Table 5.2. Storage Requirements Based on Integer Size

Length	Number of Bytes Required
$0 \leq x < 64$	1
$64 \leq x < 16,384$	2
$16,384 \leq x < 4,194,304$	3
$4,194,304 \leq x < 1,073,741,824$	4

tifier 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 $2^6 = 64$ 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. (Note: this statement is not exactly accurate. Since there is no need for zero displacement, it is possible to store a displacement of one less than is actually needed. Since such a storage approach always requires an additional increment operation, and it only favors borderline conditions, it is seldom used.) Table 5.2 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.

Table 5.3. Byte-Aligned Compression

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

Table 5.4. Baseline: No 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

5.1.2.2 Example: Fixed Length Compression

Consider an entry for an arbitrary term, t_1, which indicates that t_1 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 are shown in Table 5.3.

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 5.4.

In this example, uncompressed data requires 160 bits, while BA compression requires only 48 bits.

5.1.3 Variable Length Index Compression

Moffat and Zobel also use the differences in the posting list. They capitalize on the fact that for most long posting lists, the difference between two

Table 5.5. Gamma Encoding: First Eight Integers

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

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.

Moffat and Zobel use a family of universal codes described in [Elias, 1975]. This code represents an integer x with $2\lfloor log_2 x\rfloor + 1$ bits. The first $\lfloor log_2 x\rfloor$ bits are the unary representation of $\lfloor log_2 x\rfloor$. (Unary representation is a base one representation of integers using only the digit one. The number 5_{10} is represented as 11111_1.) 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 $\lfloor log_2 x\rfloor$ bits represent the remainder of $x - 2^{\lfloor \log_2 x\rfloor}$ in binary.

As an example, consider the compression of the decimal 14. First, $\lfloor log_2 x\rfloor =$ 3 is represented in unary as 111. Next, the stop bit is used. Subsequently, the remainder of $x - 2^{\lfloor \log_2 x\rfloor} = 14 - 8 = 6$ is stored in binary using $\lfloor \log_2 14\rfloor = 3$ bits as 110. Hence, the compressed code for 14_{10} 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 γ encoding are given in Table 5.5:

Table 5.6. Gamma Compression

Value	Compressed Bit String
1	0
2	10 0
4	110 00
63	111110 11111
180	11111110 0110100

5.1.3.1 Example: Variable Length Compression

For our same example, the differences of 1, 2, 4, 63, and 180 are given in Table 5.6. 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 0. Moffat and Zobel use the γ code to compress the term frequency in a posting list, but use a more complex coding scheme for the posting list entries.

5.1.4 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 $\sum v_i \geq N$. To code integer $x \geq 1$ 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 $\lceil \log_2 v_k \rceil = \lceil \log_2 4 \rceil = 2$ binary bits. With this generalization, the γ scheme

Table 5.7. Variable Compression based on Posting List Size

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

can be seen as using the vector V composed of powers of 2 $<1, 2, 4, 8, \ldots, >$ 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 [Zobel et al., 1992] was to vary V for each posting list such that V = $< b, 2b, 4b, 8b, 16b, 32b, 64b, \ldots, >$ where b is the median offset given in the posting list.

5.1.4.1 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 5.7 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.

5.1.4.2 Throughput-optimized Compression

Anh and Moffat developed an index compression scheme that yields good compression ratios while maintaining fast decompression time for efficient query processing [Anh and Moffat, 2004]. 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-10: 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-10 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.

5.1.4.3 Example: Simple-9

To continue our example using the differences of 1, 2, 4, 63, and 180, we show the selection table in Table 5.8. To find the appropriate coding scheme, we examine each row in the table. We cannot use row a because there is a value in the first 28 offsets greater than 2^1. We can not use row b because there is a value in the first 14 offsets greater than 2^2. Continuing down the list, we find that we also can not use row e because there is a value in the first five offsets greater than 2^5. We must use row f, since the highest value in the first four offsets (63) is less than $2^7 = 128$. This yields four 7-bit codes for the first four offsets (see Table 5.9), along with row f encoded in four bits (e.g., 0101).

The final offset in our example, 180, will not fit within the first 32-bit word; therefore, a second 32-bit word is needed to encode it. Thus, 64 total bits are required to compress our example using Simple-9. This example illustrates that these compression schemes are most effective with longer posting lists. Additionally, it should be clear that these schemes allow for very fast decompression time, as each value is encoded at a fixed length within each word.

5.1.4.4 Block-addressing compressed indexes

Another method of reducing index size is to build an index that addresses blocks of text with fixed sizes which may contain more than one document. Specific term counts can then be obtained by linearly scanning these blocks.

Table 5.8. Nine Different Ways of Partitioning 28 Data Bits (for Simple-9)

Selector	Codes	Length (bits)	Number of Unused bits
a	28	1	0
b	14	2	0
c	9	3	1
d	7	4	0
e	5	5	3
f	4	7	0
g	3	9	1
h	2	14	0
i	1	28	0

Table 5.9. Example of Simple-9 Compression

Value	Compressed Bit String
1	0000000
2	0000001
4	0000011
63	0111110

This allows for an adjustable balance between the time to create the index and the storage used for the index and query processing speed [Navarro et al., 2000].

5.1.5 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 described in [Carmel et al., 2001]. 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. A hardware implementation of this basic approach is described in [Agun and Frieder, 2003].

5.1.6 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 [Blandford and Blelloch, 2002, Silvestri et al., 2004]. Since the compression effectiveness of many 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 d_1, d_{99}, and d_{1000}, all which contain the same term t. For these documents we obtain a posting list entry for t of $t \rightarrow d_1, d_{51}, d_{101}$.

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 d_1, d_2, and d_3 which completely eliminates this gap. We note that for D documents there are 2^D 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. The idea is that similar documents will contain similar terms and document gaps are reduced if we order documents based on their similarity to one another. Each algorithm uses the Jaccard similarity coefficient (see Section 2.1.2) to obtain a measure of document similarity. Two basic approaches have been proposed: top-down (starting from the collection as a whole) or bottom-up (starting from each individual document).

5.1.6.1 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 described in [Blandford and Blelloch, 2002]. 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.

5.1.6.2 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 (see Section 3.2.3.3). The second uses *k-scan*; 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 single-pass 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 (see Section 3.2.3.4).

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.

