention was that a large document contains more n-grams than a small document, so it should be scaled based on its length.

To compute the weights for a given n-gram, D' Amore and Mah estimated the number of occurrences of an n-gram in a document. The first simplifying assumption was that n-grams occur with equal likelihood and follow a binomial distribution. Hence, it was no more likely for n-gram "ABC" to occur than "DEF." The Zipfian distribution that is widely accepted for terms is not true for n-grams. D' Amore and Mah noted that n-grams are not equally likely to occur, but the removal of frequently occurring terms from the document collection resulted in n-grams that follow a more binomial distribution than the terms.

D' Amore and Mah computed the expected number of occurrences of an ngram in a particular document. This is the product of the number of n-grams in the document (the document length) and the probability that the n-gram occurs. The n-gram's probability of occurrence is computed as the ratio of
its number of occurrences to the total number of n-grams in the document. D' Amore and Mah continued their application of the binomial distribution to derive an expected variance and, subsequently, a standard deviation for n-gram occurrences. The final weight for n-gram i in document j is:

$$w_{ij} = \frac{f_{ij} - e_{ij}}{\sigma_{ij}}$$

Where:
f_{ij}= frequency of an n-gram i in document j
e_{ij}= expected number of occurrences of an n-gram i in document
j σ_{ij} =standard deviation

The n-gram weight designates the number of standard deviations away from the expected value. The goal is to give a high weight to an n-gram that has occurred far more than expected and a low weight to an n-gram that has occurred only as often as expected.

D' Amore and Mah did several experiments to validate that the binomial model was appropriate for n-grams. Unfortunately, they were not able to test their approach against a term-based one on a large standardized corpus.

2.4.2 Damashek

Damashek expanded on D' Amore and Mah's work by implementing a five-gram- based measure of relevance Damashek's algorithm relies upon the vector space model, but computes relevance in a different fashion.Instead of using stop words and stemming to normalize the expected occurrence of n-grams, a centroid vector is used to eliminate noise. To compute the similarity between a query and a document, the following cosine measure is used:

$$SC(Q,D) = \frac{\sum_{j=1}^{t}(w_{qj} - \mu_Q)(w_{dj} - \mu_D)}{\sqrt{\sum_{j=1}^{t}(w_{qj} - \mu_q)^2 \sum_{j=1}^{t}(w_{dj} - \mu_D)^2}}$$

Here μ_q and μ_d represent centroid vectors that are used to characterize the query language and the document language. The weights, W_{qj} and W_{dj} indicate the term weight for each component in the query and the document vectors. The centroid value for each n-gram is computed as the ratio of the total number of occurrences of the n-gram to the total number of n-grams. This is the same value used by D' Amore and Mah. It is not used as an expected probability for the n-grams, but merely as a characterization of the n-gram's frequency across the document collection. The weight of a specific n-gram in a document vector is the ratio of the number of occurrences of the n-gram in the document to the total number of all of the n-grams in the document. This "within document frequency" is used to normalize based on the length of a document, and the centroid vectors are used to incorporate the frequency of the n-grams across the entire document collection. By eliminating the need to remove stop words and to support stemming, (the theory is that the stop words are characterized by the centroid so there was no need to eliminate them), the algorithm simply scans through the document and grabs n-grams without any parsing. This makes the algorithm language independent. Additionally, the use of the centroid vector provides a means of filtering out common n-grams in a document. The remaining n-grams are reverse engineered back into terms and used as automatically assigned keywords to describe a document.

2.4.3 Pearce and Nicholas

An expansion of Damashek's work uses n-grams to generate hypertext links. The links are obtained by computing similarity measures between a selected body of text and the remainder of the document.
After a user selects a body of text, the five-grams are identified, and a vector representing this selected text is constructed. Subsequently, a cosine similarity measure is computed, and the top rated documents are then displayed to the user as dynamically defined hypertext links. The user interface issues surrounding hypertext is the principal enhancement over Damashek's work. The basic idea of constructing a vector and using a centroid to eliminate noise remains intact.

2.4.4 Teufel

Teufel also uses n-grams to compute a measure of similarity using the vector space model . Stop words and stemming algorithms are used and advocated as a good means of reducing noise in the set of n-grams. However, his work varies from the others in that he used a measure of relevance that is intended to enforce similarity over similar documents. The premise was that if document A is similar to B, and B is similar to C, then A should be roughly similar to C. Typical coefficients, such as inner product, Dice, or Jaccard , are non-transitive. Teufel uses a new coefficient, H, where:

$$H=X+Y-(XY)$$

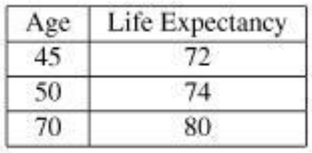
X is a direct similarity coefficient (in this case Dice was used, but Jaccard, cosine, or inner product could also have been used) and Y is an "indirect" measure that enforces transitivity. With the indirect measure, document A is identified as similar to document C. A more detailed description of the indirectsimilarity measure is given . Good precision and recall was reported for the INSPEC document collection.
Language independence was claimed in spite of reliance upon stemmingand stop words.

2.4.5 Cavnar and Vayda

Most of this work involves using n-grams to recognize postal addresses. Ngrams were used due to their resilience to errors in the address. A simple scanning algorithm that counts the number of n-gram matches that occur between a query and a single line of text in a document was used. No weighting of any kind was used, but, by using a single text line, there is no need to normalize for the length of a document. The premise is that the relevant portion of a document appears in a single line of text. Cavnar's solution was the only documented approach tested on a large standardized corpus. For the entire TIPSTER document collection, average precision of between 0.06 and 0.15 was reported. It should be noted that for the AP portion of the collection an average precision of 0.35 was obtained. These results on the AP documents caused Cavnar to avoid further tuning. Unfortunately, results on the entire collection exhibited relatively poor performance. Regarding these results, the authors claimed that,"It is unclear why there should be such variation between the retrievability of the AP documents and the other document collections."

2.5 Regression Analysis

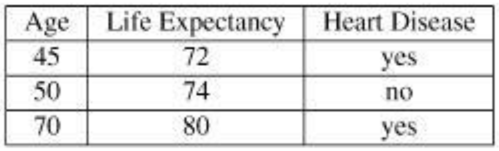
Another approach to estimating the probability of relevance is to develop variables that describe the characteristics of a match to a relevant document. Regression analysis is then used to identify the exact parameters that match the training data. For example, if trying to determine an equation that predicts a person's life expectancy given their age:



Age	Life Expectancy
45	72
50	74
70	80

A simple least squares polynomial regression could be implemented, that would identify the correct values of a and (3 to predict life expectancy (LE) based on age (A):

For a given age, it is possible to find the related life expectancy. Now, if we wish to predict the likelihood of a person having heart disease, we might obtain the following data:



Age	Life Expectancy	Heart Disease
45	72	yes
50	74	no
70	80	yes

We now try to fit a line or a curve to the data points such that if a new person shows up and asks for the chance of their having heart disease, the point on the curve that corresponds to their age ould be examined. This second example is more analogous to document retrieval because we are trying to identify characteristics in a query-document match that indicate whether or not the document is relevant. The problem is that relevance is typically given a binary (l or 0) for training data-it is rare that we have human assessments that the document is "kind of" relevant. Note that there is a basic independence assumption that says age will not be related to life expectancy (an assumption we implied was false in our preceding example). Logistic regression is typically used to estimate dichotomous variables-those that only have a small set of values, (i.e., gender, heart disease present, and relevant documents).

Focusing on information retrieval, the problem is to find the set of variables that provide some indication that the document is relevant.

Matching Terms	Size of Query	Size of Document	Relevant?
5	10	30	yes
8	20	45	no

Six variables used are given below:

• The mean of the total number of matching terms in the query.
• The square root of the number of terms in the query.
• The mean of the total number of matching terms in the document.
• The square root of the number of terms in the document.
• The average idf of the matching terms.
• The total number of matching terms in the query.

A brief overview of polynomial regression and the initial use of logistic regression is given . However, the use of logistic regression requires the variables used for the analysis to be independent. Hence, the logistic regression is given in two stages. Composite clues which are composed of independent variables are first estimated. Assume clues 1-3 above are found in one composite clue and 4-6 are in the second composite clue. The two stages proceed as follows: Stage 1:

A logistic regression is done for each composite clue.At this point the coefficients Co, C1, C2, C3 are computed to estimate the relevance for the composite clue C1. Similarly, do, d1, d2 , d3 estimate the relevance of C2.

Stage 2:

The second stage of the staged logistic regression attempts to correct for errors induced by the number of composite clues. As the number of composite clues grows, the likelihood of error increases. For N composite clues, the following logistic regression is computed:

$$logO(R|C_1, C_2, \ldots, C_N) \quad = \quad e_0 + e_1 Z + e_2 N$$

where Z is computed as the sum of the composite clues or:

$$Z = \sum_{i=1}^{N} \log O(R|C_i)$$

The results of the first stage regression are applied to the second stage. It should be noted that further stages are possible. Once the initial regression is completed, the actual computation of similarity coefficients proceeds quickly. Composite clues are only dependent on the presence or absence of terms in the document and can be precomputed. Computations based on the number of matches found in the query and the document are done at query time, but involve combining the coefficients computed in the logistic regression with the precomputed segments of the query. The question is whether or not the coefficients can be computed in a generic fashion that is resilient to changes in the document collection. The appealing aspects of this approach are that experimentation can be done to identify the best clues, and the basic independence assumptions are avoided. Additionally, the approach corrects for errors incurred by the initial logistic regression.

2.6 Thesauri

One of the most intuitive ideas for enhancing effectiveness of an information retrieval system is to include the use of a thesaurus. Almost from the dawn of the first information retrieval systems in the early 1960's, researchers focused on incorporating a thesaurus to improve precision and recall. The process of using a thesaurus to expand a query is illustrated in Figure

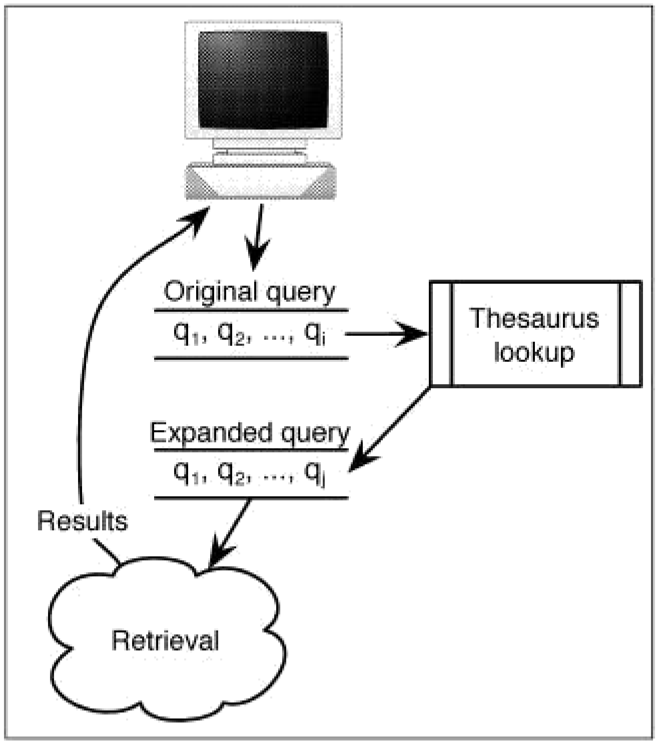


Fig: Retrieval Results

A thesaurus, at first glance, might appear to assist with a key problem-two people very rarely describe the same concepts with the same terms (i.e., one person will say that they went to a party while another person might call it a gathering). This problem makes statistical measures that rely on the number of matches between a query term and the document terms somewhat brittle when confronted with semantically equivalent terms that happen to be syntactically distinct. A query that asks for information about dogs is probably also interested in documents about canines. A document relevant to a query might not match any of the terms in the query. A thesaurus can be used either to assign a common term for all syn onyms of a term, or to expand a query to include all synonymous terms. Intuitively this should work fine, but unfortunately, results have not been promising. This section describes the use of hand-built thesauri, a very labor intensive means of building a thesaurus, as well as the quest for a sort of holy grail of information retrieval, an automatically generated thesaurus.

2.6.1 Automatically Constructed Thesauri

A hand-built thesaurus might cover general terms, but it lacks domain specific terms. A medical document collection has many terms that do not occur in a general purpose thesaurus. To avoid the need for numerous hand-built domain-specific thesauri, automatic construction methods were implemented.

2.6.1.1 Term Co-occurrence

An early discussion of automatic thesaurus is to represent each term as a vector. The terms are then compared using a similarity coefficient that measures the Euclidean distance, or angle, between the two vectors. To form a thesaurus for a given term t, related terms for t are all those terms u such that SC(t, u) is above a given threshold. Note, this is an $O(t^2)$ process so it is often common to limit the terms for which a related term list is built. This is done by using only those terms that are not so frequent that they become stop terms,but not so infrequent that there is little chance they have many synonyms.

Consider the following example:

D1 : "a dog will bark at a cat in a tree"

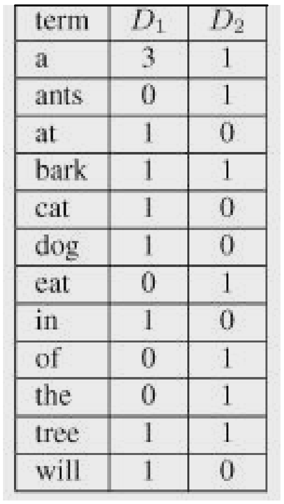
D2 : "ants eat the bark of a tree"

This results in the term-document occurrence matrix found in Table 3.1 This results in the term-document occurrence matrix found in Table .

To compute the similarity of term i with term j, a vector of size N, where N is the number of documents, is obtained for each term. The vector corresponds to a row in the following table. A dot product similarity between "bark" and "tree" is computed as:

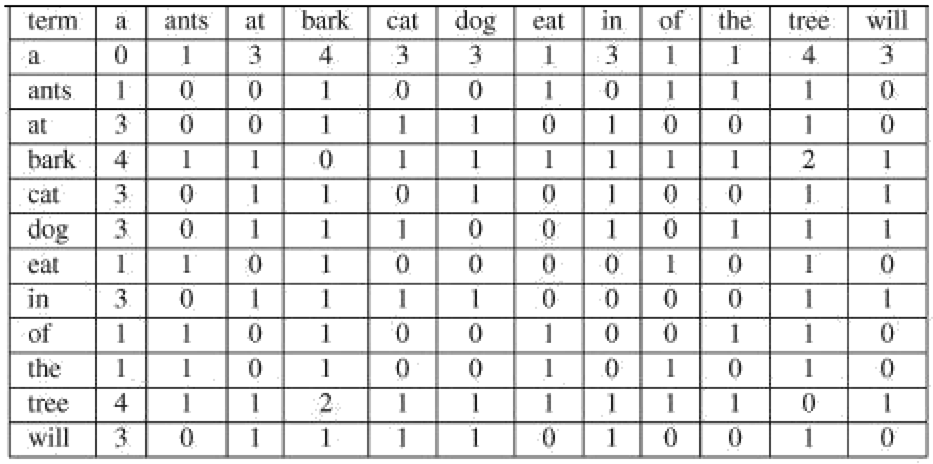
$$SC(bark,tree)=< 1\ 1 > \cdot < 1\ 1 >=2$$

The corresponding term-term similarity matrix is given in Table. The matrix is symmetric as SC(tl, t2) is equivalent to SC(t2, tl). The premise is that words are similar or related to the company they keep. Consider "tree" and "bark"; in our example, these terms co-occur twice in two documents. Hence, this pair has the highest similarity coefficient. Other simple extensions to this approach are the use of word stems instead of whole terms . The use of stemming is important here so that the term cat will not differ from cats. The tf-idf measure can be



term	D_1	D_2
a	3	1
ants	0	1
at	1	0
bark	1	1
cat	1	0
dog	1	0
eat	0	1
in	1	0
of	0	1
the	0	1
tree	1	1
will	1	0

Table:Term-Document matrix



term	a	ants	at	bark	cat	dog	eat	in	of	the	tree	will
a	0	1	3	4	3	3	1	3	1	1	4	3
ants	1	0	0	1	0	0	1	0	1	1	1	0
at	3	0	0	1	1	1	0	1	0	0	1	0
bark	4	1	1	0	1	1	1	1	1	1	2	1
cat	3	0	1	1	0	1	0	1	0	0	1	1
dog	3	0	1	1	1	0	0	1	0	1	1	1
eat	1	1	0	1	0	0	0	0	1	0	1	0
in	3	0	1	1	1	1	0	0	0	0	1	1
of	1	1	0	1	0	0	1	0	0	1	1	0
the	1	1	0	1	0	0	1	0	1	0	1	0
tree	4	1	1	2	1	1	1	1	1	1	0	1
will	3	0	1	1	1	1	0	1	0	0	1	0

used in the term-term similarity matrix to give more weight to co-occurrences between relatively infrequent terms. This summarizes much of the work done in the 1960's using term clustering, and provides some additional experiments. The common theme of these papers is that the term-term similarity matrix can be constructed, and then various clustering algorithms can be used to build clusters
of related terms. Once the clusters are built, they are used to expand the query. Each term in the original query is found in a cluster that was included in some portion or all (depending on a threshold) elements of its cluster. Much of the related work one during this time focused on different clustering algorithms and different thresholds to identify the number of terms added to the cluster. The conclusion was that the augmentation of a query using term clustering did not improve on simple queries that used weighted terms.

A domain-specific thesaurus was constructed on information about the Caenorhabditis elegans worm in support of molecular biologists. A term-term similarity measure was built with phrases and terms. A weight that used tf-idfbut also included another factor Pi, was used where Pi indicated the number of terms in phrase i. Hence, a two-term phrase was weighted double that of a single term. The new weight was:

$$w_{ij} = tf_{ij} \times \log\left(\frac{N}{df_i} \times p_i\right)$$

Using this new weight, an asymmetric similarity coefficient was also developed. The premise was that the symmetric coefficients are not as useful for ranking because a measurement between t_i t_j can become very skewed if either t_i or t_j occurs frequently. The asymmetric coefficient allows for a ranking of an arbitrary term t_i, frequent or not, with all other terms. Applying a threshold to the list means that for each term, a list of other related terms is generated-and this can be done for all terms.

The measurement for SC (t_i,t_j) is given as:

$$SC(t_i,t_j) = \left(\frac{\sum_{k=1}^{n} min(tf_{ik}, tf_{jk}) \log\left(\frac{N}{df_{ij}} \times p_j\right)}{\sum_{k=1}^{n} w_{ik}}\right) \times W_j$$

where df_{ij} is the number of co-occurrences of term i with term j. Two additional weights make this measure asymmetric: Pj and Wj . As we have said Pj is a small weight included to measure the size of term j. With all other weights being equal, the measure: SC (food, apple pie) > SC(food, apple) since phrases are weighted higher than terms. The weighting factor, Wj, gives additional preference to terms that occur infrequently without skewing the relationship between term i and term j. The weight Wj is given as:

$$W_j = \left(\frac{\log\left(\frac{N}{df_j}\right)}{\log N}\right)$$

Consider the term york and its relationship to the terms new and castle. Assume new occurs more frequently than castle. With all other weights being equal, the new weight, Wj, causesthe following to occur:

SC (york,castle) > SC (York,new)

This is done without regard for the frequency of the term york. The key is that we are trying to come up with a thesaurus, or a list of related terms, for a given term (i.e., york). When we are deriving the list of terms for new we might find that york occurs less frequently than castle so we would have:

SC (new, york) > SC (new, castle)

Note that we were able to consider the relative frequencies of york and castle with this approach. In this case:

SC (new, york) = SC (york, new)

The high frequency of the term new drowns out any real difference between york and castle-or at least that is the premise of this approach. We note in our example, that new york would probably be recognized as a phrase, but that is not really pertinent to this example. Hence, at this point, we have defined SC (ti,tj). Since the coefficient is asymmetric we now give the definition of SC(tj, ti):

A threshold was applied so that only the top one hundred terms were used for a given term. These were presented to a user. For relatively small document collections, users found that the thesaurus assisted their recall. No testing of generic precision and recall for automatic retrieval was measured.

