G.PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY,KURNOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

IV B.TECH I SEM (R13)

INFORMATION RETRIEVAL SYSTEMS

UNIT-4

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

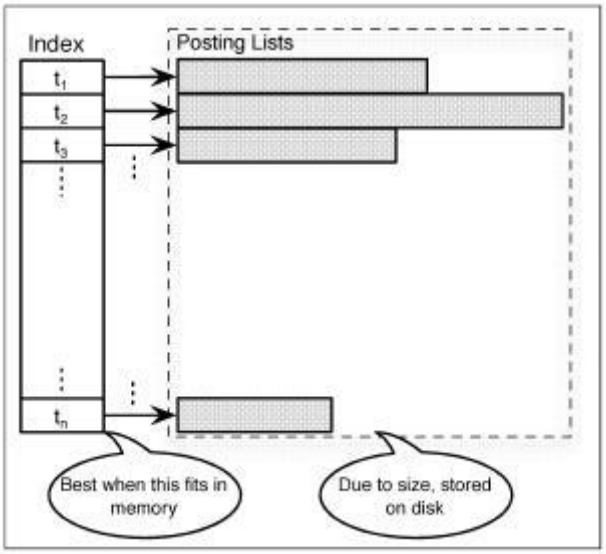
An inverted index consists of two components, a list of each distinct term referred to as the *index* and a set of lists referred to as *posting lists.* To compute relevance ranking, the term frequency or weight must be maintained. Thus, a posting list contains a set of tuples for each distinct term 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:

$$sales \rightarrow (1, 2)$$
$$vehicle \rightarrow (1, 1)\,(2, 1)$$

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:



$$t \rightarrow (d_1, tf_{1j}), (d_2, tf_{2j}), \ldots, (d_i, tf_{ij})$$

where d_i 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.

x	γ
1	0
2	10 0
3	10 1
4	110 00
5	110 01
6	110 10
7	110 11
8	1110 000

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	11111110 0110100

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

$$\sum_{j=1}^{k-1} v_j < x \leq \sum_{j=1}^{k} v_j$$

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:

$$d = x - \sum_{j=1}^{k-1} v_j - 1$$

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 eliminates 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 a 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 termas *t f max* and uses the following as a means of sorting the query.

$$tf_{max} \times idf$$

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 a 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.

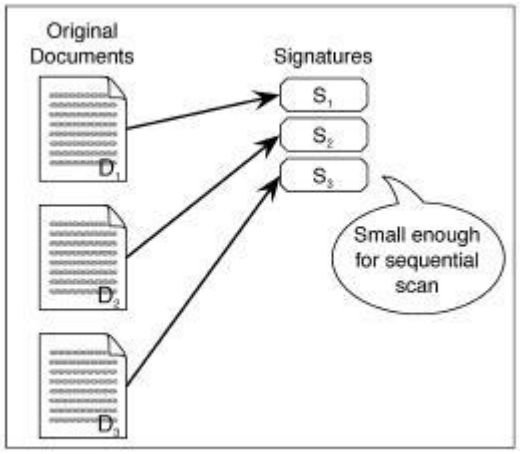


Fig: Information Retrieval.·Algorithms And Heuristics

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.

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

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 a 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

$$r(D_i, D_j) = \frac{|S(D_i) \bigcap S(D_j)|}{|S(D_i) \bigcup S(D_j)|}$$

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.