5.2 Query Processing

Recent work has focused on improving query run-time. Moffat and Zobel have shown that query performance can be improved by modifying the inverted index to support fast scanning of a posting list [Moffat and Zobel, 1996, Moffat and Zobel, 1994]. Other work has shown that reasonable precision and recall can be obtained by retrieving fewer terms in the query [Grossman et al., 1997]. Computation can be reduced even further by eliminating some of the complexity found in the vector space model [Lee and Ren, 1996].

5.2.1 Inverted Index Modifications

Moffat and Zobel show how an inverted index can be segmented to allow for a quick search of a posting list to locate a particular document [Witten et al., 1999]. 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. Moffat and Zobel suggest that 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 sim-

ply 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 (they only increment documents who contain another term in the query) 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. Moffat and Zobel propose to change the posting list by partitioning it and adding pointers to each partition. The posting list can quickly be scanned by checking the first partition pointer (which contains the document identifier of the highest document in the partition and a pointer to the next partition). 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 [Moffat and Zobel, 1996].

5.2.2 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 [Yee et al., 1993]. 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.

5.2.2.1 Cutoff Based on Document Frequency

The simplest measure of term quality is to rely on document frequency. This was described in [Grossman et al., 1997, Grossman et al., 1994] 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.

5.2.2.2 Cutoff Based on Maximum Estimated Weight

Two other measures of sorting the query terms are described in [Yee et al., 1993]. The first computes the maximum term frequency of a given query term as tf_{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.

5.2.2.3 Cutoff Based on the Weight of a Disk Page in the Posting List

The cutoffs based on term weights can be used to characterize posting lists and choose which posting list to process first. The problem is that a posting list can be quite long and might have substantial skew. To avoid this problem, a new measure sorts disk pages within a posting list instead of the entire posting list. At index creation time, the posting lists are sorted in decreasing order by

term frequency and instead of just a pointer that points to the first entry in the posting list, the index contains an entry for each page of the posting list. The entry indicates the maximum term frequency on a given page. The posting list pages are then sorted by:

$$tf_{max} \times idf \times f(I)$$

where $f(I)$ is a function that indicates the number of entries on a page. This is necessary since some pages will not be full and a normalization is needed such that they are not sorted in exactly the same way as a full page. One value that is used for $f(I)$ is i^e where $0 < e < 1$.

Unfortunately, this measure requires an entry in the index for each page in the posting list. However, results show (for a variety of query sizes) that only about forty percent of the disk pages need to be retrieved to obtain eighty percent of the documents that would be found if all one hundred percent of the pages were accessed. All of these tests were performed using small document collections.

5.2.3 Vector Space Simplifications

Recent work has shown, in many cases, that simplifications to the vector space model can be made with only limited degradation in precision and recall [Lee et al., 1997]. In this work, five variations to the basic cosine measure (see Section 2.1) were tested on five small collections and 10,000 articles from the *Wall Street Journal* portion of the TREC collection. To review, the baseline cosine coefficient is:

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

The first variation was to replace the document length normalization that is based on weight with the square root of the number of terms in D_i. The second variation was to simply remove the document length normalization (simple dot product coefficient) given by:

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

The third measure drops the idf. This eliminates one entry in the index for each term.

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

The fourth measure drops the tf but retains the idf. This eliminates the need to store the tf in each entry of the posting list. This significantly reduces both computational, storage, and I/O costs.

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

The weight, w_{qj}, is one if term j is in the query and zero if otherwise. The weight, w_{ij} is equal to idf_j if term j is in the document and zero otherwise.

The fifth and final method simply counts matches between the query and the terms. That is:

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

where w_{qj} is one if term j is in the query and zero otherwise, and w_{ij} is equal to one if term j is in the document and zero otherwise.

For the TREC subset, two tests were done. The first was with the TREC narratives (long queries) and the second was with the TREC concepts (short queries). With the narratives, the baseline cosine measure performed the best with the square root document length normalization doing slightly better. The concept queries had the interesting result that the fourth and fifth (no idf and simple match counting) methods had a higher precision than the baseline. The only explanation for this somewhat surprising result is that the concept queries are very specific in nature so the effect of additional weights did not have much impact.

5.3 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 (often the goal is to represent a signature in only a few bits). Once this is done, the signatures can be scanned instead of the entire documents. Typically, signatures do not uniquely represent a document (i.e., a signature represents multiple documents), so it is usually necessary to implement a retrieval in two phases. 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. Of-

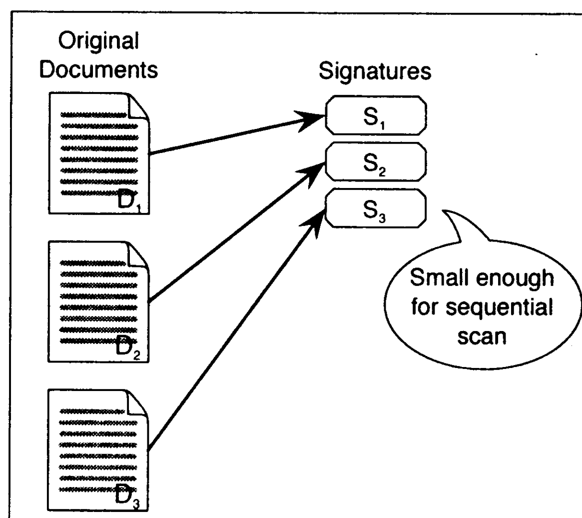


Table 5.10. Building a Signature

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

ten, 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 1001.

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 t_1 that sets bits one and three in the signature and another term t_2 that sets bits two and four in the signature. A third term t_3 might correspond to bits one and two and thereby be deemed a *match* with the signature, even though it is not equal to t_1 or t_2. Table 5.10 gives the three terms just discussed and their corresponding hash values.

Consider document d_1 that contains t_1, document d_2 contains t_1 and t_3 and document d_3 contains t_1 and t_2. Table 5.11 has the signatures for these three documents.

Table 5.11. Document Signatures

Document	Signature
d_1	0101
d_2	0111
d_3	1111

Hence, a query that is searching for term t_3 will obtain a false match on document d_3 even though it does not contain t_3.

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 [Stanfill and Thau, 1991].

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 [Stanfill and Kahle, 1986].

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 (see Section 7.1.2). 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 [Lee and Ren, 1996].