2.6.1.2 Term Context

Instead of relying on term co-occurrence, some work uses the context (surrounding terms) of each term to construct the vectors that represent each term]. The problem with the vectors given above is that they do not differentiate the senses of the words. A thesaurus relates words to different senses. In the example given below, "bark" has two entirely different senses. A typical thesaurus lists "bark" as:

bark-surface of tree (noun)
bark-dog sound (verb

Ideally an automatically generated thesaurus would have separate lists of synonyms. The term-term matrix does not specifically identify synonyms, and Gauch and Wang do not attempt this either. Instead, the relative position of nearby terms is included in the vector used to represent a term.

The key to similarity is not that two terms happen to occur in the same document; it is that the two terms appear in the same context-that is they have very similar neighboring terms. Bark, in the sense of a sound emanating from a dog, appears in different contexts than bark, in the sense of a tree surface. Consider the following three sentences:

s1: "The dog yelped at the cat."
s2 : "The dog barked at the cat."
s3 : "The bark fell from the tree to the ground."

In sentences s1 and s2, yelped is a synonym for barked, and the two terms occur in exactly the same context. It is unlikely that another sense of bark would appear in the same context. "Bark" as a suiface of tree more commonly would have articles at one position to the left instead of two positions to the left, etc.

To capture the term's context, it is necessary to identify a set of context terms. The presence or absence of these terms around a given target term will determine the content of the vector for the target term. The authors assume the highest frequency terms are the best context terms, so the 200 most frequent terms (including stop terms) are used as context terms. A window of size seven was used. This window includes the three terms to the left of the target term and the three terms to the right of the target term. The new vector that represents target term i will be of the general form:

The Vi vectors are all concatenated to form the entire Ti vector for the term. For a simple example, we build the context vectors for the terms bark and yelp based on the document collection s1, s2, and s3 . To simplify the example,we assume that stemming is done to normalize bark and barked and that the and at are the only two context terms occupying components one and two, respectively, of the context vectors. For our test document collection we would obtain:

$$T_{bark} = [< 00 >< 10 >< 10 >< 01 >< 10 >< 10 >]$$

$$T_{yelp} = [< 00 >< 10 >< 00 >< 01 >< 10 >< 00 >]$$

The matching of s1 and s2 is the driving force between the two vectors being very similar. The only differences occur because of the additional word sense that occurs in s3 .

This example uses the frequency of occurrence of a context term as the component of the context vectors. The authors use a measure that attempts to place more weight on context terms that occur
less frequently than might be expected. The actual component value of the jth component of vector Vi, is a mutual information measure. Let:

$$\begin{aligned} df_{ij} &= \text{frequency of co-occurrence of context term} \\ &\quad\; j \text{ with target term } i \\ tf_i &= \text{total occurrences of context term } i \\ &\quad\; \text{in the collection} \\ tf_j &= \text{the total occurrences of context term } j \\ &\quad\; \text{in the collection} \\ \\ v_{ij} &= \log\left(\frac{N df_{ij}}{(tf_i)(tf_j)} + 1\right) \end{aligned}$$

This gives a higher weight to a context term that appears more frequently with a given target term than predicted by the overall frequencies of the two terms.

2.6.1.3 Clustering with Singular Value Decomposition

First a matrix, A, is computed for terms that occur 2000-5000 times. The matrix contains the number of times these terms co-occur with a term window of size k (k is 40 in this work). Subsequently, these terms are clustered into 200 A-classes (group average agglomerative clustering is used). For example, one A-class, g_{Al}, might have terms (tl, t2, t3) and another, g_{A2}, would have (t4,t5).

Subsequently, a new matrix, B, is generated for the 20,000 most frequent terms based on their co-occurrence between clusters found in the matrix. For example, if term tj co-occurs with term tl ten times, term t2 five times, and term t4 six times, B[I, j] = 15 and B[2, j] = 6. Note the use of clusters has reduced the size of the B matrix and provides substantially more training information. The rows of B correspond to classes in A, and the columns correspond to terms. The B matrix is of size 200 x 20,000. The 20,000 columns are then clustered into 200 B-classes using the buckshot clustering algorithm] indicates the number oftimes $term_j$ co-occurs with the B-classes.

Once this is done, the C matrix is decomposed and singular values are computed to represent the matrix. This is similar to the technique used for latent semantic indexing . The SVD
is more tractable at this point since only 200 columns exist. A document is represented by a vector that is the sum of the context vectors (vectors that correspond to each column in the SVD). The context vector is used to match a query.

Another technique that uses the context vector matrix, is to cluster the query based on its context vectors. This is referred to as word factorization. The queries were partitioned into three separate clusters. A query is then run for each of the word factors and a given document is given the highest rank of the three. This requires a document to be ranked high by all three factors to receive an overall high rank. The premise is that queries are generally about two or three concepts and that a relevant document has information relevant to all of the concepts.

Overall, this approach seems very promising. It was run on a reasonably good-sized collection (the Category B portion of TIPSTER using term factorization, average precision improved from 0.27 to 0.32-an 18.5% overall improvement).

2.6.1.4 Using only Document Clustering to Generate a Thesaurus

Another approach to automatically build a thesaurus is, a document clustering algorithm is implemented to partition the document collection into related clusters. A document-document similarity coefficient is used. Complete link clustering is used here, but other clustering algorithms could be used (for more details on clustering algorithms . The terms found in each cluster are then obtained. Since they occur in different documents within the cluster, different operators are used to obtain the set of terms that correspond to a given cluster. Consider documents with the following terms:

D1=t1,t2,t3,t4
D2=t2,t4
D3=t1,t2

The cluster can be represented by the union of all the terms {t1, t2, t3, t4}, the intersection {t2}, or some other operation that considers the number of documents in the cluster that contain the term. Crouch found that simple clustering worked the best. The terms that represented the cluster now appear as a thesaurus class, in that they form the automatically generated thesaurus. The class is first reduced to obtain only the good terms. This is done by using a term discriminating function that is based on document frequency.

Queries are expanded based on the thesaurus class. Any term that occurs in the query that matches a term in the thesaurus class results in all terms in the class being added to the query. Average precision was shown to improve ten percent for the small ADI collection and fifteen percent for the Medlars collection. Unfortunately, both of these results were for small collections, and the document clustering is computationally expensive, requiring O(N^2) time, where N is the number of documents in the collection.

2.6.2 Use of Manually Generated Thesaurus

Although a manually generated thesaurus is far more time consuming to build, several researchers have explored the use of such a thesaurus to improve precision and recall.

2.6.2.1 Extended Relevance Ranking with Manual Thesaurus

A system developed in 1971 used computers to assist with the manual construction of a thesaurus at the Columbia University School of Library Service. The algorithm was essentially equivalent to a simple thesaurus editor.

Manual thesaurus construction is typically used for domain-specific thesauri. A group of experts is convened, and they are asked to identify the relationship between domain-specific terms. Ghose and Dhawle note that manual generation of these thesauri can be more difficult to build for social sciences than natural sciences given that there is more disagreement about the meaning of domain-specific terms in the social sciences . A series of handbuilt thesauri (each one was constructed by students) was described in . These thesauri were generated by the relationships between two terms-such as dog is-a animal. Ultimately the thesauri were combined into one that contained seven groups of relations. These groups were:

• Antonyms
• All relations but antonyms
• All relations
• Part-whole and set relations
• Co-location relations
• Taxonomy and synonymy relations
• Paradigmatic relations

The antonym relation identified terms that were opposites of one another (e.g., night, day) and is-part-of identifies entities that are involved in a bill-of-materials relationship (e.g., tire, automobile). Co-location contains relations between words that frequently co-occur in the same phrase or sentence. Taxonomy and synonym represent synonyms. Paradigmatic relations relate different forms of words that contain the same semantic core such as canine and dog. Experiments in adding each or all of the terms from these relations were done on a small document collection with relevance judgments obtained by the researchers conducting the study. Use of all relations, with the exception of antonyms, delivered the best average precision and recall, but there was little overall improvement.

A study done in esaurus containing three different relations: equivalence (synonym), hierarchical (is-a), and associative relationships. Recall of a fairly large (227,000) document collection composed of Finnish newspaper articles was shown to increase from 47 percent to 100 percent while precision only decreased from 62.5 percent to 51 percent. Fortunately, the work was done on a large collection; however, the thesaurus was hand-built for the test and contained only 1,011 concepts and a total of 1,573 terms. Only thirty queries were used, and the high results are clearly due to "good" terms found in the thesaurus.

Given the nature of the highly specific thesaurus, this result might be very similar in nature to the manual track of the TREC conference where participants are allowed to hand-modify the original query to include more discriminating terms. The synonym, narrower term, and related term searches all showed a 10 to 20% increase in recall from a 50% baseline. The union search (using all values) showed a rather high fifty percent increase in average precision. This work does represent one of the few studies outside of the TIPSTER collection that is run on a sizable collection. It is not clear, however, how applicable the results are to a more general collection that uses a more general thesaurus.

2.6.2.2 Extending Boolean Retrieval With a Hand Built Thesaurus

All work described attempts to improve relevance ranking using a thesaurus. We describe the extensions to the extended Boolean retrieval model as a means of including thesaurus information in a Boolean request . A description of the extended Boolean model is found Values for p were attempted, and a value of six (value suggested for standard extended Boolean retrieval by Salton in to perform the best. Results of this approach showed slightly higher effectiveness.

UNIT-III
SEMANTIC NETWORKS

Semantic networks are based on the idea that knowledge can be represented by concepts which are linked together by various relationships. A semantic network is simply a set of nodes and arcs. The arcs are labeled for the type of relationship they represent. Factual information about a given node, such as its individual characteristic (color, size, etc.), are often stored in a data structure called a frame. The individual entries in a frame are called slots A frame for a rose can take the form:

(rose
(has-color red)

(height 2 feet)

(is-a flower)

)

Here the frame rose is a single node in a semantic network containing an is-a link to the node flower. The slots has-color and height store individual proper-ties of the rose.Natural language understanding systems have been developed to read hu-man text and build semantic networks representing the knowledge stored in the text turns out that there are many concepts that are not easily represented (the most difficult ones are usually those that involve temporal or spatial reasoning). Storing information in the sentence, "A rose is a flower.", is easy to do as well as to store, "A rose is red", but semantic nets have difficulty with storing this information: "The rose grew three feet last Wednesday and was taller than anything else in the garden." Storing in-formation about the size of the rose on different dates, as well as, the relative location of the rose is often quite difficult in a semantic network. For a detailed discussion see the section on "Representational Thoms" about the large-scale knowledge representation project called eye.

Despite some of the problems with storing complex knowledge in a semantic network, research was done in which semantic networks were used to improve information retrieval. This work yielded limited results and is highly language specific; however, the potential for improvement still exists. Semantic networks attempt to resolve the mismatch problem in which the terms in a query do not match those found in a document, even though the document is relevant to the query. Instead of matching characters in the query terms with characters in the documents, the semantic distance between the terms is measured (by various measures) and incorporated into a semantic network. The premise behind this is that terms which share the same meaning appear relatively close together in a semantic network. Spreading activation is one means of identifying the distance between two terms in a semantic network. There is a close relationship between a thesaurus and a semantic network. From the standpoint of an information retrieval system, a thesaurus attempts to solve the same mismatch problem by expanding a user query with related terms and hoping that the related terms will match the document. A semantic network subsumes a thesaurus by incorporating links that indicate "is-a-synonym-of" or "is-related-to," but a semantic network can represent more complex information such as an is-a hierarchy which is not found in a thesaurus.

One semantic network used as a tool for information retrieval research is WorldNet. WorldNet is publicly available and contains frames specifically designed for words (some semantic networks might contains frames for more detailed concepts such as big-and-hairy-person). WorldNet can be found on the Web at: www.cogsci.princeton.edurwn.WordNet contains different entries for the various semantic meanings of a term. Additionally, various term relationships are stored including: synonyms, antonyms (roughly the opposite of a word), hyponyms (lexical relations such as is-a), and metonyms (is a part-of). Most nouns in WorldNet are placed in the is-a hierarchy while antonyms more commonly relate adjectives. Interestingly, less commonly known relations of entailment and troponyms are used to relate verbs. Two verbs are related by entailment when the first verb entails the second verb. For example, to buy something entails that you will pay for it. Hence, buy and pay are related by entailment. A troponym relation occurs when the two activities related by entailment must occur at the same time (temporally co-extensive) such as the pair (limp, walk). Software used to search WordNet is further described in
.It is reasonable to assume that WordNet would help effectiveness by expanding query terms with synsets found in WordNet. Initial work done by Voorhees however, failed to demonstrate an improvement in effectiveness. Even with manual selection of synsets, effectiveness was not improved when queries were expanded. A key obstacle was that terms in queries were not often found in WordNet due to their specificity-terms such as National Rifle Association are not in WordNet. Also, the addition of terms that have multiple meanings or word senses significantly degrade effectiveness. More recent work, with improvements to WordNet over time has incorporated carefully selected phrases and showed a small (roughly five percent) improvement Semantic networks were used to augment Boolean retrieval and automatic relevance ranking. We describe these approaches in the remainder of this section.

3.1 Distance Measures

To compute the distance between a single node in a semantic network and another node, a spreading activation algorithm is used. A pointer starts at each of the two original nodes and links are followed until an intersection occurs between the two points. The shortest path between the two nodes is used to compute the distance. Note that the simple shortest path algorithm does not apply here because there may be several links that exist between the same two nodes. The distance between nodes a and b is: Distance (a,b) = minimum number of edges separating a and b

3.1.1 R-distance

The problem of measuring the distance between two sets of nodes is more complex. Ideally the two sets line up, for example "large rose" and "tall flower" is one such example where "large" can be compared with "tall" and "rose" can be compared with "flower." The problem is that it is difficult to align the concepts such that related concepts will be compared. Hence, the R-distance takes all of the individual entries in each set and averages the distance between all the possible combinations of the two sets. If a document is viewed as a set of terms that are "AND"ed together, and a query is represented as a Boolean expression in disjunctive normal form, then the R-distance identifies a measure of distance between the Boolean query and the document. Also, a

NOT applied to a concept yields the distance that is furthest from the concept. Hence, for a query Q for terms ((a AND b AND c) OR (e AND f)) and Document D with terms (tl AND t2), the similarity is computed below.

SC(Q,D) is computed now as the MIN(CI, C2). Essentially, each concept represented in the query is compared to the whole document and the similarity measure is computed as the distance between the document and the closest query concept.

Formally, the R-distance of a disjunctive normal form query Q, and a document D with terms (tl' t2, ... ,tn) and Cij, indicates the jf.h term in concept I is defined as:

$$\begin{aligned} SC(Q,D) &= min\,(SC_1(c_1,D), SC_1(c_2,D), \ldots, SC_1(c_m,D)) \\ SC_1(c_i,D) &= \frac{1}{mn}\sum_{i=1}^{n}\sum_{j=1}^{m} d(t_i,c_{ij}) \\ SC(Q,D) &= 0,\ \texttt{if}\ Q = D \end{aligned}$$

3.1.2 K·distance

A subsequent distance measure referred to as the K-distance was developed. This measure incorporates weighted edges in the semantic network. The distance defined between two nodes is obtained by finding the shortest path between the two nodes (again by using spreading activation) and then summing the edges along the path. More formally the distance between terms ti and tj is obtained where the shortest path from ti to tj is: ti, Xl, X2, ... , tj. The authors treat NOT as a special case. The basic idea is to dramatically increase the weights of the arcs that connect the node that is being referenced with a NOT (referred to as separation edges). Once this is done, any paths that include this node are much longer than any other path that includes other terms not referenced by a NOT. To obtain the distance between two sets, A and B, of nodes with weighted arcs, the K-distance measure computes the minimum of the distances between each node in set A and set B. These minimum distances are then averaged. Since the weights on the arcs may not be equivalent in both directions, the distance measure from A to B is averaged with the distance from B to A. For our same query Q: «a AND b AND c) OR (e AND 0)

Assume document D has only two terms: (tl AND t2), the similarity is computed below.

$$c_1 = \frac{min(d(a,t_1),d(a,t_2)) + min(d(b,t_1),d(b,t_2)) + min(d(c,t_1),d(c,t_2))}{3}$$

$$c_2 = \frac{min(d(e,t_1),d(e,t_2)) + min(d(f,t_1),d(f,t_2))}{2}$$

SC(Q,D) is still the min(q, C2). The value of SC(D,Q) would then be obtained, and the two coefficients are then averaged to obtain the final similarity measure. The K-distance of a disjunctive normal form query Q and a document D with terms (tl' t2, ... , tn) is defined as:

$$SC(Q,D) = \frac{SC_1(Q,D) + SC_1(D,Q)}{2}$$

$$SC_1(Q,D) = min\,(SC_2(c_1,D), SC_2(c_2,D), \ldots, SC_2(c_m,D))$$

$$SC_2(c_i,D) = \frac{1}{n}\left(\sum_{j=1}^{n} min\,(d(c_{ij},t_j))\right)$$

$$SC(Q,D) = 0,\ \ \mathtt{if}\ Q = D$$

The R-distance satisfies the triangular inequality such that r-dist(a,c) is less than or equal to r-dist(a,b) + r-dist(b,c). The K-distance does not satisfy this inequality but it does make use of weights along the edges of the semantic network.

3.1.3 Incorporating Distance

Lee, et aI., incorporated a distance measure using a semantic network into the Extended Boolean Retrieval model and called it-KB-EBM for Knowledge Base-Extended Boolean Model . The idea was to take the existing Extended Boolean Retrieval model and modify the weights used to include a distance between two nodes in a semantic network. The Extended Boolean model uses a function F that indicates the weight of a term in a document. In our earlier description we simply called it Wi, but technically it could be represented as F(d, ti). Lee, et al., modified this weight by using a semantic network and then used the rest of the Extended Boolean model without any other changes. This cleanly handled the case of NOT. The primitive distance function, d(ti, tj), returns the length of the shortest path between two nodes. This indicates the conceptual closeness of the two terms. What is needed here is the conceptual distance, which is inversely proportional to the primitive distance function. Hence, the new F function uses:

$$distance^{-1}(t_i,t_j) = \frac{\lambda}{\lambda + distance(t_i,t_j)}$$

First, the function F is given for a document with unweighted terms. The new function, F(d, ti), computes the weight of term ti in the document as the average distance of ti to all other nodes in the document. The new function F is then:

$$F(d,t) = \frac{\sum_{i=1}^{n} distance^{-1}(t_i, t)}{1 + \frac{\lambda}{\lambda+1}(n-1)}$$

For existing weights for a term in a document, F is modified to include weights Wi. This is the weight of the i^th term in document d.

$$F(d,t) = \frac{\sum_{i=1}^{n} distance^{-1}(t_i, t) w_i}{1 + \frac{\lambda}{\lambda+1}(n-1)}$$

3.1.4 Evaluation of Distance Measures

All three distance measures were evaluated on four collections with nine, six, seven, and seven documents, respectively. Precision and recall were not measured, so evaluations were done using comparisons of the rankings produced by each distance. In some cases MESH was used-a medical semantic network-in other cases, the Computing Reviews Classification Scheme (CRCS) was used. Overall, the small size of the test collections and the lack of precision and recall measurements made it difficult to evaluate these measures. They are presented here due to their ability to use semantic networks. Most work done today is not focused on Boolean requests. However, all of these distance measures are applicable if the natural language request is viewed as a Boolean OR of the terms in the query. It would be interesting to test them against a larger collection with a general semantic network such as WordNet.

3.2 Developing Query Term Based on "Concepts"

Instead of computing the distance between query terms and document terms in a semantic network and incorporating that distance into the metric, the semantic network can be used as a thesaurus to simply replace terms in the query with "nearby" terms in the semantic network. Vectors of "concepts" can then be generated to represent the query, instead of term-based vectorsan algorithm was given that described a means of using this approach to improve an existing Boolean retrieval system. Terms in the original Boolean system were replaced with "concepts". These concepts were found in a semantic network that contained links to the original terms. The paper referred to the network as a thesaurus, but the different relationships existing between terms meet our definition of a semantic network. The system described used an automatically generated semantic network. The network was developed using two different clustering algorithms. The first was the standard cosine algorithm, while the second was developed by the authors and yields asymmetric links between nodes in the semantic net. Users were then able to manually traverse the semantic network to obtain good terms for the query, while the semantic nets were also used to find suitable terms to manually index new documents.