5.3.1 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 is given in [Lecroq, 1994]. Signature al-

gorithms can be employed without scanning for false drops (if a long enough signature is used) and no significant degradation in precision and recall occurs [Lee and Ren, 1996]. 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.

Aho and Corasick's algorithms implement deterministic finite state automata to identify matches in the text [Aho and Corasick, 1975]. Knuth, Morris, 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 [Knuth et al., 1977].

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 [Boyer and Moore, 1977]. The algorithm was improved to run in linear time even when multiple occurrences of the pattern are present [Galil, 1979].

Later, in the 1980's, a pattern matching algorithm that works by applying a hash function to the pattern and the next p characters in the text was given in [Karp and Rabin, 1987]. 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 in [Frakes and Baeza-Yates, 1993]. 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 αs where α is some constant $0 \leq \alpha \leq 1.0$ 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 [Uratani and Takeda, 1993]. 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. For only two patterns, the average probe ratio (the ratio of the number of references in the text and the length of the text) of Boyer-Moore is 0.236 while Uratani-Takeda is 0.178.

5.4 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 [Chowdhury et al., 2002, Cho et al., 1999].

5.4.1 Finding Exact Duplicates

Duplicate detection is often implemented by calculating a unique hash value 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 MD2, MD5, or SHA [SHA1, 1995]. 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.

5.4.2 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* [Broder et al., 1997]. 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.

$$r(D_i, D_j) = \frac{|S(D_i) \bigcap S(D_j)|}{|S(D_i) \bigcup S(D_j)|} \quad (5.1)$$

The resemblance r of document D_i and document D_j, as defined in Equation 5.1, 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 D_i and D_j share fifty percent of their terms and each document has 10 terms. Their resemblance would be $\frac{5}{15} = 0.33$. Many researchers have explored using resemblance to provide a threshold t to find duplicate documents [Brin et al., 1995, Garcia-Molina et al., 1996, Shivakumar and Garcia-Molina, 1996, Broder et al., 1997, Shivakumar and Garcia-Molina, 1998, Fetterly et al., 2003] where if t was exceeded the documents would be considered duplicate.

Two general issues were explored when using resemblance. 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 [Broder, 1998]. The cosine measure (see Section 2.1.2) is commonly used to 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.

5.4.2.1 Shingles

The first near-duplicate algorithm we discuss is the use of shingles. A shingle is simply a set of contiguous terms in a document. Shingling techniques, such as COPS [Brin et al., 1995], KOALA [Heintze, 1996], and DSC [Broder, 1998], essentially all compare the number of matching shingles.

The comparison of document subsets allows the algorithms to calculate a percentage of overlap between two documents using resemblance as given in Equation 5.1. 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. Frequently occurring shingles are removed in [Heintze, 1996]. Every 25th shingle is saved in [Broder et al., 1997]. This reduction, however, hinders effectiveness. Worse still, even when relatively infrequent shingles are removed (only those that occur in over 1000 documents) and keeping only every 25th shingle, the implementation of the DSC algorithm took 10 CPU days to process 30 million documents [Broder, 1998].

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. The authors, however, noted that DSC-SS does "not work well for short documents" so no runtime results were reported [Broder, 1998]. This makes sense because super shingles tend to be somewhat large and will, in all likelihood, completely encompass a short document.

5.4.2.2 Duplicate Detection via Similarity

Another approach is to simply compute the similarity coefficient between two documents. If the document similarity exceeds a threshold, the documents can be deemed duplicates of each other [Sanderson, 1997, Buckley et al., 2000, Hoad and Zobel, 2002]. These approaches are similar to work done in document clustering (see Section 3.2). Unfortunately, they require all pairs of documents to be compared, i.e., each document is compared to every other document and a similarity weight is calculated. A document to document similarity comparison approach is thus computationally prohibitive given the theoretical $O(d^2)$ runtime, where d is the number of documents. In reality, these approaches only evaluate documents with an overlap of terms. Thus, the actual runtime is data dependent and difficult to accurately predict.

5.4.2.3 Treating the Document as a Query

Another approach treats each result document as a new query and looks for other documents that match this document. This approach is not computationally feasible for large collections or dynamic collections since each document must be queried against the entire collection. Sanderson and Cooper used a variation on this where the terms are selected using Rocchio relevance feedback (see Section 3.1.1 [Sanderson, 1997, Cooper et al., 2002]). For large collections, where a common term can occur in millions of documents, this is not scalable. For term selection approaches this cost is significantly less, but still requires at least the same number of I/O operations as the number of terms selected via the relevance feedback algorithm. Kolcz et. al. proposed an optimization to the more typical cosine similarity measure. In this work, it was assumed that terms that occurred in more than five percent of the collection actually occurred in each document. This optimization produced results within ninety percent of the full cosine similarity measure, but executed an order of magnitude faster [Kolcz et al., 2004].

5.4.2.4 I-Match

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. I-Match 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 [Grossman et al., 1995, Smeaton et al., 1997]. 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 (idf), retain only significant tokens.
- Insert relevant tokens into unicode ascending ordered tree of unique tokens.
- Loop through token tree adding each unique token to the SHA1 [4] digest. Upon completion of loop, a (doc_id, SHA1 digest) tuple is defined.
- The tuple (doc_id, SHA1 digest) is inserted into the storage data structure using the key.
- If there is a collision of digest values then the documents are similar.

The overall runtime of the I-Match approach is $(O(d \log d))$ in the worst case where all documents are duplicates of each other and $(O(d))$ otherwise, where d is the number of documents in the collection. All similar documents must be grouped together. That is, the corresponding document identifiers must be stored as a group. In the most extreme case, all of the hash values are the same (all the documents are similar to each other). In such a case, to store all the document identifiers together in a data structure (tree) requires $(O(d \log d))$ time. Typically, however, the processing time of the I-Match approach is $O(d)$ time. The calculation of idf values can be approached with either of two methods. The first is with the use of a training collection to produce a set of terms idf tuples before the de-duplication work occurs. It was shown that term idf weights change slightly as collection sizes grow so this is an acceptable solution [Frieder et al., 2000a].