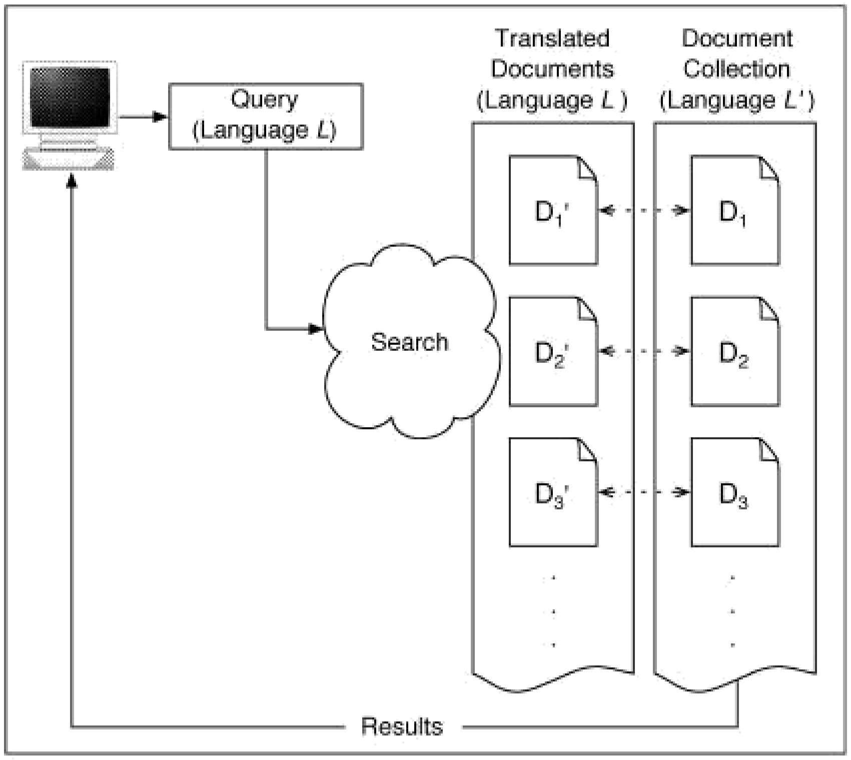


Fig :Translate the Documents

3.3 Ranking Based on Constrained Spreading Activation

Two interesting papers appeared that are frequently referenced in discussions of knowledge-based information retrieval. These describe the GRANT system in which potential funding agencies are identified based on areas of research Retrieval Utilities 139 interest. A manually built semantic network with 4,500 nodes and 700 funding agencies was constructed with links that connect agencies and areas of interest based on the topics agencies are interested in. Given a topic, the links emanating from the topic are activated and spreading activation begins. Activation stops when a funding agency is found. At each step, activation is constrained. After following the first link, three constraints are used. The first is distance. If the path exceeds a length of four ,it is no longer followed. The second is fan-out, if a path reaches a node that has more than four links emanating from it, it is not followed. This is because the node that has been reached is too general to be of much use and it will cause the search to proceed in many directions that are of little use. The third type of constraint is a rule that results in a score for the link. The score is considered an endorsement. Ultimately, the results are ranked based on the accumulation of these scores. An example of one such endorsement occurs if a researcher's area of interest is a subtopic or specialization of a general topic funded by the agency it gets a positive endorsement. An agency that funds research on database systems will fund research in temporal database systems. More formally: request-funds-for-topic(x) and IS-A(x,y) --t request-funds-for-topic(y) A negative endorsement rule exists when the area of research interest is a generalization of a funding agency's areas of research. An agency that funds database systems will probably not be interested in funding generic interest in computer science. A best-first search is used such that high-scoring endorsements are followed first. The search ends when a certain threshold number of funding agencies are identified. The GRANT system was tested operationally, and found to be superior to a simple keyword matching system that was in use. Searches that previously took hours could be done in minutes. More formal testing was done with a small set of twenty-three queries. However, the semantic network and the document collection were both relatively small so it is difficult to generalize from these results. Overall, the GRANT system is very interesting in that it uses a semantic network, but the network was constrained based on domain specific rules.

3.4 Parsing

The ability to identify a set of tokens to represent a body of text is an essential feature of every information retrieval system. Simply using every token encountered leaves a system vulnerable to fundamental semantic mismatches between a query and a document. For instance, a query that asks for information about computer chips matches documents that describe potato chips. Simple single-token approaches, both manual and automatic, are described. Although these approaches seem crude and ultimately treat text as a bag of words, they generally are easy to implement, efficient, and often result in as good or better effectiveness than many sophisticated approaches measured at the Text Retrieval Conference (TREC).

A step up from single-term approaches is the use of phrases in document retrieval. Phrases capture some of the meaning behind the bag of words and result in two-term pairs (or multi-term phrases, in the general case) so that a query that requires information about New York will not find information about the new Duke of York.
More sophisticated approaches are based on algorithms commonly used for natural language processing (NLP). These include part-of-speech taggers, syntax parsers, and information extraction heuristics. We provide a brief overview of the heuristics that are available and pay particular attention only to those that have been directly incorporated into information retrieval systems. An entire book could be written on this section as the entire field of natural language processing is relevant. Overall, it should be noted that parsing is critical to the performance of a system. For complex NLP approaches, parsing are discussed in great detail, but to date, these approaches have typically performed with no significant difference in performance than simplistic approaches

3.4.1 Single Terms

The simplest approach to search documents is to require manual intervention and to assign names of terms to each document. The problem is that it is not always easy to assign keywords that distinctly represent a document. Also, when categorizations are employed-such as the Library of Congress subject headings-it is difficult to stay current within a domain. Needless to say, the manual effort used to categorize documents is extremely high. Therefore, it was learned early in the process that manually assigned tokens did not perform significantly better than automatically assigned tokens.

Once scanning was deemed to be a good idea in, the next step was to try to normalize text to avoid simple mismatches due to differing prefixes, suffixes, or capitalization. Today, most information retrieval systems convert all text to a single case so that terms that simply start a sentence do not result in a mismatch with a query simply because they are capitalized. Stemming refers to the normalization of terms by removing suffixes or prefixes. The idea is that a user who includes the term "throw" in the query might also wish to match on "throwing", "throws", etc. Stemming algorithms have been developed for more than twenty years. The Porter and Lovins algorithms are most commonly used. These algorithms simply remove common suffixes and prefixes. A problem is that two very different terms might have the same stem. A stemmer that removes -ing and -ed results in a stem of r for terms red and ring. KSTEM uses dictionaries to ensure that any generated stem will be a valid word. Another approach uses corpus-based statistics (essentially based on term co-occurrence) to identify stems in a language-independent fashion These stemmers were shown to result in improved relevance ranking over more traditional stemmers.Stop words are terms deemed relatively meaningless in terms of document relevance and are not stored in the index. These terms represent approximately forty percent of the document collection. Removing these terms reduces index construction, time and storage cost, but may also reduce the ability to respond to some queries. A counterexample to the use of stop word removal occurs when a query requests a phrase that only con-tains stop words (e.g., "to be or not to be"). Nevertheless, stop word lists are frequently used, and some research was directed solely at determining a good stop word list. Finally, we find that other parsing rules are employed to handle special characters. Questions arise such as what to do with special characters like hyphens, apostrophes, commas, etc. Some initial rules for these questions are given, but the effect on precision and recall is not discussed. Many TREC papers talk about cleaning up their parser and the authors confess to having seen their own precision and recall results improved by very simple parsing changes. However, we are unaware of a detailed study on single-term parsing and the treatment of special characters, and its related effect on precision and recall.

3.4.2 Simple Phrases

Many TREC systems identify phrases as any pair of terms that are not separated by a stop term, punctuation mark, or special character. Subsequently, infrequently occurring phrases are not stored. In many TREC systems, phrases occurring fewer than 25 times are removed. This dramatically reduces the number of phrases which decreases memory requirements. Once phrases are employed, the question as to how they should be incorporated into the relevance ranking arises. Some systems simply add them to the query, while others do not add them to the query but do not include them in the computation of the document length normalization. The reason for this is that the terms were already being considered. Tests using just phrases or terms were performed on many systems. It was found that phrases should be used to augment, not replace the terms. Hence, a query for New York should be modified to search for new, york, and New York. Phrases used in this fashion are generally accepted to yield about a ten percent improvement in precision and recall over simple terms.

3.4.3 Complex Phrases

The quest to employ NLP to answer a user query . In fact, NLP systems were often seen as diametrically opposed to information retrieval systems because the NLP systems were trying to understand a document by building a canonical structure that represents the document. The goal behind the canonical structure is to reduce the inherent ambiguity found in language. A query that asks for information about walking should match documents that describe people who are moving slowly by gradually placing one foot in front of the other. A NLP system stores information about walking and moving slowly with the exact same canonical structure-it does this by first parsing the document syntactically-identifying the key elements of the document (subject, verb, object, etc.) and then building a single structure for the document. Simple primitives that encompass large categories of verbs were proposed such as PTRANS (physically transport), in which John drove to work and John used his car to get to work both result in the same simple structure John PTRANS work. Progress in NLP has occurred, but the reality is that many problems in knowledge representation make it extremely difficult to actually build the necessary canonical structures. The CYC project has spent the last fifteen years hand-building a knowledge base and has encountered substantial difficulty in identifying the exact means of representing the knowledge found in text. A side effect of full-scale NLP systems is that many tools that do not work perfectly for full language understanding are becoming quite usable for information retrieval systems. We may not be able to build a perfect knowledge representation of a document, but by using the same part-of-speech tagger and syntactic parser that might be used by an NLP system; we can develop several algorithms to identify key phrases in documents.

3.4.3.1 Use of POS and Word Sense Tagging

Part-of-speech taggers are based on either statistical or rule-based methods. The goal is to take a section of text and identify the parts of speech for each token. One approach incorporates a pretagged corpus to identify two measures: the frequency a given term is assigned a particular tag and the frequency with which different tag sequences occur . For example, duck might appear as a noun (creature that swims in ponds) eighty percent of a time and a verb (to get out of the way of a ball thrown at your head) twenty percent of the time. Additionally, "noun noun verb" may occur ten percent of the time while "noun noun noun" may occur thirty percent of the time. Using these two lists (generated based on a pretagged training corpus) a dynamic programming algorithm can be obtained to optimize the assignment of a tag to a token for a given step. Rule-based taggers in which tags are assigned based on the firing of sequences of rules are described. Part-of-speech taggers can be used to identify phrases. One use is to identify all sequences of nouns such as Virginia Beach or sequences of adjectives followed by nouns such as big red truck. Another use of a tagger is to modify processing such that a match of a term in the query only occurs if it matches the same part-of-speech found in the document. In this fashion, duck as a verb does not match a reference to duck as a noun. Although this seems sensible, it has not been shown to be particularly effective. One reason is that words such as bark have many different senses within a part of speech. In the sentences A dog's bark is often stronger than its bite and Here is a nice piece of tree bark, bark is a noun in both cases with very different word senses. Some initial development of word sense taggers exists. This work identifies word senses by using a dictionary based stemmer. Recent work on sense disambiguation for acronyms is found.

3.4.3.2 Syntactic Parsing

As we move along the continuum of increasingly more complex NLP tools, we now discuss syntactic parsing. These tools attempt to identify the key syntactic components of a sentence, such as subject, verb, object, etc. For simple sentences the problem is not so hard. Whales eat fish has the simple subject of Whales, the verb of eat, and the object of fish. Typically, parsers work by first invoking a part-of-speech tagger. Subsequently, a couple of different approaches are employed. One method is to apply a grammar. The first attempt at parsers used augmented transition networks (ATNs) that were essentially non-deterministic finite state automata in which: subject-verb-object would be a sequence of states. The problem is, that for complex sentences, many different paths occur through the automata.
Also, some sentences recursively start the whole finite state automata (FSA), in that they contain structures that have all the individual components of a sentence. Relative clauses that occur in sentences such as Mary, who is a nice girl that plays on the tennis team, likes seafood. Here, the main structure of Mary likes seafood also has a substructure of Mary plays tennis. After ATN s, rule-based approaches that attempt to parse based on firing rules, were attempted.

Other parsing algorithms, such as the Word Usage Parser (WUP) by Gomez, use a dictionary lookup for each word, and each word generates a specialized sequence of states .In other words, the ATN is dynamically generated based on individual word occurrences. Although this is much faster than an ATN, it requires substantial manual intervention to build the dictionary of word usages. Some parsers such as the Apple Pie Parser, are based on light parsing in which rules are followed to quickly scan for key elements of a sentence, but more complex sentences are not fully parsed. Once the parse is obtained, an information retrieval system makes use of the component structures. A simple use of a parser is to use the various component phrases such as SUBJECT or OBJECT as the only components of a query and match them against the document. Phrases generated in this fashion match many variations found in English. A query with American President will match phrases that include President of America, president who is in charge of America, etc. One effort that identified head-modifier pairs (e.g., "America+president") was evaluated against a patent collection and demonstrated as much as a sixteen percent improvement in average precision. On the TREC-5 dataset, separate indexes based on stems, simple phrases (essentially adjective-noun pairs or noun-noun pairs), head-modifier pairs, and people name's were all separately indexed. These streams were then combined and a twenty percent improvement in average precision was observed. To date, this work has not resulted in substantial improvements in effectiveness, although it dramatically increases the run-time performance of the system.

3.4.3.3 Information Extraction

The Message Understanding Conference (MUC) focuses on information extraction-the problem of finding various structured data within an unstructured document. Identification of people's names, places, amounts, etc. is the essential problem found in MUC, and numerous algorithms that attempt to solve this problem exist. Again, these are either rule-based or statistical algorithms. The first step in many of these algorithms is to generate a syntactic parse of the sentence, or at the very least, generate a part-of-speech tag. Details of these algorithms are found in the MUC Proceedings. More recently the Special Interest Group on Natural Language Learning of the Association for Computational Linguistics held a shared task on Language-Independent Named Entity Recognition. All of the proceedings may be found at. In this task, language independent Retrieval Utilities 145 algorithms were used to process standard test collections in English and German. Named entity taggers identify people names, organizations, and locations. We present a brief example that we created with a rule-based extractor from BBN Corporation to obtain this new document. This extractor works by using hundreds of hand-crafted rules that use surrounding terms to identify when a term should be extracted. First, we show the pre-extracted text-a paragraph about the guitarist Allen Collins.

<TEXT>
Collins began his rise to success as the lightning-fingered guitarist for the Jacksonville band by a group of high school students. The band enjoyed national fame in the 1970's with such hits as "Free Bird," "Gimme Three Steps," "Saturday Night Special" and Ronnie Van Zant's feisty "Sweet Home Alabama."

</TEXT>
The following output is generated by the extractor. Tags such as PERSON and LOCATION are now marked.
<TEXT>

<ENAMEX TYPE="PERSON">Collins<IENAMEX> began his rise to success as the lightning-fingered guitarist for the <ENAMEX TYPE="LOCATION">Jacksonville<IENAMEX> bandformed in <TIMEX TYPE="DATE"> 1 966<ITIMEX> by a group of high school students. The band enjoyed national fame in the <TIMEX TYPE="DATE">1970s <lTIMEX> with such hits as "Free <ENAMEX TYPE="PERSON"> Bird <IENAMEX>," "Gimme Three Steps," "Saturday Night Special" and <ENAMEX TYPE="PERSON">Ronnie Van Zant<IENAMEX>'s feisty "Sweet Home <ENAMEX TYPE="LOCATION">Alabama<IENAMEX>."

</TEXT>

In this example, and in many we have hand-checked, the extractor performs welL Many extractors are now performing at much higher levels of precision and recall than those of the. However, they are not perfect. Notice the label of PERSON being assigned to the term "Bird" in the phrase "Free Bird." Using extracted data makes it possible for a user to be shown a list of all person names, locations, and organizations that appear in the document collection.These could be used as suggested query terms for a user.The simplest use of an extractor is to recognize key phrases in the documents. An information retrieval system could incorporate extraction by increasing term weights for extracted terms. Given that extractors are only recently running fast enough to even consider using for large volumes of text,research in the area of using extractors for information retrieval is in its infancy.

3.5 Cross-Language Information Retrieval

Cross-Language Information Retrieval (CUR) is quickly becoming a mature area in the information retrieval world. The goal is to allow a user to issue a query in language L and have that query retrieve documents in language *L'*. The idea is that the user wants to issue a single query against a document collection that contains documents in a myriad of languages. An implicit assumption is that the user understands results obtained in multiple languages. If this is not the case, it is necessary for the retrieval system to translate the selected foreign language documents into a language that the user can understand.

3.5.1 Introduction

The key difference between CUR and monolingual information retrieval is that the query and the documents cannot be matched directly**.** In addition to the inherent difficulty in matching the inherent style, tone, word usage, and other features of the query with that of the document, we must now cross the language barrier between the query and the document. focuses on the core problems involved in crossing the language barrier..

3.5.1.1 Resources

Numerous resources are needed to implement cross-language retrieval systems. Most approaches use bilingual term lists, term dictionaries, a comparable corpus or a parallel corpus. A *comparable corpus* is a collection of documents in language L and another collection about the same topic in language *L'*. The key here is that the documents happen to have been written in different languages, but the documents are not literal translations of each other. A news article in language L by a newspaper in a country which speaks language L and an article in language *L'* by a newspaper in a country which speaks language *L'* is an example of comparable documents. The two newspapers wrote their own article; they with comparable corpora are that they must be *about the same topic.* A book in French on medicine and a book in Spanish on law are not comparable. If both books are about medicine or about law they are comparable. We will discuss CUR techniques using a comparable corpus. A *parallel corpus* provides documents in one language L that are then direct translations of language *L'* or vice versa. The key is that each document is in language L is a direct translation of a corresponding document in language *L'*. Hence, it is possible to align a parallel corpus at the document level, the paragraph level, the sentence level, the phrase level, or even the individual term level. Legislative documents in countries or organizations that are required to publish their proceedings in at least two languages are a common source of parallel corpora. In general, a parallel corpus will be most useful if it is used to implement cross-language retrieval of documents that are in a similar domain to the parallel corpus. Recent work shows that significant effectiveness can be obtained if the correct domain is selected. We discuss parallel corpus CUR techniques and also note that even within a single language such as Arabic, there are many different character sets. Language processing resources exist to not only detect a language but also to detect a character set. Cross-language systems often struggle with intricacies involved in working with different character sets within a single language. Unicode was developed to map the character representation for numerous scripts into a single character set, but not all electronic documents are currently stored in Unicode.

3.5.1.2 Evaluation

Different measures are used to evaluate the performance of cross-language information retrieval systems. The most obvious is simply to compute the average precision of the cross-language query.
Another approach is to compute the percentage of monolingual performance. This can occasionally be misleading because the techniques used to achieve a given monolingual performance may be quite different than those used for cross-language performance. Straightforward techniques typically result in 50% of monolingual performance, but the CUR literature contains results that exceed 100% because of the inherent query expansion that occurs when doing a translation. We note that queries with relevance judgments exist in Arabic, Chinese, Dutch, Finnish, French, German, Italian, Japanese, Korean, Swedish and Spanish. These have been used at various evaluations at To cross the language barrier, we must answer four core questions:

• What should be translated? Either the queries may be translated, the documents, or both queries and documents may be translated to some internal representation.

• Which tokens should be used to do a translation (e.g.; stems, words, phrases, etc.)?

• How should we use a translation? In other words, a single term in language L may map to several terms in Language Lt. We may use one of these terms, some of these terms, or all of these terms. Additionally, we might weight some terms higher than other terms if we have reason to believe that one translation is more likely than another.

• How can we remove spurious translations? Typically, there are spurious translations that can lead to poor retrieval. Techniques exist to remove these translations.As with monolingual retrieval, various strategies and utilities exist for cross language retrieval. As with the monolingual retrieval we organize the chapter according to strategies and utilities. Again, a strategy will take a query in language L and identify a measure of similarity between the query and documents in the target language L'. A utility enhances the work of any strategy.