A second approach is to run two passes over the documents, where the first pass calculates the idf weights of the terms, and the second pass finds duplicates with the I-Match algorithm. This approach increases the actual run time of the algorithm, but the theoretical complexity would remain unchanged. Conrad et. al. examined these approaches when using a dynamic collection and only using high idf terms and found the approach not stable if a dynamic vocabulary is used [Conrad et al., 2003]. Recently, they developed a new test collection for inexact duplicate document detection [Conrad and Schriber, 2004]. This suggests that the first approach may be the more applicable for this problem.

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. Kolcz et. al. showed that this randomization approach improved the recall effectiveness of I-Match by 40-60 percent without hurting precision, when using 10 random lexicons [Kolcz et al., 2004].

5.5 Summary

Performance evaluation considerations of information retrieval systems involve both effectiveness (accuracy) and efficiency (run-time and storage overhead) measures. In this chapter, we focused on the efficiency considerations.

Initially, we described the concept of and motivation for the use of an inverted index. An inverted index is a many-to-many mapping of terms onto documents. Using an inverted index, only documents that contain the specified query terms are accessed, thus significantly reducing the I/O requirements as compared to other search processing structures. Having described the concept of an inverted index, we continued by illustrating a method to implement an inverted index and a pruned variation. We also outlined various techniques for compressing the index. Two compression techniques were reviewed. The first, fixed length compression, has the advantage of simplicity and slightly more efficient query processing times as compared to the second, variable length compression. Variable length compression, however, does result in a slightly better compression ratio.

We concluded the chapter with an overview of signature files. Signature files contain a set of document signatures, one signature per document. A document signature is an encoding of each document. Key terms contained in the document are hashed onto a vector; the existence of a term j in the document i is denoted by a one in the $j^{\underline{th}}$ bit of document signature i. To determine which documents are relevant to a particular query, only the signature file must be examined. Since term hashing can result in false positive indications, a two pass search strategy is necessary. In the first pass, involving the examination of the signature file, all potential candidates are determined. In the second phase, a full text scan of the potential candidates determined in the first pass is performed.

Although greater attention has traditionally been placed on the effectiveness of information retrieval systems, efficiency issues are critical. Failure to optimize the efficiency of an information retrieval system can result in a highly accurate system that suffers prohibitive execution or storage performance. As storage technology continues to improve and decrease in cost, storage constraints are becoming less critical. However, with the continued exponential growth of online data, storage constraints are still a concern and run-time performance considerations are of paramount importance. Parallel processing techniques used to improve the overall run-time performance are described in Chapter 7.

5.6 Exercises

1 Write a utility called *index* that builds an inverted index of *Alice in Wonderland.* Assume ten lines of input is a separate document. Assume you have enough memory to store all of the posting lists in memory while you are building the inverted index. Identify how much space your index requires and how long it takes to build it. Store the *idf* for each term in the index. Each posting list entry should contain the term frequency in the document. Use the 100 most frequent terms as stop terms. Test your index by computing a vector space *tf-idf* similarity measure for the following five queries.

- *rabbit watch*
- *looking glass*
- *tea party*
- *cheshire cat*
- *queen of hearts*

2 Now modify the code you just wrote to use an inverted index compression technique. Pick one in this chapter. Measure query performance for the same five queries, storage overhead, and the time to build the index. Repeat this, an now use a pruned index.

3 Pick a query that contains ten terms. Execute it and retrieve the top documents choosing ten that are relevant. Now, sort the query terms by their term frequency across the collection. Re-execute the query with one term—the least frequently occurring term in the collection. Identify the number of relevant documents found with just this term. Repeat this process, adding a single term to the query each time. Are all ten terms needed to find the relevant documents you found with the original query? Talk about what you have learned with this exercise and how this technique could be used to improve run-time without a corresponding loss in accuracy.

4 Develop a signature-based index where you build a signature for each "document" in the book. Use a 24-bit signature for each document. Now implement the ten queries used in the previous exercises as a simple Boolean OR. Compare run-time performance of the use of signatures to the inverted index. Describe the loss in functionality inherent in the use of signatures. Identify a heuristic in which signatures could be used as a "first-pass filter" for a very large collection and then describe how an inverted index could be used for detailed analysis.

Chapter 6

INTEGRATING STRUCTURED DATA AND TEXT

Essential problems associated with searching and retrieving relevant documents were discussed in the preceding chapters. However, simply searching massive quantities of unstructured data is not sufficient.

Terabytes of structured data currently exist. NCR recently demonstrated the use of its database system on a 300 terabyte database [Holmes, 2004]. It is reasonable to expect databases to soon grow into the petabyte range. The study of database management systems (DBMS) focuses on the algorithms necessary to support thousands of concurrent users adding, deleting, updating, and retrieving structured data.

It is difficult to formally characterize *structured data*. Structured data are data that have a certain repetitive nature—data that fit within an easily recognizable datatype. Examples of structured data include *name, address, phone number, and salary*. Each occurrence of a structured data item is recognizable, sometimes it is possible to list only a few valid values for a structured data element (i.e., gender has only two valid values—*male* or *female*).

Airline reservation systems, automated teller machines, and credit card validation devices are all systems that pervade everyday life. Each is replete with structured data. One large production structured database has 300 terabytes and uses 1,1016 processors [Holmes, 2004].

There is clearly a need to integrate both structured data and text. Most production systems implemented with a relational database management system (RDBMS) have some text—such as a *comment* field—which allows users to enter a free text comment about a particular order. Commercial database systems allow users to store these unstructured fields in Binary Large OBject (BLOB) or Character Large OBjects (CLOB) datatypes unstructured data to be stored in a relation. The problem is that these unstructured fields cannot be accessed efficiently. Access methods such as inverted indexes found

in information retrieval systems are lacking, and when they do exist, they are implemented in a non-standard fashion.

Similarly, information retrieval systems typically have large quantities of structured information, (i.e., author of a document, publication date, etc.) and usually have the ability to store data in *zoned* fields. These fields have a particular start and stop delimiter that identifies a *zone* in a document. The problem is that these structured fields cannot be accessed efficiently. Access methods for structured data (e.g., B-trees) and query optimization techniques that determine the best access method to the data are not usually found in information retrieval systems.

A database management system (DBMS) and an information retrieval system are analogous to a martial artist who is trained to fight others who are trained in the same art. A Tai Kwon Do master is capable of defending against other Tai Kwon Do masters. An information retrieval system is capable of efficiently handling unstructured data. A Judo master is capable of defending against other Judo masters. A database management system is capable of efficiently handling structured data. The problem arises when the Tai Kwon Do master faces a Judo master. This is analogous to accessing unstructured data in a structured database system.

The approach described in the remainder of this chapter is to build some unstructured data handling techniques on top of an existing relational database management system. This is analogous to teaching the Judo master some Tai Kwon Do techniques, but doing so in a way that still relies upon Judo.

It is possible to start with a database system and extend it to handle unstructured data or to start with an unstructured system and extend it to handle structured data. The approach taken in this chapter is to extend the database system. Information retrieval is then treated as an application of the database system (see Figure 6). The reason for this is that relational database systems, over the years, have developed substantially more infrastructure than information retrieval systems. Hence, to solve the integration problem, a straightforward approach is to start with an existing database system and add the necessary information retrieval functionality. In addition to providing integration, two additional benefits are obtained: parallel processing and dynamic updates.
Parallel processing takes advantage of multiple processors to improve run-time performance.

In Chapter 7, several parallel information retrieval algorithms are described. Although these algorithms do improve performance, none of them have shown particularly good speedup, that is, when additional processors are added they are not fully used. However, most major database vendors (i.e., IBM, Sybase, Oracle, Microsoft) all have parallel solutions. Some database vendors specialize in special-purpose parallel hardware that implements a proprietary database

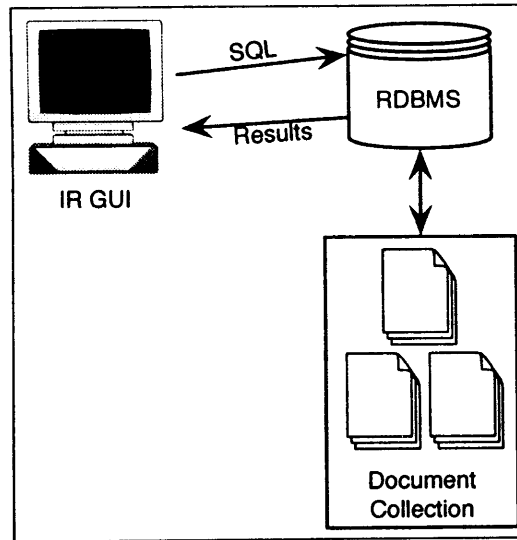


Figure 6.1. IR as an Application of a RDBMS

system (the NCR Teradata). Relational, set-theoretic operators are intrinsically unordered, and it is this lack of order that makes it easier to implement parallel operations. Treating information retrieval as a database application is intrinsically a parallel information retrieval algorithm because the underlying DBMS can be parallelized.

A second advantage of treating information retrieval as an application of a relational database management system (RDBMS) is that document data can be easily updated. Most information retrieval systems have a lengthy preprocessing phase in which the inverted index is constructed. To add, modify, or delete an existing document usually requires a process in which the inverted index is modified. Most information retrieval systems do not support on-line modifications to a document. A RDBMS has substantial infrastructure (concurrency control and recovery management) to ensure that updates may be done in real-time, and if an error occurs in the middle of an update, pieces of the update are not partially stored in the database systems.

A key question remains: Which database model should we use and how should the information retrieval functionality be added? Database system models include the inverted list, hierarchical, network, relational, and object-oriented models [Date, 1994]. Most current commercial systems rely upon the relational model. Although it is interesting to contemplate whether or not another model would be better suited for unstructured data, pragmatism dictates the use of the relational model. Sales of RDBMS software reached $7.1 billion in

the past year. Using a different data model to obtain integration would mean that countless sites would have to convert their existing DBMS to a new model. The cost for this would be astronomical.

To gauge how long relational systems will dominate the market, it is useful to look at their predecessors. IMS, a hierarchical system, and IDMS, a network system, dominated the market in the 1970's. By 1980, both were well established. At that time, Oracle, the first relational vendor, was founded. Relational systems had been advocated heavily in the research community during the mid-1970's with substantial work having been done with a full-fledged prototype named System R.

IBM introduced its first commercial relational system, SQL/DS, in 1984 and its successor, DB2, in 1986. Relational systems did not gain significant market share until the early 1990's, a full ten years after Oracle was founded. We first use the relational model, and we later discuss the use of more recently developed multi-dimensional database systems.

The final question is: How should we use the relational model? Two choices exist: extend the relational engine or treat information retrieval as an application of an RDBMS.

Section 6.2 reviews prior attempts to extend the relational model. The main problem with these attempts is that they are all non-standard. Portability is lost because each relational extension is somewhat different, and users are not able to move applications from one system to another. Other problems are that query optimization must be modified to support any additions to the engine. Additionally, adding new functionality to the engine makes an already complex engine even more complex. Some additions allow users to add functions to the engine. This makes integrity an issue as a malicious or negligent user can intentionally or unintentionally introduce bugs into the database engine. Finally, parallel algorithms must be developed for each addition.

By treating information retrieval as an application of a RDBMS, these problems are eliminated. The key concern is to develop efficient unchanged Structured Query Language (SQL) algorithms that adhere to the ANSI SQL-2003 standard [SQL, 2003] for each type of information retrieval functionality. This chapter describes relational approaches for the following information retrieval functionality:

- Boolean keyword search
- Proximity search
- Relevance ranking with terms
- Relevance feedback

Relevance ranking with Spanish, phrases, passages, n-grams, and relevance feedback have all been implemented as an application of a relational DBMS with standard, unchanged SQL by using straightforward modifications to the approaches described in this chapter. Details are found in [Lundquist et al., 1997, Grossman et al., 1997].

In Section 6.1, we briefly review the relational model and SQL. A historical progression of integrating database technology with information retrieval functionality is provided in Section 6.2. In Section 6.3, we describe the algorithms used to treat the previously described information retrieval strategies and utilities together as an application of a relational system. Only standard SQL is used.

Next, in Section 6.4, we describe a method to support semi-structured data search. Here, we provide a description of how a fully featured XML retrieval engine can be built, once again, as an application of a relational database system. Thus, as a whole, we demonstrate the integration, all via standard relational database techniques, of structured, semi-structured, and text data.

Continuing with Section 6.5, we describe the use of a multi-dimensional data model to support the integration of hierarchically structured data and text. Using this approach, naturally occurring hierarchies can be supported directly by the integrating fabric.