UNIT-IV

EFFECIENCY

Retrieval strategies and utilities all focus on finding the relevant documents for a query. They are not concerned with how long it takes to find them. However, users of production systems clearly are concerned with run-time performance. A system that takes too long to find relevant documents is not as useful as one that finds relevant documents quickly. The bulk of information retrieval research has focused on improvements to precision and recall since the hope has been that machines would continue to speed up. Also, there is valid concern that there is little merit in speeding up a heuristic if it is not retrieving relevant documents. Sequential information retrieval algorithms are difficult to analyze in detail as their performance is often based on the selectivity of an information retrieval query. Most algorithms are on the order of *O (q(tfmax))* where *q* is the number of terms in the query and *t f max* is the maximum selectivity of any of the query terms. This is, in fact, a high estimate for query response time as many terms appear infrequently (about half are *hapax legomena,* or those that occur once We are not aware of a standard analytical model that effectively can be used to estimate query performance. Given this, sequential information retrieval algorithms are all measured empirically with experiments that require large volumes of data and are somewhat time consuming. The good news is that given larger and larger document collections, more work is appearing on improvements to run-time performance.

4.1 Inverted Index

Indexing requires additional overhead since the entire collection is scanned and substantial I/O is required to generate an efficiently represented inverted index for use in secondary storage. Indexing was shown to dramatically reduce the amount of I/O required to satisfy an ad hoc query. Upon receiving a query, the index is consulted, the corresponding posting lists are retrieved, and the algorithm ranks the documents based on the contents of the posting lists. The size of the index is another concern. Many indexes can be equal to the size of the original text. This means that storage requirements are doubled due to the index. However, compression of the index typically results in a space requirement of less than ten percent of the original text. The terms or phrases stored in the index depend on the parsing algorithms that are employed The size of posting lists in the inverted index can be approximated by the that the term frequency distribution in a natural language is such that if all terms were ordered and assigned a rank, the product of their frequency and their rank would be constant.

4.1.1 Building an Inverted Index

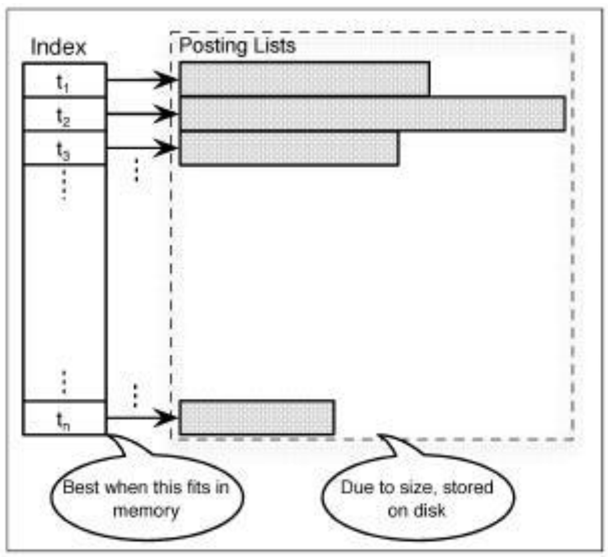
An inverted index consists of two components, a list of each distinct term referred to as the *index* and a set of lists referred to as *posting lists.* To compute relevance ranking, the term frequency or weight must be maintained. Thus, a posting list contains a set of tuples for each distinct term in the collection. The set of tuples is of the form <*decide, if*> for each distinct term in the collection. A typical uncompressed index spends four bytes on the document identifier and two bytes on the term frequency since a long document can have a term that appears more than 255 times. Consider a document collection in which document one contains two occurrences of *sales* and one occurrence of *vehicle.* Document two contains one occurrence of *vehicle.* The index would contain the entries *vehicle* and *sales*. The posting list is simply a linked list that is associated with each of these terms. For this example, we would have:

$$\textit{sales} \rightarrow (1, 2)$$
$$\textit{vehicle} \rightarrow (1, 1)\ (2, 1)$$

The entries in the posting lists are stored in ascending order by document number. Clearly, the construction of this inverted index is expensive, but once built, queries can be efficiently implemented. The algorithms underlying the implementation of the query processing and the construction of the inverted index are now described. A possible approach to index creation is as follows: An inverted index is constructed by stepping through the entire document collection, one term at a time. The output of the index construction algorithm is a set of files written todisk. These files are:

• **Index** file. Contains the actual posting list for each distinct term in the collection. A term, *t* that occurs in i different documents will have a posting list of the form:

Fig: Index file



where *di* indicates the document identifier of document i and *t lij* indicates the number of times term *j* occurs in document i.

• **Document** file. Contains information about each distinct document---document identifier, long document name, date published, etc .

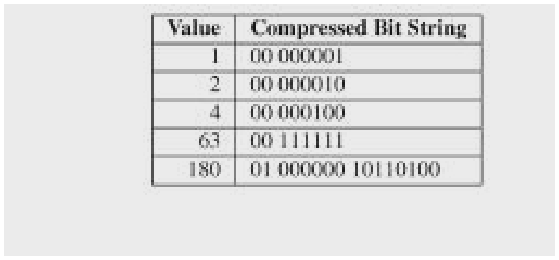
• **Weight** file. Contains the weight for each document. This is the denominator for the cosine coefficient-defined as the cosine of the angle between the query and document vector The construction of the inverted index is implemented by scanning the entire collection, one term at a time. When a term is encountered, a check is made to see if this term is a stop word (if stop word removal is used) or if it is a previously identified term. A hash function is used to quickly locate the term in an array. Collisions caused by the hash function are resolved via a linear linked list. Different hashing functions and their relative performance are given, Once the posting list corresponding tothis term is identified, the first entry of the list is checked to see if its document identifier matches the current document. If it does, the term frequency is merely incremented. Otherwise, this is the first occurrence of this term in the document, so a new posting list entry is added to the start of the list. The posting list is stored entirely in memory. Memory is allocated dynamically for each new posting list entry. With each memory allocation, a check is made to determine if the memory reserved for indexing has been exceeded. If it has, processing halts while all posting lists resident in memory are written to disk. Once processing continues, new posting lists are written. With each output to disk, posting list entries for the same term are chained together. Processing is completed when all of the terms are processed. At this point, the inverse document frequency for each term is computed by scanning the entire list of unique terms. Once the inverse document frequency is computed, it is possible to compute the document weight .This is done by scanning the entire posting list for each term.

4.1.2 Compressing an Inverted Index

Inverted index files is to develop algorithms that reduce I/O bandwidth and storage overhead. The size of the index file determines the storage overhead imposed. Furthermore, since large index files demand greater I/O bandwidth to read them, the size also directly affects the processing times. Although compression of text was extensively studied relatively little work was done in the area of inverted index compression, an index was generated that was relatively easy to decompress. It comprised less than ten percent of the original document collection, and, more impressively, *included stop terms.* Two primary areas in which an inverted index might be compressed are the term dictionary and the posting lists. Given relatively inexpensive memory costs, we do not focus on compression of indexes. The number of new terms always slightly increases as new domains are encountered, but it is reasonable to expect that it will stabilize at around one or two million terms. With an average term length of six, a four byte document frequency counter, and a four byte pointer to the first entry in the posting list, fourteen bytes are required for each term. For the conservative estimate of two million terms, the uncompressed index is likely to fit comfortably within 32 ME. Even if we are off by an order of magnitude, the amount of memory needed to store the index is conservatively under a gigabyte. Given the relatively small size of an index and the ease with which it should fit in memory, we do not describe a detailed discussion of techniques used to compress the index. We note that stemming reduces this requirement Also, the use of phrases improves precision and recall Storage of phrases in the index may well require compression. This depends upon how phrases are identified and restricted. Most systems eliminate phrases that occur infrequently. To introduce index compression algorithms, we first describe a relatively straightforward one that is referred to as the Byte Aligned (BA) index compression BA compression is done within byte boundaries to improve runtime at a slight cost to the compression ratio. This algorithm is easy to implement and provides good compression Variable length encoding although such efforts yield better compression, they do so at the expense of increased implementation complexity.

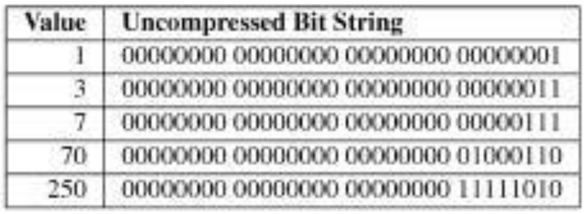
4.1.2.1 Fixed Length Index Compression

Posting list are in ascending order by document identifier. An exception to this document ordering occurs when a pruned inverted index approach is used. Hence, run-length encoding is applicable for document identifiers. For any document identifier, only the offset between the current identifier and the identifier immediately preceding it are computed. For the case in which no other document identifier exists, a compressed version of the document identifier is stored.Using this technique, a high proportion of relatively low numerical values is assured. This scheme effectively reduces the domain of the identifiers, allowing them to be stored in a more concise format. Subsequently, the following method is applied to compress the data. For a given input value, the two left-most bits are reserved to store a count for the number of bytes that are used in storing the value. There are four possible combinations of two bit representations; thus, a two bit length indicator is used for all document identifiers. Integers are stored in either 6, 14, 22, or 30 bits. Optimally, a reduction of each individual data record size by a factor of four is obtained by this method since, in the best case; all values are less than 26 = 64 and can be stored in a single byte. Without compression, four bytes are used for all document identifiers. For each value to be compressed, the minimum number of bytes required to store this value is computed indicates the range of values that can be stored, as well as the length indicator for one, two, three, and four bytes. For document collections exceeding 2^30 documents, this scheme can be extended to include a three bit length indicator which extends the range to 2^61-- 1. For term frequencies, there is no concept of using an offset between the successive values as each frequency is independent of the preceding value. However, the same encoding scheme can be used. Since we do not expect a document to contain a term more than 2^15 = 32, 768 times, either one or two bytes are used to store the value with one bit serving as the length indicator.



Value	Compressed Bit String
1	00 000001
2	00 000010
4	00 000100
63	00 111111
180	01 000000 10110100

Fig: Byte-Aligned Compression



Value	Uncompressed Bit String
1	00000000 00000000 00000000 00000001
3	00000000 00000000 00000000 00000011
7	00000000 00000000 00000000 00000111
70	00000000 00000000 00000000 01000110
250	00000000 00000000 00000000 11111010

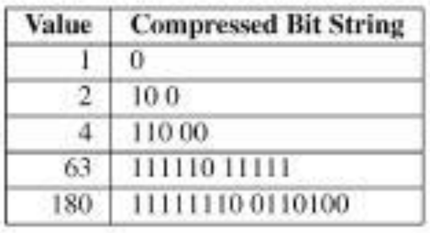
Fig: Baseline: No Compression

4.1.2.2 Example: Fixed Length Compression

Consider an entry for an arbitrary term, tl, which indicates that tl occurs in documents 1, 3, 7, 70, and 250. Byte-aligned (BA) compression uses the leading two high order bits to indicate the number of bytes required to represent the value. For the first four values, only one byte is required; for the final value, 180, two bytes are required. Note that only the differences between entries in the posting list must be computed. The difference of 250 - 70 = 180 is all that must be computed for this final value. The values and their corresponding compressed bit strings using no compression, the five entries in the posting list require four bytes each for a total of twenty bytes. The values and their corresponding compressed bit strings are shown in Table above. In this example, uncompressed data requires 160 bits, while BA compression requires only 48 bits.

4.1.3 Variable Length Index Compression

The differences in the posting list. They capitalize on the fact that for most long posting lists, the difference between two entries is relatively small.They first mention that patterns can be seen in these differences and that Huffman encoding provides the best compression. In this method, the frequency distribution of all of the offsets is obtained through an initial pass over the text, a compression scheme is developed based on the frequency distribution, and a second pass uses the new compression scheme. For example, if it was found that an offset of one has the highest frequency throughout the entire index, the scheme would use a single bit to represent the offset of one. This code represents an integer *x* with [*2llog2x*] + 1 bits. The first [*log2 x]* bits are the unary representation of [*log2x]*. (Unary representation is a base one representation of integers using only the digit one. The number 510 is represented as 111111.) After the leading unary representation, the next bit is a single stop bit of zero. At this point, the highest power of two that does not exceed *x* has been represented. The next [*log2x]* bits represent the remainder of *x* - 2 ^l[og2 x] in binary. As an example, consider the compression of the decimal 14. First, [*log2 x]* =3 is represented in unary as 111. Next, the stop bit is used. Subsequently, the remainder of *x* – 2^[lo g2 *X*] = 14 - 8 = 6 is stored in binary using [log2 14] = 3 bits as 110. Hence, the compressed code for 1410 is 1110110. Decompression requires only one pass, because it is known that for a number with *n* bits prior to the stop bit, there will be *n* bits after the stop bit. The first eight integers using the Elias, encoding.



Value	Compressed Bit String
1	0
2	10 0
4	110 00
63	111110 11111
180	1111110 0110100

Fig: Baseline: With Compression

4.1.3.1 Example: Variable Length Compression

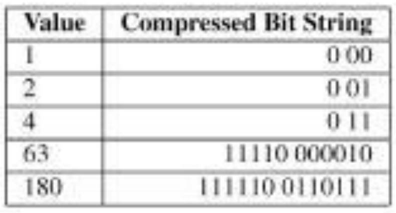
For our same example, the differences of 1, 2, 4, 63, and 180 are given in Table. This requires only 35 bits, thirteen less than the simple BA compression. Also, our example contained an even distribution of relatively large offsets to small ones. The real gain can be seen in that very small offsets require only a 1 or a O. Moffat and Zobel use the, code to compress the term frequency in a posting list, but use a more complex coding scheme for the posting list entries.

4.1.4 Varying Compression Based on Posting List Size

The *gamma* scheme can be generalized as a coding paradigm based on the vector V with positive integers i where 2: *Vi* 2: *N.* To code integer *x* 2: 1 relative to V, find *k* such that

In other words, find the first component of V such that the sum of all preceding components is greater than or equal to the value, *x,* to be encoded. For our example of 7, using a vector V of <1,2,4, 8, 16,32> we find the first three components that are needed (1, 2, 4) to equal or exceed 7, so *k* is equal to three. Now *k* can be encoded in some representation (unary is typically used) followed by the difference:

Using this sum we have: *d* = 7 - (1 + 2) - 1 = 3 which is now coded in[log2] *Vk* =[log2 4] = 2 binary bits. With this generalization, the, scheme



Value	Compressed Bit String
1	0 00
2	0 01
4	0 11
63	11110 000010
180	111110 0110111

Table: Variable Compression based on Posting List Size

can be seen as using the vector V composed of powers of 2 < 1, 2, 4, 8, ... , > and coding *k* in binary. Clearly, V can be changed to give different compression characteristics. Low values in *v* optimize compression for low numbers, while higher values in *v* provide more resilience for high numbers. A clever solution given by was to vary V for each posting list such that V = < *b, 2b, 4b, 8b, 16b, 32b, 64b, ...* , > where *b* is the median offset given in the posting list.

4.1.4.1 Example: Using the Posting List Size

Using our example of 1,2,4,63, 180, the median, *b,* has four results in the vector V = < 4, 8, 16,32,64, 128,256>. Table contains an example for the five posting lists using this scheme. This requires thirty-three bits as well and we can see that, for this example, the use of the median was not such a good choice as there was wide skew in the numbers. A more typical posting list in which numbers were uniformly closer to the median could result in better compression.

4.1.4.2 Throughput-optimized Compression

Index compression scheme that yields good compression ratios while maintaining fast decompression time for efficient query processing . They developed a variable-length encoding that takes advantage of the distribution of the document identifier offsets for each posting list. This is a hybrid of bit-aligned and byte-aligned compression; each 32-bit word contains encodings for a variable number of integers, but each integer within the word is encoded using an equal number of bits. Words are divided into bits used for a "selector" field and bits used for storing data. The selector field contains an index into a table of inter-word partitioning strategies based on the number of bits available for storing data, ensuring that each integer encoded in the word uses the same number of bits. The appropriate partitioning strategy is chosen based on the largest document identifier offset in the posting list. Anh and Moffat propose three variants based on this strategy, differing primarily in how the bits in a word are partitioned:

• Simple-9: Uses 28 bits for data and 4 bits for the selector field; the selection table has nine rows, as there are nine different ways to split 28 bits equally.

• Relative-W: Similar to Simple-9, but uses only two bits for the selector field, leaving 30 data bits with *10* partitions. The key difference is that, with only 2 selector bits, each word can only chose from 4 of the *10* available partitions - these are chosen relative to the selector value of the previous word. This algorithm obtains slight improvements over Simple-9.

• Carryover-12: This is a variant of Relative-W where some of the wasted space due to partitioning is reclaimed by using the leftover bits to store the selector value for the next word, allowing that word to use all of its bits for data storage. This obtains the best compression of the three, but it is the most complex, requiring more decompression time.

4.1.5 Index Pruning

To this point, we have discussed lossless approaches for inverted index compression. A lossy approach is called static index pruning. The basic idea was essentially, posting list entries may be removed or pruned without significantly degrading precision. Experiments were done with both term specific pruning and uniform pruning. With term specific pruning, different levels of pruning are done for each term. Static pruning simply eliminates posting list entries in a uniform fashion - regardless of the term. It was shown that pruning at levels of nearly seventy percent of the full inverted index did not significantly affect average precision.

4.1.6 Reordering Documents Prior to Indexing

Index compression efficiency can also be improved if we use an algorithm to reorder documents prior to compressing the inverted index Since the compression effectiveness ofmany encoding schemes is largely dependent upon the *gap* between document identifiers, the idea is that if we can feed documents to the algorithm correctly, we could reduce the average gap, thereby maximizing compression. Consider documents *d1, dgg,* and d1000, all which contain the same term *t.* For these documents we obtain a posting list entry for *t* of *t* --t *d* 1, *d51, d 101.* The document gap between each posting list entry is 50. If however, we arranged the documents prior to submitting them to the index, we could submit these documents as *dl, d2,* and *d3* which completely eliminate this gap. We note that for *D* documents there are 2D possible orderings, so any attempt to order documents will be faced with significant scalability concerns. The algorithms compare documents to other documents prior to submitting them for indexing

4.1.6.1 Top-Down

Generally, the two top-down algorithms consist of four main phases. In the first phase, called *center selection,* two groups of documents are selected from the collection and used as partitions in subsequent phases. In the *redistribution* phase, all remaining documents are divided among the selected centers according to their similarity. In the *recursion* phase, the previous phases are repeated recursively over the two resulting partitions until each one becomes a singleton. Finally, in the *merging* phase, the partitions formed from each recursive call are merged bottom-up, creating an ordering. The first of the two proposed top-down algorithms is called *transactional B & B*, as it is an implementation of the Blelloch and Blandford algorithm.This reordering algorithm obtains the best compression ratios of the four, however it is not scalable. The second top-down algorithm is called *Bisecting,* so named because its *center selection* phase consists of choosing two random documents as centers, thereby dramatically reducing the cost of this phase. Since its center selection is so simple, the *Bisecting* algorithm obtains less effective compression but it is more efficient.

4.1.6.2 Bottom-Up

The bottom-up algorithms begin by considering each document in the collection separately and they progressively group documents based on their similarity. The first bottom-up algorithms is inspired by the popular *k-means* approach to document clustering .The second uses *kscan;* an algorithm that is a simplified version of k-means which is based on a centroid-search algorithm. The *k-means* algorithm initially chooses *k* documents as cluster representatives, and assigns all remaining documents to those clusters based on a measure of similarity. At the end of the first pass, the cluster centroids are recomputed and the documents are reassigned according to their similarity to the new centroids. This iteration continues until the cluster centroids stabilize. The singlepass version of this algorithm only performs the first pass of this algorithm, and the authors select the *k* initial centers using the *Buckshot* clustering technique.The *k-scan* algorithm is a simplified version of single-pass k-means, requiring only *k* steps to complete. It forms clusters in place at each step, by first selecting a document to serve as the centroid for a cluster, and then assigning a portion of unassigned documents that have the highest similarity to that cluster.