Finally, in the section on mediators, Section 6.6, we review a method to integrate disparate data stored geographically across multiple domains that supports a question-answer paradigm. As always, a summary and our future projections conclude this chapter.

6.1 Review of the Relational Model

The relational model was initially described by Codd [Codd, 1970]. Prior data models were navigational, in that application developers had to indicate the means by which the database should be traversed. They specifically described how to find the data. The relational model stores data in relations and enables the developer to simply describe *what* data are required, not *how* to obtain the data. During the early 1970's, relational systems were not developed as they incur additional computational and storage overhead. Over the years, algorithms were developed to improve query optimization. These algorithms reduce the amount of overhead expended when using a relational system.

Over time, the benefits of the relational model have outmatched the costs, and the relational model is now the centerpiece of most production database systems. For some extremely high-performance applications, navigational systems are used, but relational systems have prevailed.

Table 6.1. Employee (EMP)

emp_no	*emp_name*	*age*	*salary*
100	Hank	35	$10,000
200	Fred	40	$20,000
300	Mary	25	$30,000
400	Sue	23	$40,000
500	Mike	30	$50,000

6.1.1 Relational Database, Primitives and Nomenclature

A relational database system stores data in set-theoretic relations. An attribute within a relation is any symbol from a finite set $\mathcal{L} = \{A_0, A_1, A_2, \ldots, A_n\}$. A relation $\mathcal{R}$ on the set $\mathcal{L}$ is a subset of the Cartesian product dom(A_0)$\times$ dom(A_1) $\times$ dom(A_2) $\times$ ... dom(A_n) where dom(A_i) is the domain of A_i. R[$A_0 A_1 A_2 \ldots A_n$] represents $\mathcal{R}$ on the set $\{A_0, A_1, A_2, \ldots, A_n\}$ and is referred to as the schema of $\mathcal{R}$. In R[$A_0 A_1 A_2 \ldots A_n$], each column A_i is called an *attribute* of R, and is denoted as $R.A_i$.

Simply stated, each attribute contains values, preferably a singular value, chosen from a given domain of values. An attribute *color* can have a domain of *red, green, black, etc.* A relation is then a collection of attributes. A row, or *tuple*, in the relation has a value for each attribute such that the value comes from the domain for that attribute.

Each tuple of R is designated by $< a_0, a_1, a_2, \ldots, a_n >$, where $a_i \in$ dom(A_i). The value of attribute A_i of tuple $x \in R$ is denoted as x[A_i]. Similarly, if tuple $x \in R$, then x[W] is the value of the attributes of attribute set W in tuple x.

Table 6.2. Employee-Project (EMP_PROJ)

emp_no	*project*
100	A
100	B
100	C
200	B
300	A
300	C
400	A

Consider the relations EMP and EMP-PROJ (see Tables 6.1 and 6.2). Relation EMP has four attributes (*emp_no, emp_name, age, salary*) while the EMP-PROJ relation has two attributes (*emp_no, project*). The EMP relation contains a tuple for each employee in the organization indicating the employee's unique identification number, name, age, and salary. An employee can also be as-

signed to an arbitrary number of projects. Simply adding a *project* attribute to the EMP relation would not work since it would only hold a single value. Another solution—adding *project1, project2, project3* attributes is also inadequate because an employee may have worked on more than three projects. In this case, there would be no place to store the 4th to n^{th} project.

Data models primarily differ in how they handle this type of *multi-valued relationship*. This is referred to as a MANY-MANY relationship in that one employee can be assigned to many projects while a project can be assigned to many employees. In a navigational model, a pointer points from the EMP master record with all single-valued occurrences to a list that contains the multi-valued occurrences. A user who wishes to see which projects an employee is assigned to issues a request to traverse the link from the master record to the multi-valued list.

Additional relations are developed for the relational solution. In our case, a single relation EMP-PROJ can be added to store the multi-valued information. Notice that EMP-PROJ has an attribute *emp_no* that matches values in the EMP relation. Hence, employee number 100 works on projects A, B, and C. The key point is that no *a priori* link between EMP and EMP_PROJ exists. At query time, a user may request that all tuples having matching values in the two relations be obtained. In this fashion, the user has only specified *what* is required not *how* to obtain the data.

This is important since requests for data may occur on an ad hoc basis long after the database has been created and populated with data. The relational model is well-suited to ad hoc requests because work is not required to redefine relationships between the data. Additionally, data independence is intended to reduce application development time because developers are not forced to learn all of the intricacies of retrieving data from multi-valued relationships. The database optimizer makes decisions and chooses the best access path to the data.

A problem exists if it is necessary to track single-valued information about a project such as the delivery date for the completed project or the budget for the project. If the EMP-PROJ relation is modified to include these additional attributes, needless repetition occurs (see Table 6.3).

Notice the attributes *delivery_date* and *budget* are single-valued descriptors of a project (all dates are assumed to be represented as Julian dates, and hence, are single-valued descriptors). These are repeated for each employee who is working on a project. Employees 100, 300, and 400 all work on Project A, and the *delivery_date* and *budget* are replicated for each of these tuples. If an update was required (i.e., the budget increased), it would be necessary to update each occurrence. To avoid these problems, a third relation is typically used to store the single-valued data for PROJECT. It would appear as given (see Table 6.4).

Table 6.3. Employee-Project (EMP_PROJ)

emp_no	*project*	*delivery_date*	*budget*
100	A	06/30/1997	$90,000,000
100	B	09/15/1997	$25,000,000
100	C	03/31/1998	$60,000,000
200	B	09/15/1997	$25,000,000
300	A	06/30/1997	$90,000,000
300	C	03/31/1998	$60,000,000
400	A	06/30/1997	$90,000,000

Table 6.4. Project

project	*delivery_date*	Budget
A	06/30/1997	$90,000,000
B	09/15/1997	$25,000,000
C	03/31/1998	$60,000,000

At this point three relations exist, one to represent the EMP entity, one to represent the PROJECT entity, and one to represent the relation EMP_PROJ that exists between the two relations. It should be clear that an update to single-valued information about a project only involves a single tuple.

Peter Chen, in a seminal paper, described the entity-relationship (ER) diagram in which entities and relationships are defined first, and the actual underlying relations are subsequently defined [Chen, 1976]. Typically, for large relational systems, an ER diagram is developed to ensure the developers understand all of the relationships between the data. Once complete, a normalized database design is implemented.