4.2 Query Processing

The query performance can be improved by modifying the inverted index to support fast scanning of a posting list. Other work has shown that reasonable precision and recall can be obtained by retrieving fewer terms in the query. Computation can be reduced even further by eliminating some of the complexity found in the vector space model.

4.2.1 Inverted Index Modifications

Inverted index can be segmented to allow for a quick search of a posting list to locate a particular document. The typical ranking algorithm scans the entire posting list for each term in the query. An array of document scores is updated for each entry in the posting list.The least frequent terms should be processed first. The premise is that less frequent terms carry the most meaning and probably have the most significant contribution to high-ranking documents. The entire posting lists for these terms are processed. Some algorithms suggest that processing should stop after *d* documents are assigned a non-zero score. The premise is that at this point, the high-frequency terms in the query will simply be generating scores for documents that will not end up in the final top *t* documents, where *t* is the number of documents that are displayed to the user. A suggested improvement is to continue processing all the terms in the query, but only update the weights found in the *d* documents. In other words, after some threshold of *d* scores has been reached, the remaining query terms become part of an AND instead of the usual vector space OR. At this point, it is cheaper to reverse the order of the nested loop that is used to increment scores. Prior to reaching *d* scores, the basic algorithm is: For each term *t* in the query Q

```
Obtain the posting list entries for $t$
For each posting list entry that indicates $t$ is in doc
 i Update score for document i
```

For query terms with small posting lists, the inner loop is small; however, when terms that are very frequent are examined, extremely long posting lists are prevalent. Also, after *d* documents are accessed, there is no need to update the score for every document, it is only necessary to update the score for those documents that have a non-zero score. To avoid scanning very long posting lists, the algorithm is modified to be:

```
For each term $t$ in the query Q, Obtain posting list, $p$, for documents that contain $t$.
For each document $x$ in the reversed list of $d$ documents
Scan posting list $p$ for $x$ ,if $x$ exists update score for document $x$
```

The key here is that the inverted index must be changed to allow quick access to a posting list entry. It is assumed that the entries in the posting list are sorted by a document identifier. As a new document is encountered, its entry can be appended to the existing posting list. The posting list can quickly be scanned by checking the first partition pointer This check indicates whether or not a jump should be made to the next partition or if the current partition should be scanned. The process continues until the partition is found, and the document we are looking for is matched against the elements of the partition. A small size, *d*, of about 1,000 resulted in the best CPU time for a set of TREC queries against the TREC data.

4.2.2 Partial Result Set Retrieval

Another way to improve run-time performance is to stop processing after some threshold of computational resources is expended. One approach counts disk I/O operations and stops after a threshold of disk I/O operations is reached The key to this approach is to sort the terms in the query based on some indicator of term *goodness* and process the terms in this order. By doing this, query processing stops after the important terms have been processed. Sorting the terms is analogous to sorting their posting lists. Three measures used to characterize a posting list are now described.

4.2.2.1 Cutoff Based on Document Frequency

The simplest measure of term quality is to rely on document frequency. This was described in which showed that using between twenty-five to seventy-five percent of the query terms after they were sorted by document frequency resulted in almost no degradation in precision and recall for the TREC-4 document collection. In some cases, precision and recall improves with fewer terms because lower ranked terms are sometimes noise terms such as *good*, *nice*, *useful*, *etc*. These terms have long posting lists that result in scoring thousands of documents and do little to improve the quality of the result. Using term frequency is a means of implementing a dynamic stop word list in which high-frequency terms are eliminated without using a static set of stop words.

4.2.2.2 Cutoff Based on Maximum Estimated Weight

Two other measures of sorting the query terms. The first computes the maximum term frequency of a given query term tfmax and uses the following as a means of sorting the query.

$$\text{tfmax X idf}$$

The idea is that a term that appears frequently in all the documents, in which it appears, is probably of more importance than a term that appears infrequently in the documents that it appears in. The assumption is that the maximum value is a good indicator of how often the term appears in a document.

4.3 Signature Files

The use of signature files lies between a sequential scan of the original text and the construction of an inverted index. A signature is an encoding of a document. The idea is to encode all documents as relatively small signatures. Once this isdone, the signatures can be scanned instead of the entire documents. Typically, signatures do not uniquely represent a document, so it is usually necessary to implement retrieval in twophases. The first phase scans all of the signatures and identifies possible hits, and the second phase scans the original text of the documents in the possible hit list to ensure that they are correct matches. Hence, signature files are combined with pattern matching. Figure?? Illustrates the mapping of documents onto the signatures. Construction of a signature is often done with different hashing functions. One or more hashing functions are applied to each word in the document.

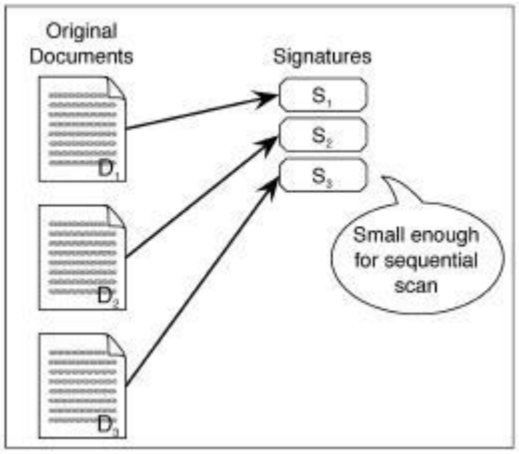
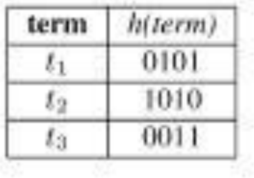


Fig: Information Retrieval.·Algorithms And Heuristics



term	*h(term)*
t_1	0101
t_2	1010
t_3	0011

Table: Building a Signature

The hashing function is used to set a bit in the signature. For example, if the terms *information* and *retrieval* were in a document and *h (information)* and *h(retrieval)* corresponded to bits one and four respectively, a four bit binary signature for this document would appear as 100l. A false match occurs when a word that is not in the list of *w* signatures has the same bitmap as one of these signatures. For example, consider a term iI that sets bits one and three in the signature and another term t2 that sets bits two and four in the signature. A third term t3 might correspond to bits one and two and thereby be deemed a *match* with the signature, even though it is not equal to tl or b Table gives the three terms just discussed and their corresponding hash values. Consider document dl that contains tl, document *d2* contains iI and t3 and document *d3* contains tl and *t2*. Table has the signatures for these three documents.

Hence, a query that is searching for term *t3* will obtain a false match on document *d3* even though it does not contain *t3*. By lengthening the signature to 1,024 bits and keeping the number of words stored in a signature small, the chance of a false match can be shown to be less than three percent. To implement document retrieval, a signature is constructed for the query. A Boolean AND is executed between the query signature and each document signature. If the AND returns TRUE, the document is added to the possible hit list. Similarly, a Boolean OR can be executed if it is only necessary for any word in the query to be in the document. To minimize false positives, multiple hashing functions are applied to the same word. A Boolean signature cannot store proximity information or information about the weight of a term as it appears in a document. Most measures of relevance determine that a document that contains a query term multiple times will be ranked higher than a document that contains the same term only once. With Boolean signatures, it is not possible to represent the number of times a term appears in a document; therefore, these measures of relevance cannot be implemented. Signatures are useful if they can fit into memory. Also, it is easier to add or delete documents in a signature file than to an inverted index, and the order of an entry in the signature file does not matter. This somewhat orderless processing is amenable to parallel processing. However, there is always a need to check for false matches, and the basic definition does not support ranked queries. A modification to allow support for document ranking is to partition a signature into groups where each term frequency is associated with a group.

4.3.1 Scanning to Remove False Positives

Once a signature has found a match, scanning algorithms are employed to verify whether or not the match is a false positive due to collisions. We do not cover these in detail as a lengthy survey surrounding the implementation of many text scanning algorithms. Signature algorithms can be employed without scanning for false drops and no significant degradation in precision and recall occurs . However, for completeness, we do provide a brief summary of text scanning algorithms. Pattern matching algorithms are related to the use of scanning in information retrieval since they strive to find a pattern in a string of text characters. Typically, pattern matching is defined as finding all positions in the input text that contain the start of a given pattern. If the pattern is of size *p* and the text is of size s, the naive nested loop pattern match requires *O (ps)* comparisons. to identify, and Pratt (KMP) also describe an algorithm that runs in O(s) time that scans forward along the text, but uses preprocessing of the pattern to determine appropriate skips in the string that can be safely taken The Boyer-Moore algorithm is another approach that preprocesses the pattern, but starts at the last character of the pattern and works backwards towards the beginning of the string. Two preprocessed functions of the pattern are developed to skip parts of the pattern when repetition in the pattern occurs and to skip text that simply cannot match the pattern. These functions use knowledge gleaned from the present search point.If a match in the hash function occurs (i.e., a collision between *h(pattern)* and *h(text)),* the contents of the pattern and text are examined. The goal is to reduce false collisions. By using large prime numbers, collisions occur extremely rarely, if at all. Another pattern matching algorithm is presented .This algorithm uses a set of bit strings which represent Boolean states that are constantly updated as the pattern is streamed through the text. The best of these algorithms runs in a linear time *as* where *a* is some constant *a* :::; 1.0 and s is the size of the string. The goal is to lower the constant. In the worst case, s comparisons must be done, but the average case for these algorithms is often sublinear. An effort is made in these algorithms to avoid having to look backward in the text. The scan continues to move forward with each comparison to facilitate a physically contiguous scan of a disk. The KMP algorithm builds finite state automata for many patterns so it is directly applicable. An algorithm by Uratani and Takeda combines the FSA approach by Aho and Corasick with the Boyer-Moore idea of avoiding much of the search space. Essentially, the FSA is built by using some of the search space reductions given by Boyer-Moore. The FSA scans text from right to left, as done in Boyer-Moore. Note this is done for a query that contains multiple terms]. **In** a direct comparison with repeated use of the Boyer-Moore algorithm, the Uratani and Takeda algorithm is shown to execute ten times fewer probes for a query of 100 patterns.

4.4 Duplicate Document Detection

A method to improve both efficiency and effectiveness of an information retrieval system is to remove duplicates or near duplicates. Duplicates can be removed either at the time documents are added to an inverted index or upon retrieving the results of a query. The difficulty is that we do not simply wish to remove exact duplicates; we may well be interested in removing *near* duplicates as well. However, we do not wish our threshold for *nearness* to be so broad that documents are deemed duplicate when, in fact, they are sufficiently different that the user would have preferred to see each of them as individual documents. For Web search, the duplicate document problem is particularly acute. A search for the term *apache* might yield numerous copies of Web pages about, the Web server product and numerous duplicates about the Indian tribe. The user should only be shown two hyperlinks, but instead is shown thousands. Additionally, these redundant pages can affect term and document weighting schemes. Additionally, they can increase indexing time and reduce search efficiency.

4.4.1 Finding Exact Duplicates

For each document. Each document is then examined for duplication by looking up the value (hash) in either an in-memory hash or persistent lookup system. Several common hash functions used are. These functions are used because they have three desirable properties, namely: they can be calculated on arbitrary document lengths, they are easy to compute, and they have very low probabilities of collisions. While this approach is both fast and easy to implement, the problem is that it will find only *exact* duplicates. The slightest change (e.g.; one extra white space) results in two similar documents being deemed unique. For example, a Web page that displays the number of visitors along with the content of the page will continually produce different signatures even though the document is the same. The only difference is the counter, and it will be enough to generate a different hash value for each document.

4.4.2 Finding Similar Duplicates

While it is not possible to define precisely at which point a document is no longer a duplicate of another, researchers have examined several metrics for determining the similarity of a document to another. The first is *resemblance*. This work suggests that if a document contains roughly the same semantic content, it is a duplicate whether or not it is a precise syntactic match.

The resemblance *r* of document *Di* and document *Dj* , as defined in Equation is the intersection of features over the union of features from two documents. This metric can be used to calculate a fuzzy similarity between two documents. For example, assume *Di* and *Dj* share fifty percent of their terms and each document has 10 terms. Their resemblance would be 155 = 0.33. The first is what features and threshold t should be used. The second is efficiency issues that come into play with large collections and the optimizations that can be applied identify the similarity between two documents. For duplicate detection a binary feature representation produces a similarity of two documents similar to term-based resemblance. Thus, as the distance of two documents approaches 1.0, they become more similar in relation to the features being compared.

4.4.2.1 Shingles

The comparison of document subsets allows the algorithms to calculate a percentage of overlap between two documents using resemblance as given in This relies on hash values for each document subsection/feature set and filters those hash values to reduce the number of comparisons the algorithm must perform. This filtration of features, therefore, improves the runtime performance. Note that the simplest filter is strictly a syntactic filter based on simple syntactic rules, and the trivial subset is the entire collection. In the shingling approaches, rather than comparing documents, subdocuments are compared, and thus, each document can produce many potential duplicates. Returning many potential matches requires vast user involvement to sort out potential duplicates, diluting the potential usefulness of these types of approaches. To combat the inherent efficiency issues, several optimization techniques were proposed to reduce the number of comparisons made. The DSC algorithm reduces the number of shingles used. The DSC algorithm has a more efficient alternative, DSC-SS, which uses super shingles. This algorithm takes several shingles and combines them into a super shingle. The result is a document with a few super shingles instead of many shingles. Resemblance is defined as matching a single super shingle in two documents. This is much more efficient because it no longer requires a full counting of all overlaps.

4.4.2.2 **Duplicate Detection** via **Similarity**

I-Match uses a hashing scheme that uses only *some* terms in a document. The decision of which terms to use is key to the success of the algorithm. IMatch is a hash of the document that uses collection statistics, for example, *idf,* to identify which terms should be used as the basis for comparison. The use of collection statistics allows one to determine the usefulness of terms for duplicate document detection. Previously, it was shown that terms with high collection frequencies often do not add to the semantic content of the document I-Match assumes that that removal of very infrequent terms or very common terms results in good document representations for identifying duplicates.

Pseudo-code for the algorithm is as follows.

• Get document

• Parse document into a token steam, removing format tags.

• Using term thresholds (idt), retain only significant tokens.

• Insert relevant tokens into unicode ascending ordered tree of unique tokens.

• The tuple is inserted into the storage data structure using the key.

• If there is a collision of digest values then the documents are similar.

The I-Match time complexity is comparable to the DSC-SS algorithm, which generates a single super shingle if the super shingle size is large enough to encompass the whole document. Otherwise, it generates *k* super shingles while I-Match only generates one. Since *k* is a constant in the DSC-SS analysis, the two algorithms are equivalent. I-Match, is more efficient in practice. However, the real benefit of I-Match over DSC-SS is that small documents are not ignored. With DSC-SS, it is quite likely that for sufficiently small documents, no shingles are identified for duplicate detection. Hence, those short documents are not considered even though they may be duplicated. While I-Match is efficient it suffers from the same brittleness that the original hashing techniques suffered from, when some slight variation in that hash is made. One recent enhancement to I-Match has been the use of random lexicon variations of the feature *idf* range. These variations are then used to produce multiple signatures of a document. All of the hashes can be considered a valid signature, this modification to I-Match reduces the brittleness of I-Match.

UNIT-V
Integrated Structured Data and Text

5.1 A Historical Progression

Previous work can be partitioned into systems that combine information retrieval and DBMS together, or systems that extend relational DBMS to include information retrieval functionality.

5.1.1 Combining Separate Systems

Integrated solutions consist of writing a central layer of software to send requests to underlying DBMS and information retrieval systems. Queries are parsed and the structured portions are submitted as a query to the DBMS, while text search portions of the query are submitted to an information retrieval system. The results are combined and presented to the user. The key advantage of this approach is that the DBMS and information retrieval system are commercial products that are continuously improved. Additionally, software development costs are minimized. The disadvantages include poor data integrity, portability, and run-time performance.

5.1.1.1 Data Integrity

Data integrity is sacrificed because the DBMS transaction log and the information retrieval transaction log are not coordinated. If a failure occurs in the middle of an update transaction, the DBMS will end in a state where the entire transaction is either completed or it is entirely undone. It is not possible to complete half of an update. The information retrieval log (if present) would not know about the DBMS log. Hence, the umbrella application that coordinates work between the two systems must handle all recovery. Recovery done within an application is typically error prone and, in many cases, applications simply ignore this coding. Hence, if a failure should occur in the information retrieval system, the DBMS will not know about it. An update that must take place in both systems can succeed in the DBMS, but fail in the information retrieval system. A partial update is clearly possible, but is logically flawed.

5.1.1.2 Portability

Portability is sacrificed because the query language is not standard. Presently, a standard information retrieval query language does not exist. However, some work is being done to develop standard information retrieval query languages. If one existed, it would require many years for widespread commercial acceptance to occur. The problem is that developers must be retrained each time a new DBMS and information retrieval system is brought in. Additionally, system administration is far more difficult with multiple systems.

5.1.1.3 Performance

Run-time performance suffers because of the lack of parallel processing and query optimization. Although most commercial DBMS have parallel implementations, most information retrieval systems do not. Query optimization exists in every relational DBMS. The optimizer's goal is to choose the appropriate access path to the data. A rule-based optimizer uses pre-defined rules, while a cost-based optimizer estimates the cost of using different access paths and chooses the cheapest one. In either case, no rules exist for the unstructured portion of the query and no cost estimates could be obtained because the optimizer would be unaware of the access paths that may be chosen by the information retrieval system. Thus, any optimization that included both structured and unstructured data would have to be done by the umbrella application. This would be a complex process. Hence, run-time performance would suffer due to a lack of parallel algorithms and limited global query optimization.

5.1.1.4 Extensions to SQL

Proposals have been made that SQL could be modified to support text. The SMART information retrieval prototype developed in the 1980's used the INGRES relational database system to store its data. In this relevance ranking as well as Boolean searches were supported. The focus was on the problem of efficiently searching text in a RDBMS. RDBMS would store the inverted index in another table thereby making it possible to easily view the contents of the index. An information retrieval system typically hides the inverted index as simply an access structure that is used to obtain data. By storing the index as a relation, the authors pointed out that users could easily view the contents of the index and make changes if necessary. Extensions such as RELEVANCE (*) were mentioned, that would compute the relevance of a document to a query using some pre-defined relevance function. More recently, a language called SQLX was used to access documents in a multimedia database. SQLX assumes that an initial cluster based search has been performed based on keywords. SQLX extensions allow for a search of the results with special connector attributes that obviate the need to explicitly specify joins.

5.1.2 User-defined Operators

User-defined operators that allow users to modify SQL by adding their own functions to the DBMS engine was described. Commercialization of this idea has given rise to several products including the Teradata Multimedia Object Manager, Oracle Cartridges, ruM DB2 Text Extender, as well as features in Microsoft SQL Server. An example query that uses the user-defined area function is given below. Area must be defined as a function that accepts a single argument. The datatype of the argument is given as rectangle. Hence, this example uses both a user-defined function and a user-defined datatype.

Ex: 1 SELECT MAX (AREA (Rectangle)) FROM SHAPE

In the information retrieval domain, an operator such as proximity () could be defined to compute the result set for a proximity search. In this fashion the "spartan simplicity of SQL" is preserved, but users may add whatever functionality is needed. A few years later user-defined operators were defined to implement information retrieval.

The following query obtains all documents that contain the terms termI, term2,and term3:

```
Ex: 2 SELECT DocJd
FROM DOC
WHERE SEARCH-TERM (Text, Terml, Term2,Term3)
```

This query can take advantage of an inverted index to rapidly identify the terms. To do this, the optimizer would need to be made aware of the new access method. Hence, user-defined functions also may require user-defined access methods.

The following query uses the proximity function to ensure that the three query terms are found within a window of five terms.

```
Ex: 3 SELECT DocJd
FROM DOC
WHERE PROXIMITY (Text, 5, Terml, Term2, Term3)
```

The advantages of user-defined operators are that they not only solve the problem for text, but also solve it for spatial data, image processing, etc. Users may add whatever functionality is required. The key problems with user-defined operators again are integrity, portability, and run-time performance.