Normalization is the process of ensuring the database design satisfies very specific rules developed to reinforce the consistency and integrity of the data. First Normal Form (1NF) simply indicates that data are stored in single-valued attributes. Our example relation, EMP, is clearly in 1NF. However, if the *name* attribute were expanded to allow the employee's full first and last name in the same attribute, this entity would no longer be in 1NF because the *name* attribute would permit the values for both the first and last names to coexist in a single data element.

A relation is in second normal form (2NF) if all attributes are fully dependent on the primary key of the relation. Our example of the modified EMP_PROJ relation is not in 2NF because the attributes of the relation *delivery_date* and *budget* are not fully dependent on the composite primary key of *emp_number* and *project*. Instead, *delivery_date* and *budget* are dependent solely on the *project* attribute. An entity is in third normal form (3NF) if all

attributes of the entity are dependent on the primary key of the relation and are not also dependent on another key. The *primary key* is one or more attributes that uniquely identifies a tuple in a relation. A database should satisfy at least 3NF.

It should be clear that no *a priori* linkages exist between any of the relationships, and any linking of relations is done at query processing time rather than data definition time.

Since the relations are based on set theory, all typical set-theoretic operations: Cartesian product, union, intersection, and set difference are implemented in the relational model. Additional operations include:

Select—The selection on R[XYZ], denoted as $\sigma_{A=a}(R)$, is defined by:

$$\sigma_{A=a}(R) = \{x | x[A] = a, x \in R\}$$

where A is an attribute of R.

Project— The projection on R[XYZ], denoted as $\pi_A(R)$, is defined by:

$$\pi_A(R) = \{x[A] | x \in R\}$$

where A is a set of attributes of R.

Join— The join of two relations R[XYZ] and S[VWX] (sharing the common attribute X) is denoted as:

$$R[XYZ] \bowtie S[VWX] = \{x \mid x[VWX] \in S \text{ and } x[XYZ] \in R\}$$

where V, W, X, Y, and Z are a disjoint set of attributes. If no common attribute exists, the join of R and S is the Cartesian product of R and S.

When the relational model was first proposed nearly thirty-five years ago, relational algebra and calculus were used to compute data manipulation. The select, project, and join operators form a part of relational algebra. Since this was not very user friendly, two different query languages QUEL and SQL (originally derived from SEQUEL) were developed. SQL became popular with IBM's adoption in its commercial database system, SQL/DS, in 1982 and with ANSI's adoption of the first SQL standard in 1985. Today, SQL is one of the few standards that is agreed upon by industry, academia, and various international standards committees. SQL-2003 was recently adopted [SQL, 2003].

A good overview of SQL can be found in [Date, 1994]. A SQL query has the structure:

```
SELECT <list of attributes>
   FROM <list of relations>
[ WHERE <list of conditions> ]
[ ORDER BY <list of attributes> ]
[ GROUP BY <list of attributes> ]
[ HAVING <list of conditions> ]
```

A list of attributes is specified after the SELECT keyword. The FROM clause indicates the relations that are used. The WHERE clause describes conditions that must be satisfied for a tuple to be returned. Hence, the entire query is actually a *specification of a result*. The following query indicates that only the employee numbers from the EMP table should be retrieved. It does not, in any form, indicate how the employee numbers should be retrieved. Another form includes the addition of a WHERE clause.

```
SELECT emp_no
   FROM EMP
```

The following query indicates that only tuples with an *emp_no* of 400 are to be retrieved. Nothing is indicated as to how to find this tuple. If the system has a B-tree index on the *emp_no* attribute, an $O(\log n)$ algorithm traverses the tree and finds all such tuples, otherwise, a linear scan is used. In any event, the author of the query does not specify the algorithm to use to retrieve these data.

```
SELECT emp_no
   FROM EMP
   WHERE emp_no = 400
```

GROUP BY is used to partition the result set into groups and apply an aggregate function to the group. Aggregate functions in the SQL standard include COUNT (size of the partition), SUM (the total of an attribute in the partition), MIN (the smallest value in the partition), MAX (highest value in the partition), and AVG (average of all values in the partition). If a GROUP BY is not present, these operators work on the entire result set.

Consider a request to develop a report that contains each employee's number and the total number of projects to which they have been assigned. The following query obtains this information:

```
SELECT emp_no, COUNT(*)
   FROM EMP_PROJ
   GROUP BY emp_no
```

Grouping by the employee number partitions the EMP_PROJ relation into a partition for each employee. COUNT returns zero if no tuples are found. If a WHERE clause existed it would specify that the partitions should consider only the tuples identified by the WHERE clause.

HAVING restricts groups, typically based on an aggregate. The following query finds all employees who worked on at least 4 projects:

```
SELECT emp_no, COUNT(*)
   FROM EMP_PROJ
   GROUP BY emp_no
   HAVING COUNT(*) > 3
```

ORDER BY is used to sort the tuples in the order of the attributes specified in the ORDER BY clause. Since sets do not have any inherent ordering, the result set of a query may be obtained in an arbitrary order unless the ORDER BY clause is used. Executing this query results in a list comprising all employee numbers in ascending order (the DESC option must be used to obtain descending order).

```
SELECT emp_no
   FROM EMP_PROJ
   ORDER BY emp_no
```

A JOIN is implemented by first specifying multiple relations in the FROM clause and then adding the JOIN condition in the WHERE clause. The following query implements a join to find the age of all employees who worked on project A.

```
SELECT a.emp_no, a.age
   FROM EMP a, EMP_PROJ b
   WHERE a.emp_no = b.emp_no AND
            b.project = 'A'
```

This query joins the two relations. Again nothing is said about the join order or the order in which the WHERE clause is executed.

6.2 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. We now describe each of these approaches in detail.

6.2.1 Combining Separate Systems

Several researchers proposed integrated solutions which consist of writing a central layer of software to send requests to underlying DBMS and information retrieval systems [Schek and Pistor, 1982]. 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. It does not take long to build this software, and since information retrieval systems and DBMS are readily available, this is often seen as an attractive solution.

The key advantage of this approach is that the DBMS and information retrieval system are commercial products that are continuously improved upon by vendors. Additionally, software development costs are minimized. The disadvantages include poor data integrity, portability, and run-time performance.

6.2.1.1 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.