5.1.2.1 Integrity

User-defined operators allow application developers to add functionality to the DBMS rather than the application that uses the DBMS. This unfortunately opens the door for application developers to circumvent the integrity of the DBMS. For user-defined operators to be efficient, they must be linked into the same module as the entire DBMS, giving them access to the entire address space of the DBMS. Data that reside in memory or on disk files that are currently opened can be accessed by the user-defined operator. It is possible that the user-defined operator could corrupt these data.

To protect the DBMS from a faulty user-defined operator, a remote procedure call (RPC) can be used to invoke the user-defined operator. This ensures the operator has access only to its address space, not the entire DBMS address space. Unfortunately, the RPC incurs substantial overhead, so this is not a solution for applications that require high performance.

5.1.2.2 Portability

A user-defined operator implemented at SITE A may not be present at SITE B. Worse, the operator may appear to exist, but it may perform an entirely different function. Without user-defined operators, anyone with an RDBMS may write an application and expect it to run at any site that runs that RDBMS. With user-defined operators, this perspective changes as the application is limited to only those sites with the user-defined operator.

5.1.2.3 Performance

Query optimization, by default, does not know much about the specific user defined operators. Optimization is often based on substantial information about the query. A query with an EQUAL operator can be expected to retrieve fewer rows than a LESS THAN operator. This knowledge assists the optimizer in choosing an access path. knowing the semantics of a user-defined operator, the optimizer is unable to efficiently use it. Some user-defined operators might require a completely different access structure like an inverted index. Unless the optimizer knows that an inverted index is present and should be included in path selection, this path is not chosen.

For user-defined operators to gain widespread acceptance, some means of providing information about them to the optimizer is needed. Additionally, parallel processing of a user-defined operator would be something that must be defined inside of the user-defined operator. The remainder of the DBMS would have no knowledge of the user-defined operator, and as such, would not know how to parallelize the operator.

5.1.3 Non-first Normal Form Approaches

Non-first normal form (NFN) approaches have been proposed. The idea is that many-many relationships are stored in a cumbersome fashion when 3NF (third normal form) is used. Typically, two relations are used to store the entities that share the relationship, and a separate relation is used to store the relationship between the two entities.

For an inverted index, a many-many relationship exists between documents and terms. One term may appear in many documents, while one document may have many terms. This, as will be shown later, may be modelled with a DOC relation to store data about documents, a TERM relation to store data about individual terms, and an INDEX relation to track an occurrence of a term in a document.

Instead of three relations, a single NFN relation could store information about a document, and a nested relation would indicate which terms appeared in that document.

Although this is clearly advantageous from a run-time performance standpoint, portability is a key issue. No standards currently exist for NFN collections. Additionally, NFN makes it more difficult to implement ad hoc queries. Since both user-defined operators and NFN approaches have deficiencies, we describe an approach using the unchanged, standard relational model to implement a variety of information retrieval functionality. This support integrity and portability while still yielding acceptable runtime performance. Some applications, such as image processing or CAD/CAM may require user-defined operators, as their processing is fundamentally not set-oriented and is difficult to implement with standard SQL.

5.1.4 Bibliographic Search with Unchanged SQL

The potential of relational systems to provide typical information retrieval functionality included queries using structured data (e.g., affiliation of an author) with unstructured data (e.g., text found in the title of a document). The following relations model the document collection.

- DIRECTORY (name, institution)-identifies the author's name and the institution the author is affiliated with.
- AUTHOR (name, Doc/d)-indicates who wrote a particular document.
- INDEX(term, Doc/d)-identifies terms used to index a particular document

The following query ranks institutions based on the number of publications that contain input_term in the document.

```
Ex: 4 SELECT UNIQUE institution, COUNT (UNIQUE name)
          FROM DIRECTORY   WHERE name IN
                (SELECT name FROM AUTHOR WHERE DocId IN
                     SELECT DocId FROM INDEX WHERE term =
                     inputterm
                          ORDER BY 2 DESCENDING)
```

There are several benefits for using the relational model as a foundation for document retrieval. These benefits are the basis for providing typical information retrieval functionality in the relational model, so we will list some of them here.

- Recovery routines
- Performance measurement facilities
- Database reorganization routines
- Data migration routines
- Concurrency control
- Elaborate authorization mechanisms
- Logical and physical data independence
- Data compression and encoding routines
- Automatic enforcement of integrity constraints
- Flexible definition of transaction boundaries (e.g., commit and rollback)
- Ability to embed the query language in a sequential applications language

5.3 Information Retrieval as a Relational Application

Initial extensions are based on the use of a QUERY(term) relation that stores the terms in the query, and an INDEX (DocId, term)relation that indicates which terms appear in which documents. The following query lists all the identifiers of documents that contain at least one term in QUERY:

```
Ex: 5 SELECT DISTINCT (i.DocId) FROM INDEX i, QUERY q WHERE i.term = q.term
```

Frequently used terms or stop terms are typically eliminated from the document collection. Therefore, a STOP _TERM relation may be used to store the frequently used terms. The STOP TERM relation contains a single attribute (term). A query to identify documents that contain any of the terms in the query except those in the STOP _TERM relation is given below:

```
Ex: 6 SELECT DISTINCT (i.DocId) FROM INDEX i, QUERY q, STOP TERM
           s WHERE i.term = q.term AND i.term i= s.term
```

Finally, to implement a logical AND of the terms InputTerml, InputTerm2 and InputTerm3,

```
Ex: 7 SELECT DocId FROM INDEX WHERE term = InputTermI INTERSECT
         SELECT DocId FROM INDEX WHERE term = InputTerm2 INTERSECT
            SELECT DocId FROM INDEX WHERE term = InputTerm3
```

The query consists of three components. Each component results in a set of documents that contain a single term in the query. The INTERSECT keyword is used to find the intersection of the three sets. After processing, an AND is implemented.

The key extension for relevance ranking was a corr() function-a built-in function to determine the similarity of a document to a query.

The SEQUEL (a precursor to SQL) example that was given was:

```
Ex: 8 SELECT DocId FROM INDEX i, QUERY q WHERE i.term =
           q.term GROUP BY DocId HAVING CORRO > 60
```

Other extensions, such as the ability to obtain the first n tuples in the answer set, were given. We now describe the unchanged relational model to implement information retrieval functionality with standard SQL. First, a discussion of preprocessing text into files for loading into a relational DBMS is required.

5.3.1 Preprocessing

Input text is originally stored in source files either at remote sites or locally on CD-ROM. For purposes of this discussion, it is assumed that the data files are in ASCII or can be easily converted to ASCII with SGML markers. SGML markers are a standard means by which different portions of the document are marked. The markers in the working example are found in the TIPSTER collection which was in previous years as the standard dataset for TREC. These markers begin with a < and end with a > (e.g., <TAG).A preprocessor that reads the input file and outputs separate flat files is used. Each term is read and checked against a list of SGML markers. The main algorithm for the preprocessor simply parses terms and then applies a hash function to hash them into a small hash table. If the term has not occurred for this document, a new entry is added to the hash table. Collisions are handled by a single linked list associated with the hash table. If the term already exists, its term frequency is updated. When an end-of-document marker is encountered, the hash table is scanned. For each entry in the hash table a record is generated. The record contains the document identifier for the current document, the

term, and its term frequency. Once the hash table is output, the contents are set to NULL and the process repeats for the next document.

After processing, two output files are stored on disk. The output files are then bulk-loaded into a relational database. Each file corresponds to a relation. The first relation, DOC, contains information about each document. The second relation, INDEX, models the inverted index and indicates which term appears in which document and how often the term has appeared.

The relations are:

INDEX (DocId, Term, TermFrequency)

DOC (DocId, DocName, PubDate, Dateline)

These two relations are built by the preprocessor. A third TERM relation tracks statistics for each term based on its number of occurrences across the document collection. At a minimum, this relation contains the document frequency (df) and the inverse document frequency (idf). The term relation is of the form: TERM(Term, Idf). It is possible to use an application programming interface (API) so that the preprocessor stores data directly into the database. However, for some applications, the INDEX relation has one hundred million tuples or more. This requires one hundred million separate calls to the DBMS INSERT function. With each insert, a transaction log is updated. All relational DBMS provide some type of bulk-load facility in which a large flat file may be quickly migrated to a relation without significant overhead. Logging is often turned off (something not typically possible via an on-line API) and most vendors provide efficient load implementations. For parallel implementations, flat files are loaded using multiple processors. This is much faster than anything that can be done with the API.

For all examples in this chapter, assume the relations were initially populated via an execution of the preprocessor, followed by a bulk load. Notice that the DOC and INDEX tables are output by the preprocessor. The TERM relation is not output. In the initial testing of the preprocessor, it was found that this table was easier to build using the DBMS than within the preprocessor. To compute the TERM relation once the INDEX relation is created, the following SQL statement is used:

Ex: 9
```
INSERT INTO TERM SELECT Term, log(N / COUNT(*)) FROM INDEX GROUP BY
    Term
```

N is the total number of documents in the collection, and it is usually known prior to executing this query. However, if it is not known then SELECT COUNT (*) FROM DOC will obtain this value. This statement partitions the INDEX relation by each term, and COUNT (*) obtains the number of documents represented in each partition (i.e., the document frequency). The idf is computed by dividing N by the document frequency.

Consider the following working example. Input text is provided, and the preprocessor creates two files which are then loaded into the relational DBMS to form DOC and INDEX. Subsequently, SQL is used to populate the TERM relation.

5.3.2 A Working Example

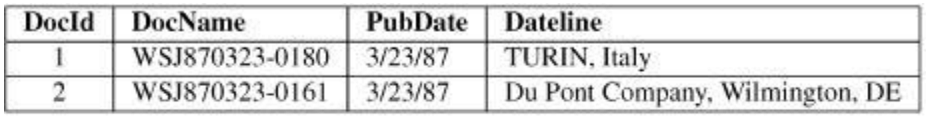
Two documents are taken from the TIPSTER collection and modelled using relations. The documents contain both structured and unstructured data and are given below.

```
<DOC>
<DOCNO> WSJ870323-0180 </DOCNO>
<HL> Italy's Commercial Vehicle Sales </HL>
<DD> 03/23/87 </DD>
<DATELINE> TURIN, Italy </DATELINE>
<TEXT>
Commercial-vehicle sales in Italy rose 11.4% in February from a year earlier, to 8,848 units,
according to provisional figures from the Italian Association of Auto Makers.
</TEXT>
</DOC>
<DOC>
<DOCNO> WSJ870323-0161 </DOCNO>
<HL> Who's News: Du Pont Co. </HL>
<DD> 03/23/87 </DD>
<DATELINE> Du Pont Company, Wilmington, DE
</DATELINE> <TEXT>
John A. Krol was named group vice president, Agriculture Products department, of this
diversified chemicals company, succeeding Dale E. Wolf, who will retire May 1. Mr. Krol was
formerly vice president in the Agricultural Products department. </TEXT>

</DOC>
```

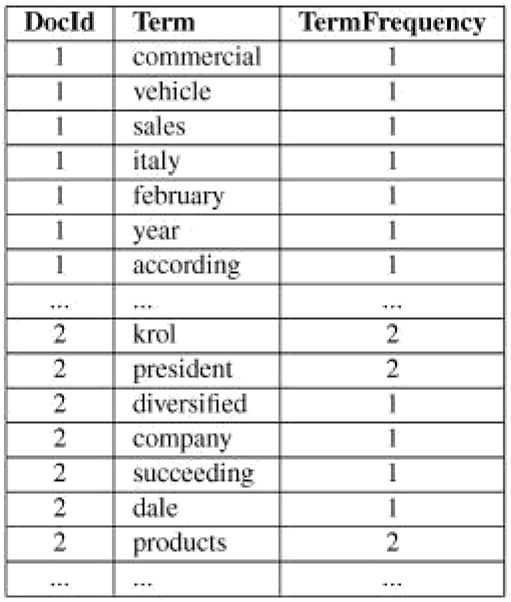
The preprocessor accepts these two documents as input and creates the two files that are then loaded into the relational DBMS. The corresponding DOC and INDEX relations are given in following Tables.

Table: DOC



DocId	DocName	PubDate	Dateline
1	WSJ870323-0180	3/23/87	TURIN, Italy
2	WSJ870323-0161	3/23/87	Du Pont Company, Wilmington, DE

Table: INDEX



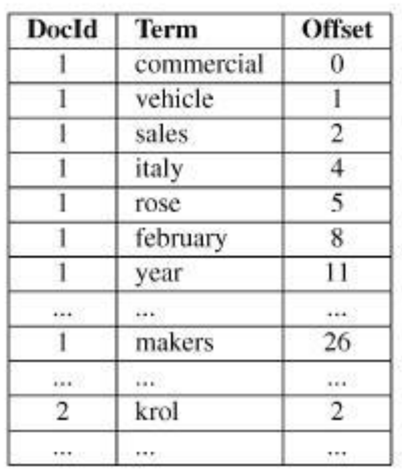
DocId	Term	TermFrequency
1	commercial	1
1	vehicle	1
1	sales	1
1	italy	1
1	february	1
1	year	1
1	according	1
...	...	...
2	krol	2
2	president	2
2	diversified	1
2	company	1
2	succeeding	1
2	dale	1
2	products	2
...	...	...

INDEX models an inverted index by storing the occurrences of a term in a document. Without this relation, it is not possible to obtain high performance text search within the relational model. Simply storing the entire document in a Binary Large OBject (BLOB) removes the storage problem, but most searching operations on BLOB's are limited, in that BLOB's typically cannot be indexed. Hence, any search of a BLOB involves a linear scan, which is significantly slower than the O (log n) nature of an inverted index. In a typical information retrieval system, a lengthy preprocessing phase occurs in which parsing is done and all stored terms are identified. A posting list that indicates, for each term, which documents contain that term is identified. A pointer from the term to the posting list is implemented. In this fashion, a hashing function can be used to quickly jump to the term, and the pointer can be followed to the posting list.

The fact that one term can appear in many documents and one document contains many terms indicates that a many-many relationship exists between terms and documents. To model this, document and term may be thought of as entities (analogous to employee and project), and a linking relation that describes the relationship EMP_PROJ must be modeled. The INDEX relation described below models the relationship. A tuple in the INDEX relation is equivalent to an assertion that a given term appears in a given document. Note that the term frequency (tf) or number of occurrences of a term within a document is a specific characteristic of the APPEARS-IN relationship; thus, it is stored in this table. The primary key for this relation is (DocId,Term), hence, term frequency is entirely dependent upon this key. For proximity searches such as "Find all documents in which the phrase vice president exists," an additional offset attribute is required. Without this, the INDEX relation indicates that vice and president co-occur in the same document, but no information as to their location is given. To indicate that vice is adjacent to president, the offset attribute identifies the current term offset in the document. The first term is given an offset of zero, the second an offset of one, and, in general, the n^{th} is given an offset of n - 1. The INDEX_PROX relation given in Table contains the necessary offset attribute required to implement proximity searches.

Several observations about the INDEX_PROX relation should be noted. Since stop words are not included, offsets are not contiguously numbered. An offset is required for each occurrence of a term. Thus, terms are listed multiple times instead of only once, as was the case in the original INDEX relation.

Table: INDEX_PROX



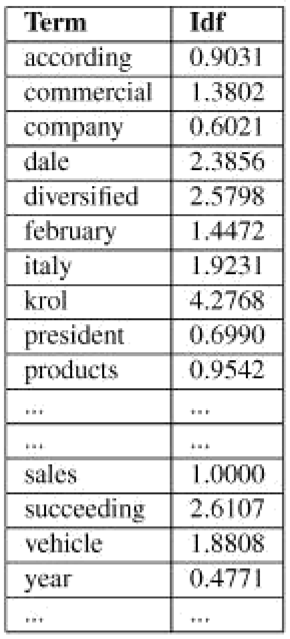
DocId	Term	Offset
1	commercial	0
1	vehicle	1
1	sales	2
1	italy	4
1	rose	5
1	february	8
1	year	11
...	...	...
1	makers	26
...	...	...
2	krol	2
...	...	...

To obtain the INDEX relation from INDEX...PROX, the following statement can be used:

Ex: 10 INSERT INTO INDEX SELECT DocId, Term, COUNT (*) FROM
INDEX..PROX GROUP BY DocId, Term

Finally, single-valued information about terms is required. The TERM relation contains the idf for a given term. To review, a term that occurs frequently has a low idf and is assumed to be relatively unimportant. A term that occurs infrequently is assumed very important. Since each term has only one idf, this is a single-valued relationship which is stored in a collection-wide single TERM relation.

Table: Term

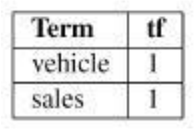


Term	Idf
according	0.9031
commercial	1.3802
company	0.6021
dale	2.3856
diversified	2.5798
february	1.4472
italy	1.9231
krol	4.2768
president	0.6990
products	0.9542
...	...
...	...
sales	1.0000
succeeding	2.6107
vehicle	1.8808
year	0.4771
...	...

maintain a syntactically fixed set of SQL queries for information retrieval processing, and to reduce the syntactic complexities of the queries themselves, a QUERY relation is used. The QUERY relation contains a single tuple for each query term. Queries are simplified because the QUERY relation can be joined to INDEX to see if any of the terms in QUERY are found in INDEX. Without QUERY, a lengthy WHERE clause is required to specifically request each term in the query.

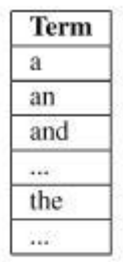
Finally, STOP_TERM is used to indicate all of the terms that are omitted during the parsing phase. This relation illustrates that the relational model can store internal structures that are used during data definition and population.

Table: query



Term	**tf**
vehicle	1
sales	1

Table: STOP_TERM



Term
a
an
and
...
the
...

The following query illustrates the potential of this approach. The SQL satisfies the request to "Find all documents that describe vehicles and sales written on 3/23/87." The keyword search covers unstructured data, while the publication date is an element of structured data. This example is given to quickly show how to integrate both structured data and text. Most information retrieval systems support this kind of search by making DATE a "zoned field"-a portion of text that is marked and always occurs in a particular section or zone of a document. These fields can then be parsed and stored in a relational structure. Example illustrates a sequence of queries that use much more complicated unstructured data, which could not easily be queried with an information retrieval system.

```
Ex: 11 SELECT d.Doc1d FROM DOC d, INDEX I WHERE i.Term IN ("vehicle", "sales")
          AND d.PubDate = "3/23/87" AND d.DocId = i.DocId
```

5.3.3 Boolean Retrieval

A Boolean query is given with the usual operators-AND, OR, and NOT. The result set must contain all documents that satisfy the Boolean condition. For small bibliographic systems (e.g., card catalog systems), Boolean queries are useful. They quickly allow users to specify their information need and return all matches. For large document collections, they are less useful because the result set is unordered, and a query can result in thousands of matches. The user is then forced to tune the Boolean conditions and retry the query until the result is obtained. Relevance ranking avoids this problem by ranking documents based on a measure of relevance between the documents and the query. The user then looks at the top-ranked documents and determines whether or not they fill the information need. We start with the use of SQL to implement Boolean retrieval. We then show how a proximity search can be implemented with unchanged SQL, and finally, a relevance ranking implementation with SQL is described. The following SQL query returns all documents that contain an arbitrary term, InputTerm.

```
Ex: 12 SELECT DISTINCT (i.DocId) FROM INDEX i WHERE i.Term = InputTerm
```

Obtaining the actual text of the document can now be performed in an application specific fashion. The text is found in a single large attribute that contains a BLOB or CLOB (binary or character large object), possibly divided into separate components (i.e., paragraphs, lines, sentences, phrases, etc.). If the text is
found in a single large attribute (in this example we call it Text), the query can be extended to execute a subquery to obtain the document identifiers. Then the identifiers can be used to find the appropriate text in DOC.

```
Ex: 13 SELECT d.Text FROM DOC d WHERE d.DocId IN
            (SELECT DISTINCT (i.DocId) FROM INDEX i WHERE i.Term = InputTerm)
```

It is natural to attempt to extend the query in Example 12 to allow for n terms. If the Boolean request is an OR, the extension is straightforward and does not increase the number of joins found in the query.

```
Ex: 14 SELECT DISTINCT (i.DocId) FROM INDEX I WHERE i. Term = InputTerml OR
                                                    i. Term = InputTerm2 OR
                                                    i.Term = InputTerm3 OR
                                                           ….
                                                    i. Term = InputTermN
```

Unfortunately, a Boolean AND results in a dramatically more complex query. For a query containing n input terms, the INDEX relation must be joined n times. This results in the following query.

```
Ex: 15 SELECT a.DocId FROM INDEX a, INDEX b, INDEX c, ... INDEX n - 1, INDEX
          n WHERE a.Term = InputTerml AND
                  b.Term = InJYUtTerm2 AND
                  c.Term = InputTerm3 AND
                      …..
                  n.Term = InputTermn AND
                 a.DocId = b.DocId AND
                  b.DocId = c.DocId AND
                       …
                  n - l.DocId = n.DocId
```

Multiple joins are expensive. The order that the joins are computed affects performance, so a cost-based optimizer will compute costs for many of the orderings . Pruning is expensive. In addition to performance concerns, the reality is that commercial systems are unable to implement more than a fixed number of joins. Although it is theoretically possible to execute a join of n terms, most implementations impose limits on the number of joins (around sixteen is common). It is the complexity of this simple Boolean AND that has led many researchers to develop extensions to SQL or user-defined operators to allow for a more simplistic SQL query.
An approach that requires a fixed number of joins regardless of the number of terms found in the input query reduces the number of conditions in the query. However, an additional sort is needed (due to a GROUP BY) in the query where one previously did not exist.

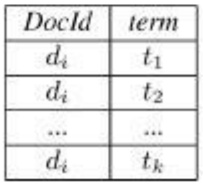
The following query computes a Boolean AND using standard syntactically fixed SQL:

```
Ex: 16 SELECT i.DocId FROM INDEX i, QUERY q WHERE i.Term = q.Term
```

GROUP BY i.DocId HAVING COUNT (i.Term) = (SELECT COUNT(*) FROM QUERY)

The WHERE clause ensures that only the terms in the query relation that match those in INDEX are included in the result set. The GROUP BY specifies that the result set is partitioned into groups of terms for each document. The HA VING ensures that the only groups in the result set will be those whose cardinality is equivalent to that of the query relation. For a query with k terms (t1, t2, ... , tk), the tuples as given in Table are generated for document d_i containing all k terms.

Table: Result Set



DocId	*term*
d_i	t_1
d_i	t_2
...	...
d_i	t_k

The GROUP BY clause causes the cardinality, k, of this document to be computed. At this point, the HAVING clause determines if the k terms in this group matches the number of terms in the query. If so, a tuple di appears in the final result set.

Until this point, we assumed that the INDEX relation contains only one occurrence of a given term for each document. This is consistent with our example where a term frequency is used to record the number of occurrences of a term within a document. In proximity searches, a term is stored multiple times in the INDEX relation for a single document. Hence, the query must be modified because a single term in a document might occur k times which results in d_i being placed in the final result set, even when it does not contain the remaining k - 1 terms.

The following query uses the DISTINCT keyword to ensure that only the distinct terms in the document are considered. This query is used on INDEX relations in which term repetition in a document results in term repetition in the INDEX relation.

```
Ex: 17 SELECT i.DocId FROM INDEX i, QUERY q WHERE i. Term = q. Term
          GROUP BY i.DocId HAVING COUNT(DISTINCT(i.Term)) = (SELECT COUNT(*)
          FROM QUERY)
```

This query executes whether or not duplicates are present, but if it is known that duplicate terms within a document do not occur, this query is somewhat less efficient than its predecessor. The DISTINCT keyword typically requires a sort.

Using a set-oriented approach to Boolean keyword searches results in the fortunate side-effect that a Threshold AND (TAND) is easily implemented. A partial AND is one in which the condition is true if k subconditions are true. All of the subconditions are not required. The following query returns all documents that have k or more terms matching those found in the query.

```
Ex: 18 SELECT i.DocId FROM INDEX i,QUERY q WHERE i. Term = q. Term
              GROUP BY i.DocId HAVING COUNT(DISTINCT(i.Term)) ≥k
```

5.3.4 Proximity Searches

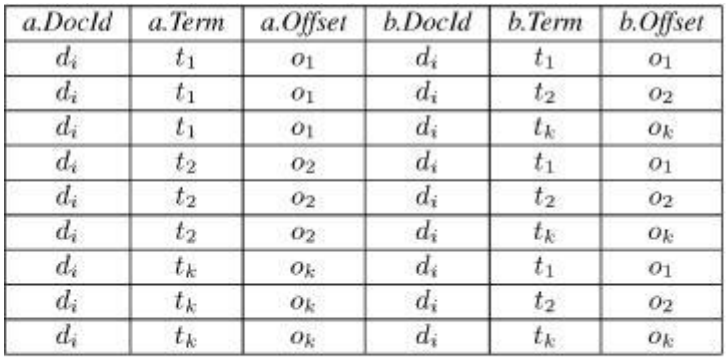
To briefly review, proximity searches are used in information retrieval systems to ensure that the terms in the query are found in a particular sequence or at least within a particular window of the document. Most users searching for a query of "vice president" do not wish to retrieve documents that contain the sentence, "His primary vice was yearning to be president of the company." To implement proximity searches, the INDEXYROX given in our working example is used. The offset attribute indicates the relative position of each term in the document.
The following query, albeit a little complicated at first glance, uses unchanged SQL to identify all documents that contain all of the terms in QUERY within a term window of width terms. For the query given in our working example, "vice" and "president" occur in positions seven and eight, respectively.
Document two would be retrieved if a window of two or larger was used.

```
Ex: 19 SELECT a.DocId FROM INDEXYROX a, INDEXJ>ROX b
          WHERE a.Term IN (SELECT q.Term FROM QUERY q) AND
                     b. Term IN (SELECT q. Term FROM QUERY q) AND
                     a.DocId = b.DocId AND
                    (b. Offset - a.Offset) BETWEEN 0 AND (width - 1)
          GROUP BY a.DocId, a.Term, a.Offset
           HAVING COUNT (DISTINCT (b.Term)) =(SELECT COUNT(*) FROM QUERY)
```

The INDEX_PROX table must be joined to itself since the distance between each term and every other term in the document must be evaluated. For a document di that contains k terms (t1, t2, ... , tk) in the corresponding term offsets of (01, 02, ... , Ok), the first two conditions ensure that we are only examining offsets for terms in the document that match those in the query. The third condition ensures that the offsets we are comparing do not span across documents.
The following tuples make the first three conditions evaluate to TRUE.
In following Table, we illustrate the logic of the query. Drawing out the first step of the join of INDEX_ PROX to itself for an arbitrary document di yields tuples in which each term in INDEX_TERM is matched with all other terms. This table shows only those terms within

document di that matched with other terms in document di. This is because only these tuples evaluate to TRUE when the condition "a.DocId = b.DocId" is applied. We also assume that the terms in the table below match those found in the query, thereby satisfying the condition "b.term IN (SELECT q.term FROM QUERY)."

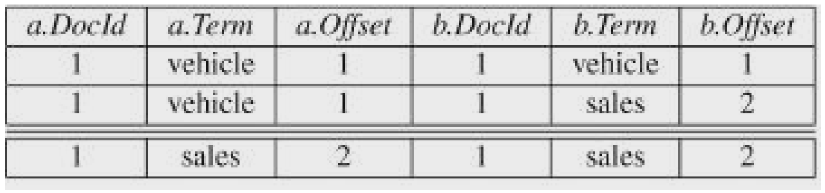
Table : Result of self-join of INDEX_PROX



a.DocId	*a.Term*	*a.Offset*	*b.DocId*	*b.Term*	*b.Offset*
d_i	t_1	o_1	d_i	t_1	o_1
d_i	t_1	o_1	d_i	t_2	o_2
d_i	t_1	o_1	d_i	t_k	o_k
d_i	t_2	o_2	d_i	t_1	o_1
d_i	t_2	o_2	d_i	t_2	o_2
d_i	t_2	o_2	d_i	t_k	o_k
d_i	t_k	o_k	d_i	t_1	o_1
d_i	t_k	o_k	d_i	t_2	o_2
d_i	t_k	o_k	d_i	t_k	o_k

The fourth condition examines the offsets and returns TRUE only if the terms exist within the specified window. The GROUP BY clause partitions each particular offset within a document. The HAVING clause ensures that the size of this partition is equal to the size of the query. If this is the case, the document has all of terms in QUERY within a window of size offset. Thus, document di is included in the final result set. For an example query with "vehicle" and "sales" within a two term window, all four conditions of the WHERE clause evaluate to TRUE for the following tuples. The first three have eliminated those terms that were not in the query, and the fourth eliminated those terms that were outside of the term window. The GROUP BY clause results in a partition in which "vehicle", at offset one, is in one partition and "sales", at offset two, is in the other partition. The first partition has two terms which match the size of the query, so document one is included in the final result set (see Table).

Table : Result after all four conditions of the WHERE clause



a.DocId	*a.Term*	*a.Offset*	*b.DocId*	*b.Term*	*b.Offset*
1	vehicle	1	1	vehicle	1
1	vehicle	1	1	sales	2
1	sales	2	1	sales	2

5.3.5 Computing Relevance Using Unchanged SQL

Relevance ranking is critical for large document collections as a Boolean query frequently returns many thousands of documents. Numerous algorithms exist to compute a measure of similarity between a query and a document. The vector-space model is commonly used. Systems based on this model have repeatedly performed well at the Text REtrieval Conference (TREC). Recall, that in the vector space model, documents and queries are represented by a vector of size t, where t is the number of distinct terms in the document collection. The distance between the query vector Q and the document vector Di is used to rank documents. The following dot product measure computes this distance:

$$SC(Q, D_i) = \sum_{j=1}^{t} w_{qj} \times d_{ij}$$

where Wqj is weight of the j^{th} term in the query q, and d_{ij} is the weight of the j^{th} term in the i^{th} document. In the simplest case, each component of the vector is assigned a weight of zero or one (one indicates that the term corresponding to this component exists). Numerous weighting schemes exist, an example of which is tf-idf Here, the term frequency is combined with the inverse document frequency.
The following SQL implements a dot product query with the tf-idf weight.

```
Ex: 20 SELECT i.DocId, SUM(q.tf*t.idf*i.tf*t.idf) FROM QUERY q, INDEX i, TERM
          t WHERE q.Term = t.Term AND i.Term = t.Term
                  GROUP BY i.DocId ORDER BY 2 DESC
```

The WHERE clause ensures that only terms found in QUERY are included in the computation. Since all terms not found in the query are given a zero weight in the query vector, they do not contribute to the summation. The idf is obtained from the TERM relation and is used to compute the tf-idf weight in the select-list. The ORDER BY clause ensures that the result is sorted by the similarity coefficient. At this point, we have used a simple similarity coefficient. Unchanged SQL can be used to implement these coefficients as well. Typically, the cosine coefficient or its variants is commonly used. The cosine coefficient is defined as:

$$SC(Q, D_i) = \frac{\sum_{j=1}^{t} w_{qj} d_{ij}}{\sqrt{\sum_{j=1}^{t} (d_{ij})^2 \sum_{j=1}^{t} (w_{qj})^2}}$$

The numerator is the same as the dot product, but the denominator requires a normalization which uses the size of the document vector and the size of the query vector. Each of these normalization factors could be computed at query time, but the syntax of the query becomes overly complex. To simplify the SQL, two separate relations are created: DOC_WT (DocId, Weight) and QUERY _WT (Weight). DOC_WT stores the size of the document vector for each document and QUERY _WT contains a single tuple that indicates the size of the query vector. These relations may be populated with the following SQL:

```
Ex: 21 INSERT INTO DOC_WT SELECT DocId, SQRT (SUM(i.tf* t.idf* i.tf*
            t.idf)) FROM INDEX i, TERM t WHERE i.Term = t.Term GROUP BY
            DocId
Ex: 22 INSERT INTO QRLWT SELECT SQRT (SUM(q.tf* t.idf * q.tf*
        t.idf)) FROM QUERY q, TERM t WHERE q.Term = t.Term
```

For each of these INSERT-SELECT statements, the weights for the vector are computed, squared, and then summed to obtain a total vector weight. The following query computes the cosine.

```
Ex: 23 SELECT i.DocId, SUM (q.tf* t.idf* i.tf* t.idf)/(dw. Weight * qw.
              Weight) FROM QUERY q, INDEX i, TERM t, DOCWT dw,
              QRY_WT qw
                        WHERE q. Term = t. Term AND i. Term = t. Term AND i.DocId =
                                                                          dw.DocId
                          GROUP BY i.DocId, dw. Weight, qw. Weight ORDER BY 2 DESC
```

The inner product is modified to use the normalized weights by joining the two new relations, DOC_W T and QRY _W T. An additional condition is added to the WHERE clause in order to obtain the weight for each document. To implement this coefficient, it is necessary to use the built-in square root function which is often present in many SQL implementations. We note that these queries can all be implemented without the non-standard square root function simply by squaring the entire coefficient. This modification does not affect the document ranking as as a≤ b□ $a^2 \leq b^2$ for a, b ≥0. For simplicity of presentation, we used a built-in sqrt function (which is present in many commercial SQL implementations) to compute the square root of an argument. Modifications to the SUM () element permit implementation of other similarity measures. For instance, with the additional computation and storage of some document statistics, (log of the average term frequency), some collection statistics (average document length and the number of documents) and term statistics (document frequency), pivoted normalization and a probabilistic measure can be implemented. Essentially, the only change is that the SUM operator is modified to contain new weights. The result is that fusion of multiple similarity measures can be easily implemented in SQL.

5.3.6 Relevance Feedback in the Relational Model

Relevance feedback can be supported using the relational model. Recall, relevance feedback is the process of adding new terms to a query based on documents presumed to be relevant in an

initial running of the query. Separate SQL statements were used for each of the following steps:
Step 1: Run the initial query. This is done using the SQL we have just described.

Step 2: Obtain the terms in the top n documents. A query of the INDEX relation given a list of document identifiers (these could be stored in a temporary relation generated by Step 1) will result in a distinct list of terms in the top n documents. This query will run significantly faster if the DBMS has the ability to limit the number of tuples returned by a single query (many commercial systems have this capability). An INSERT-SELECT can be used to insert the terms obtained in this query into the QUERY relation.

Step 3: Run the modified query. The SQL remains the same as used in Step 1.

5.3.7 A Relational Information Retrieval System

The need to integrate structured and unstructured data led to the development of a scalable, standard SQL-based information retrieval prototype engine called SIRE. The SIRE approach, initial built for the National Institutes of Health National Center for Complementary and Alternative Medicine, leverages the investment of the commercial relational database industry by running as an application of the Oracle DBMS. It also includes all the capabilities of the more traditional customized information retrieval approach. Furthermore, additional functionality common in the relational database world, such as concurrency control, recovery, security, portability, scalability, and robustness, are provided without additional effort. Such functionality is not common in the traditional information retrieval market. Also, since database vendors continuously improve these features and likewise incorporate advances made in hardware and software, a SIRE-based solution keeps up with the technology curve with less investment on the part of the user as compared to a more traditional (custom) information retrieval system solution. To demonstrate the applicability and versatility of SIRE, key information retrieval strategies and utilities such as leading similarity measures, proximity searching, n-grams, passages, phrase indexing, and relevance feedback were all implemented using standard SQL. By implementing SIRE on a host of relational database platforms including NCR DBC-1012, Microsoft SQL Server, Sybase, Oracle, IBM DB2 and SQLIDS, and even mySQL, system portability was demonstrated. Efficiency was enhanced using several optimization approaches and some specific to relational database technology. These included the use of a pruned index and query thresholds as well as clustered indexes. All of these optimizations reduced the I/O volume, hence significantly reduced query execution time. More recent related efforts have focused on scaling the SIRE-based approach using parallel technology and incorporating efficient document updating into the paradigm. With the constant changes to text available particularly in Web environments, updating of the documents is becoming a necessity. Traditionally, information retrieval was a "read only" environment. Early parallel processing efforts used an NCR machine configured with 24 processors and achieved a speedup of 22-fold.

5.4 Semi-Structured Search using a Relational Schema

Numerous proprietary approaches exist for searching eXtensible Markup Language (XML) documents, but these lack the ability to integrate with other structured or unstructured data. Relational systems have been used to support XML by building a separate relational schema to map to a particular XML schema or DTD (Document-type Definitions) .

5.4.1 Background

XML has become the standard for platform-independent data exchange. There were a variety of methods proposed for storing XML data and accessing them efficiently .One approach is a customized tree file structure, but this lacks portability and does not leverage existing database technology. Other approaches include building a database system specifically tailored to storing semi-structured data from the ground or using a full inverted index . There are several popular XML query languages.

5.4.2 Static Relational Schema to support XML-QL

We describe a static relational schema that supports arbitrary XML schema to provide support for XML query processing. Later, IIT Information Retrieval Laboratory (www.ir.iit.edu) shown that a full XML-QL query language could be built using this basic structure. This is done by translating semi-structured XML-QL to SQL. The use of a static schema accommodates data of any XML schema without the need for document-type definitions or XSchemas.

The static relational storage schema stores each unique XML path and its value from each document as a separate row in a relation. This is similar to the edge table named for the fact that each row corresponds to an edge in the XML graph representation. This static relational schema is capable of storing an arbitrary XML document. The hierarchy of XML documents is kept in tact such that any document indexed into the database can be reconstructed using only the information in the tables. The relations used are:

TAG_NAME (TagId, tag)   ATTRIBUTE (AttributeId, attribute)
TAG_PATH (TagId, path)   DOCUMENT (DocId, fileName)
INDEX (Id, parent, path, type, tagId, attrId, pos, value)

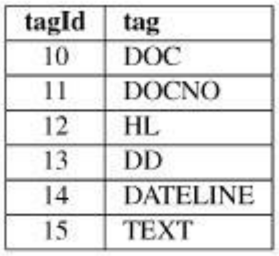
5.4.3 Storing XML Metadata

These tables store the metadata (data about the data) of the XML files. TAG_NAME and TAG_PATH together store the information about tags and paths within the XML file. TAG-.NAME stores the name of each unique tag in the XML collection. TAG_PATH stores the unique paths found in the XML documents. The ATTRIBUTE relation stores the names of all the attributes. In our example, we have added an attribute called LANGUAGE which is an attribute of the tag TEXT. In the TAG_NAME and TAG_PATH relations, the tagld is a unique key assigned by the preprocessing stage. Similarly, attributeId is uniquely assigned as well. As with our examples earlier in the chapter, these tables are populated each time a new XML file is indexed. This process consists of parsing the XML file and extracting all of this information and storing it into these tables.

5.4.4 Tracking XML Documents

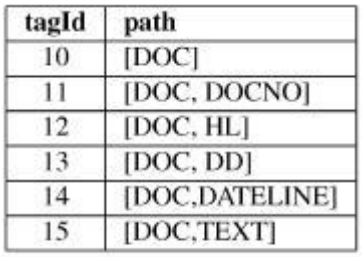
Since XML-QL allows users to specify what file(s) they wish to query, many times we do not want to look at each record in the database but only at a subset of records that correspond to that file. Each time a new file is indexed,

Table: TAG_NAME



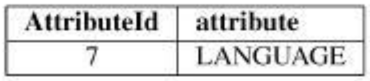
tagId	tag
10	DOC
11	DOCNO
12	HL
13	DD
14	DATELINE
15	TEXT

Table: TAG_PATH



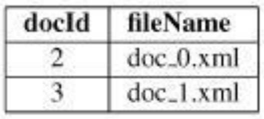
tagId	path
10	[DOC]
11	[DOC, DOCNO]
12	[DOC, HL]
13	[DOC, DD]
14	[DOC,DATELINE]
15	[DOC,TEXT]

Table: ATTRIBUTE



AttributeId	attribute
7	LANGUAGE

It receives a unique identifier that is known as the pin value. This value corresponds to a single XML file. The DOCUMENT relation contains a tuple for each XML document. For our example, we only store the actual file name that contains this document. Other relevant attributes might include the length of the document - or the normalized length .

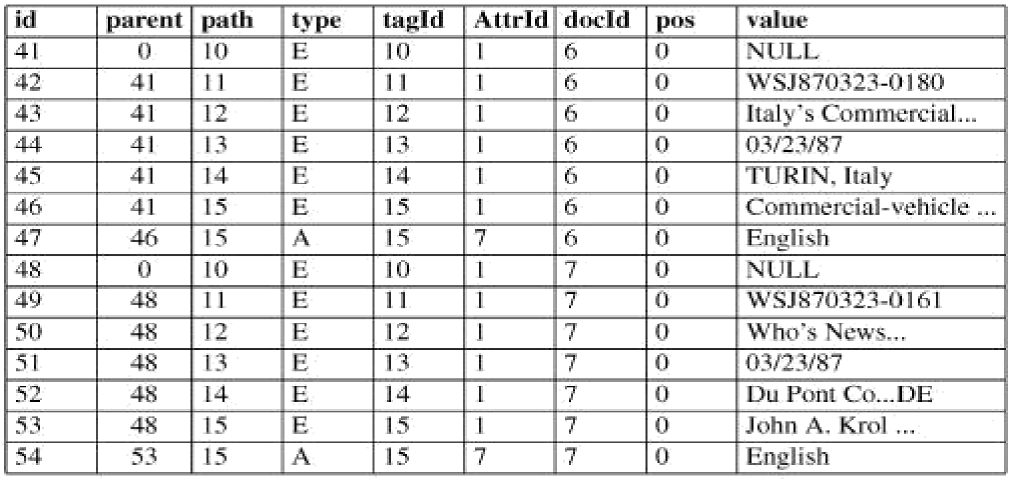


docId	fileName
2	doc_0.xml
3	doc_1.xml

Table : DOCUMENT

5.4.5 INDEX

The INDEX table models an XML index. It contains the mapping of each tag, attribute or value to each document that contains this value. Also, since the order of attributes and tags is important in XML (e.g.; there is a notion of the first occurrence, the second occurrence, etc.), the position or order of the tags is also stored. The id column is a unique integer assigned to each element and attribute in a document. The parent attribute indicates the id of the tag that is the parent of the current tag. This is needed to preserve the inherently hierarchical nature of XML documents. The path corresponds to the primary key value in the TagPath. The type indicates whether the path terminates with an element or an attribute (E or A). The Tagld and Attrld is a foreign key to the Tagld in the TagName and Attribute tables. The DocId attribute indicates the XML document for a given row. The pos tracks the order of a given tag and is used for queries that use the index expression feature of XML-QL and indicates the position of this element relative to others under the same parent (starting at zero). This column stores the original ordering of the input XML for explicit usage in users' queries. Finally, value contains the atomic unit in XML - the value inside the lowest level tags. Once we have reached the point of value, all of the prior means of using relations to model an inverted index for these values apply.



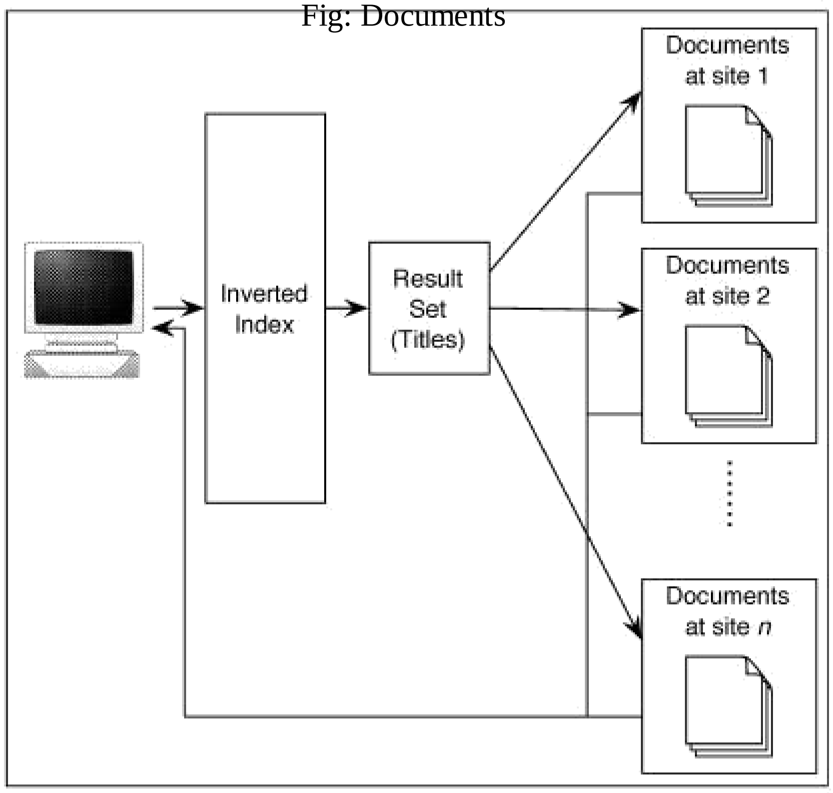
id	parent	path	type	tagId	AttrId	docId	pos	value
41	0	10	E	10	1	6	0	NULL
42	41	11	E	11	1	6	0	WSJ870323-0180
43	41	12	E	12	1	6	0	Italy's Commercial...
44	41	13	E	13	1	6	0	03/23/87
45	41	14	E	14	1	6	0	TURIN, Italy
46	41	15	E	15	1	6	0	Commercial-vehicle ...
47	46	15	A	15	7	6	0	English
48	0	10	E	10	1	7	0	NULL
49	48	11	E	11	1	7	0	WSJ870323-0161
50	48	12	E	12	1	7	0	Who's News...
51	48	13	E	13	1	7	0	03/23/87
52	48	14	E	14	1	7	0	Du Pont Co...DE
53	48	15	E	15	1	7	0	John A. Krol ...
54	53	15	A	15	7	7	0	English

Table: INDEX

5.5 DISTRIBUTED INFORMATION RETRIEVAL

Until now, we focused on the use of a single machine to provide an information retrieval service and the use of a single machine with multiple processors to improve performance. Although efficient performance is critical for user acceptance of the system, today, document collections are often scattered across many different geographical areas. Thus, the ability to process the data where they are located is arguably even more important than the ability to efficiently process them. Possible constraints prohibiting the centralization of the data include data security, their sheer volume prohibiting their physical transfer, their rate of change, political and legal constraints, as well as other proprietary motivations. One of the latest popular processing infrastructures is the "Grid". The grid is named after the global electrical power grid. In the power grid, appliances (systems in our domain) simply "plug in" and immediately operate and become readily available for use or access by the global community. A similar notion in modem search world is the use of Distributed Information Retrieval Systems (DIRS). DIRS provides access to data located in many different geographical areas on many different machines. In the early 1980's, it was already clear that distributed information retrieval systems would become a necessity. Initially, a theoretical model was developed that described some of the key components of a distributed information retrieval system.

Fig: Documents



5.5.1 A Theoretical Model of Distributed Retrieval

We first define a model for a centralized information retrieval system and then expand that model to include a distributed information retrieval system.

5.5.1.1 Centralized Information Retrieval System Model

Formally, an information retrieval system is defined as a triple, I = (D, R, δ) where D is a document collection, R is the set of queries, and $\delta j : Rj\text{-} 2^{Dj}$ is a mapping assigning the jth query to a set of relevant documents.

Many information retrieval systems rely upon a thesaurus, in which a user query is expanded, to include synonyms of the keywords to match synonyms in a document. Hence, a query that contains the term curtain will also include documents containing the term drapery. To include the thesaurus in the model, The triple be expanded to a quadruple as:

$$I= (T,D,R, \delta)$$

where T is a set of distinct terms and the relation ρ∩T x T such that ρ (tl, t2) implies that tl is a synonym of t2. Using the synonym relation, it is possible to represent documents as a set of descriptors and a set of ascriptors. Consider a document Di , the set of descriptors d consists of all terms in DI such that:

- Each descriptor is unique

- No descriptor is a synonym of another descriptor

An ascriptor is defined as a term that is a synonym of a descriptor. Each ascriptor must be synonymous with only one descriptor. Hence, the descriptors represent a minimal description of the document. The generalization relation assumes that it is possible to construct a hierarchical knowledge base of all pairs of descriptors. Construction of such knowledge bases was attempted both automatically and manually, but many terms are difficult to define. Relationships pertaining to spatial and temporal substances, ideas, beliefs, etc. tend to be difficult to represent in this fashion. However, this model does not discuss how to construct such a knowledge base, only some interesting properties that occur if one could be constructed.

The motivation behind the use of a thesaurus is to simplify the description of a document to only those terms that are not synonymous with one another. The idea being that additional synonyms do not add to the semantic value of the document. The generalization relation is used to allow for the processing of a query that states "List all animals" to return documents that include information about dogs, cats etc. even though the term dog or cat does not appear in the document.

The generalization can then be used to define a partial ordering of documents. Let the partial ordering be denoted by ≤ and let t(di) indicate the list of descriptors for document di . Partial ordering,≤, is defined as:

$$t(d_1) \preceq t(d_2) \Leftrightarrow (\forall t' \in t(d_1))(\exists t'' \in t(d_2))(\gamma(t', t''))$$

Hence, a document d1 whose descriptors are all generalizations of the descriptors found in d2 will have the ordering d1 ≤ d2. For example, a document with the terms animal and person will precede a document with terms dog and John. Note that this is a partial ordering because two documents with terms that have no relationship between any pairs of terms will be unordered. To be inclusive, the documents that correspond to a general query q1 must include (be a superset ot) all documents that correspond to the documents that correspond to a more specific query q2, where q1 ≤ q2. Formally:

$$(q_1, q_2 \in Q) \wedge (q_1 \preceq q_2) \rightarrow (\delta(q_1) \supset \delta(q_2))$$

This means that if two queries, q1 and q2, are presented to a system such that q1 is more general than q2, it is not necessary to retrieve from the entire document collection for each query. It is only necessary to obtain the answer set for ɣ(q1) to obtain the ɣ(q2).

5.5.1.2 Distributed Information Retrieval System Model

The centralized information retrieval system can be partitioned into n local information retrieval systems s1,s2 , ... ,sn . Each system sj is of the form: sj = (Tj, D j , Rj , δj), where Tj is the thesaurus; D j is the document collection; Rj the set of queries; and δj : Rj □ 2^{Dj} maps the queries to documents. By taking the union of the local sites, it is possible to define the distributed information retrieval system as:

s= (T,D,R, δ)

where:

$$T = \bigcup_{j=1}^{n} T_j$$

$$s_j = s \bigcap (T_j \times T_j), R_j = R \bigcap (d_j \times d_j)$$

This state that the global thesaurus can be reconstructed from the local thesauri, and the queries at the sites j will only include descriptors at site j. This is done so that the terms found in the query that are not descriptors will not retrieve any documents.

$$D = \bigcup_{j=1}^{n} D_j$$

The document collection, D, can be constructed by combining the document collection at each site.

$$R \supset \bigcup_{j=1}^{n} R_j, \preceq_j = \preceq \bigcap (R_j \times R_j)$$

The queries can be obtained by combining the queries at each local site. The partial ordering defined at site j will only pertain to queries at site j.

$$(\forall r \in R)(\delta(r) = d : d \in D \wedge r \preceq t(d))$$

For each query in the system, the document collection for that query contains documents in the collection where the documents are at least as specific as the query. The hierarchy represented by I is partitioned among the different sites. A query sent to the originating site would be sent to each local site and a local query would be performed. The local responses are sent to the originating site where they are combined into a final result set. The model allows for this methodology if the local sites satisfy the criteria of being a subsystem of the information retrieval system.

$S_1 = (T_1, D_1, R_1, \delta_1)$ **is a subsystem of** $S_2 = (T_2, D_2, R_2, \delta_2)$ **if:**

$$(T_1 \supset T_2) \wedge (R_1 = R_2) \bigcap (d_1 \times d_2) \wedge (s_1 = s_2) \bigcap (T_1 \times T_2)$$

The thesaurus of T1 is a superset of T2

$$D_1 \supset D_2$$

The document collection at site s1 contains the collection D2

$$R_1 \in R_2 \wedge \preceq_1 = \preceq_2 \bigcap (R_1 \times R_2)$$

The queries at site s1 contain those found in s2.

$$\delta_1(r) = \delta_2(r) \bigcap D_1 for\ r \in R$$

The document collection returned by queries in s1 will include all documents returned by queries in s2.The following example illustrates that an arbitrary partition of a hierarchy may not yield valid subsystems.

Consider the people hierarchy:

ɣ(people, Harold), ɣ(people, Herbert), ɣ (people, Mary)

and the second animal hierarchy:

ɣ(animal, cat), ɣ(animal, dog), ɣ,(cat, black-cat), ɣ(cat, cheshire), ɣ(dog, doberman), ɣ(dog, poodle)

Assume that the hierarchy is split into sites s1 and s2. The hierarchy at s1 is:

ɣ(people, Harold), ɣpeople, Mary)

ɣ(animal, cat), ɣ(animal, dog), ɣ(dog, doberman), ɣ(dog, poodle)

The hierarchy at s2 is:

ɣ(people, Herbert), ɣ(people, Harold), ɣ(animal, cat), ɣ(animal, doberman), ɣ(cat, cheshire), ɣ(cat, black-cat)

Consider a set of documents with the following descriptors:

D1 = (Mary, Harold, Herbert)

D2 = (Herbert, dog)

D3 = (people, dog)

D4 = (Mary, cheshire)

D5 = (Mary, dog)

D6 = (Herbert, black-cat, doberman)

D7 = (Herbert, doberman)

A query of the most general terms (people, animal) should return all documents 2 through 7 (document 1 contains no animals, and the query is effectively a Boolean AND). However, the hierarchy given above as s1 will only retrieve documents D3 and D5 , and s2 will only retrieve documents D6 and D7. Hence, documents D2 and D4 are missing from the final result if the local results sets are simply concatenated. Since, the document collections cannot simply be concatenated, the information retrieval systems at sites s1 and s2 fail to meet the necessary criterion to establish a subsystem.

In practical applications, there is another problem with the use of a generalization hierarchy. Not only are they hard to construct, but also it is non-trivial to partition them. This distributed model was expanded to include weighted keywords for use with relevance.

5.6 Web Search

Search tools that access Web pages via the Internet are prime examples of the implementation of many of the algorithms and heuristics. These systems are, by nature, distributed in that they access data stored on Web servers around the world. Most of these systems have a centralized index, but all of them store pointers in the form of hypertext links to various Web servers. These systems service tens of millions of user queries a day, and all of them index several Terabytes of Web pages. We do not describe each search engine in vast detail because search engines change very. We note that sixteen different Web search engines are listed at www.searchenginewatch.com while www.searchengineguide.com lists over 2,500 specialized search engines.

5.6.1 Evaluation of Web Search Engines

The traditional information retrieval environments, individual systems are evaluated using standard queries and data. In the Web environment, such evaluation conditions are unavailable. Furthermore, manual evaluations on any grand scale are virtually impossible due to the vast size and dynamic nature of the Web. To automatically evaluate Web search engines, a method using online taxonomies that were created as part of Open Directory Project (ODP) . Online directories were used as known relevant items for a query. If a query matches either the title of the item stored or the directory file name containing a known item then it is considered a match. The authors compared the system rankings achieved using this automated approach versus a limited scale, human user based system rankings created using multiple individual users. The two sets of rankings were statistically identical.

5.6.2 High Precision Search

Another concern in evaluating Web search engines is the differing measures of success as compared to traditional environments. Traditionally, precision and recall measures are the main evaluation metrics, while response time and space requirements are likely addressed. However, in the Web environment, response time is critical. Furthermore, recall estimation is very difficult, and precision is of limited concern since most users never access any links that appear beyond the first answer screen (first ten potential reference links). Thus, Web search engine developers focus on guaranteeing that the first results screen is generated quickly, is highly accurate, and that no severe accuracy mismatch exists. Text is efficiently extracted from template generated Web documents; the remainders of the frame or frames are discarded to prevent identifying a document as relevant as a result of potentially an advertisement frame matching the query. Efficient, high-precision measures are used to quickly sift and discard any item that is not with great certainty relevant as a top-line item to display in a current news listing service.

5.6.3 Query Log Analysis

In summary, although similar and relying on much the same techniques as used in traditional information retrieval system domains, the Web environment provides for many new opportunities to revisit old issues particularly in terms of performance and accuracy optimizations and evaluation measures of search accuracy. In that light, recently, an hourly analysis of a very large topically categorized Web query log was published. Using the results presented, it is possible to generate many system optimizations. For example, as indicated in the findings presented, user request patterns repeat according to the time of day and day of week. Thus, depending on the time of day and day of week, it is possible to pre-cache likely Web pages in anticipation of a set of user requests. Thus, page access delays are reduced increasing system throughput. Furthermore, in terms of accuracy optimization, it is likewise possible to adjust the ranking measures to better tune for certain anticipated user subject requests. In short, many optimizations are possible.

5.6.4 Page Rank

The most popular algorithm for improving Web search is PageRank algorithm (named after Page) was first. It extends the notion of hubs and authorities in the Web graph. PageRank is at the heart of the popular Web search engine, Google. Essentially, the Page Rank algorithm uses incoming and outgoing links to adjust the score of a Web page with respect to its popularity, independent of the user's query. Hence, if a traditional retrieval strategy might have previously ranked two documents equal, the PageRank algorithm will boost the similarity measure for a popular document. Here, popular is defined as having a number of other Web pages link to the document. This algorithm works well on Web pages, but has no bearing on documents that do not have any hyperlinks. The calculation of PageRank for page A over all pages linking to it D1 ... Dn is defined as follows:

$$PageRank(A) = (1-d) + d \sum_{D_1 \ldots D_n} \frac{PageRank(D_i)}{C(D_i)}$$

where C(Di) is the number of links out from page Di and d is a dampening factor from 0-1. This dampening factor serves to give some non-zero PageRank to pages that have no links to them. It also smooths the weight given to other links when determining a given page's PageRank. This significantly affects the time needed for PageRank to converge. The calculation is performed iteratively. Initially all pages are assigned an arbitrary PageRank. The calculation is repeated using the previously calculated scores until the new scores do not change significantly. The example in above Figure, using the common
dampening factor of 0.85 and initializing each PageRank to 1.0, it took 8 iterations before the scores converged.
Simple pagerank calculation

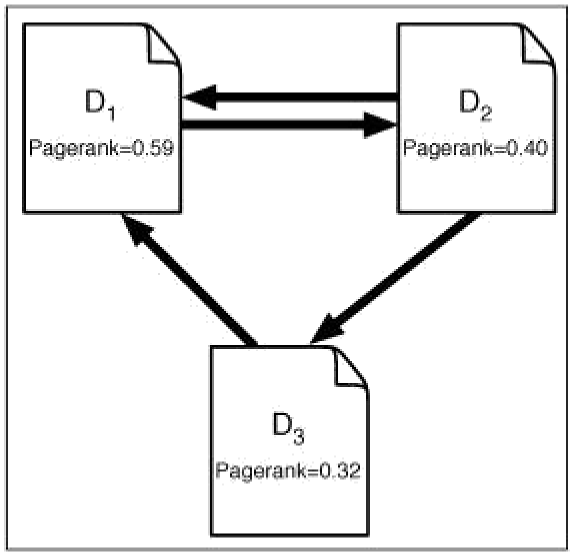


Fig: Page rank

5.6.5 Improving Effectiveness of Web Search Engines

Using a Web server to implement an information retrieval system does not dramatically vary the types of algorithms that might be used. For a single machine, all of the algorithms given in Chapter 5 are relevant. Compression of the inverted index is the same, partial relevance ranking is the same, etc.

However, there were and are some efforts specifically focused on improving the performance of Web-based information retrieval systems. In terms of accuracy improvements, it is reasonable to believe that by sending a request to a variety of different search engines and merging the obtained results one could improve the accuracy of a Web search engine. This was proposed as early as TREC-4. Later, the CYBERosetta prototype system developed by the Software Productivity Consortium (SPC) for use by DARPA, identical copies of a request were simultaneously sent to multiple search servers. After a timeout limit was reached, the obtained results were merged into a single result and presented to the user.