6.2.1.2 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.

6.2.1.3 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. The difficulties with such optimization were discussed by the authors who suggested this approach [Lynch and Stonebraker, 1988]. Hence, run-time performance would suffer due to a lack of parallel algorithms and limited global query optimization.

6.2.1.4 Extensions to SQL

Blair, in an unpublished paper in 1974, proposed that SQL (actually a precursor named SEQUEL) could be modified to support text [Blair, 1974]. Subsequently, a series of papers between 1978 and 1981 were written that described several extensions to SQL [Macleod, 1978, Macleod, 1979, Crawford, 1981]. The SMART information retrieval prototype initially developed in the 1980's used the INGRES relational database system to store its data [Fox, 1983b].

These papers described extensions to support relevance ranking as well as Boolean searches. The authors focused on the problem of efficiently searching text in a RDBMS. They went on to indicate that the 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. The authors mentioned extensions, such as RELEVANCE(*), 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 [Ozkarahan, 1995]. SQLX assumes that an initial cluster-based search has been performed based on keywords (see Section 3.2 for a description of document clustering). SQLX extensions allow for a search of

the results with special connector attributes that obviate the need to explicitly specify joins.

6.2.2 User-defined Operators

User-defined operators that allow users to modify SQL by adding their own functions to the DBMS engine were described as early as [Stonebraker et al., 1983]. Commercialization of this idea has given rise to several products including the Teradata Multimedia Object Manager, Oracle Cartridges, IBM DB2 Text Extender, as well as features in Microsoft SQL Server [Connell et al., 1996, Loney, 1997]. 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 [Lynch and Stonebraker, 1988].
The following query obtains all documents that contain the terms *term1*, *term2*, and *term3*:

Ex: 2
```
SELECT Doc_Id
  FROM DOC
  WHERE SEARCH-TERM(Text, Term1, 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 Doc_Id
  FROM DOC
  WHERE PROXIMITY(Text, 5, Term1, 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.

6.2.2.1 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.

6.2.2.2 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.

6.2.2.3 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.

Without 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.

Lynch's work discussed information that must be stored with each user-defined operator to assist with query optimization. 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.

6.2.3 Non-first Normal Form Approaches

Non-first normal form (NFN) approaches have also been proposed [Desai et al., 1987, Schek and Pistor, 1982, Niemi and Jarvelin, 1995]. 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 approach was shown to support integrity and portability while still yielding acceptable run-time performance [Grossman et al., 1997].

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.

6.2.4 Bibliographic Search with Unchanged SQL

Blair explored the potential of relational systems to provide typical information retrieval functionality [Blair, 1988]. Blair's work 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, DocId*)—indicates who wrote a particular document.
- INDEX(*term, DocId*)—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 = input_term
            ORDER BY 2 DESCENDING)
```

Blair cites 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

6.3 Information Retrieval as a Relational Application

Work with extensions to SQL started first in an unpublished paper [Blair, 1974] and continued with several papers by Macleod and Crawford between 1978 and 1981 [Macleod, 1978, Crawford, 1981].

Initial extensions described by Macleod 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 ≠ s.term
```

Finally, to implement a logical AND of the terms *InputTerm1, InputTerm2*, and *InputTerm3*, Macleod and Crawford proposed the following query:

Ex: 7
```
SELECT DocId
  FROM INDEX
 WHERE term = InputTerm1
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.

Macleod and Crawford went on to present extensions for relevance ranking. The key extension 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 CORR() > 60
```

Other extensions, such as the ability to obtain the first n tuples in the answer set, were given. Macleod and Crawford gave detailed design examples as to how a document retrieval system should be treated as a database application.

We now describe work that relies on the unchanged relational model to implement information retrieval functionality with standard SQL [Grossman et al., 1997]. First, a discussion of preprocessing text into files for loading into a relational DBMS is required.

6.3.1 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 [Goldfarb, 1990]. 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. A variety of experiments designed to identify the most efficient means of implementing the preprocessor are given in [Pulley, 1994].

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). These were described in Section 2.1. 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.

6.3.2 A Working Example

Throughout this chapter, the following working example is used. 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 Tables 6.5 and 6.6.

Table 6.5. 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 6.6. 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 (see Section 5.1 for a brief overview of inverted indexes). 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. This inverted file technique is so effective that it was used in some of the earliest structured systems in the mid-1960's such as TDBMS [Bleir, 1967].

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 $n^{\underline{th}}$ is given an offset of $n-1$. The INDEX_PROX relation given in Table 6.7 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 6.7. 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 (see Table 6.8) 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 6.8. 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
...	...

To 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 (see Table 6.9) 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 (see Table 6.10) is used to indicate all of the terms that are omitted during the parsing phase. This relation is not used in this chapter, but illustrates that the relational model can store internal structures that are used during data definition and population.

Table 6.9. QUERY

Term	tf
vehicle	1
sales	1

Table 6.10. 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 6.1.1 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.DocId
  FROM DOC d, INDEX i
  WHERE i.Term IN ("vehicle", "sales") AND
        d.PubDate = "3/23/87" AND
        d.DocId = i.DocId
```

6.3.3 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)
```

For the remainder of the section, we are only concerned with obtaining the document identifiers found in the answer set. Either a separate query may be executed using the document identifiers in an application specific fashion or the queries can be extended in the form given in Example 13.

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 = InputTerm1 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 = InputTerm_1 AND
       b.Term = InputTerm_2 AND
       c.Term = InputTerm_3 AND
       ...
       n.Term = InputTerm_n AND
       a.DocId = b.DocId AND
       b.DocId = c.DocId AND
       ...
       n - 1.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 [Elmasri and Navathe, 1994]. Pruning the list is discussed in [Selinger, 1979], but it is still 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 implementions impose limits on the number of joins (around sixteen is common) [White and Date, 1989, McNally, 1997]. 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 is given in [Grossman et al., 1997]. This reduces the number of conditions found 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 HAVING 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 $(t_1, t_2, \ldots, t_k)$, the tuples as given in Table 6.11 are generated for document d_i containing all k terms.

Table 6.11. 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 d_i 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
```

6.3.4 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 INDEX_PROX 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 INDEX_PROX a, INDEX_PROX 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 d_i that contains k terms $(t_1, t_2, \ldots, t_k)$ in the corresponding term offsets of $(o_1, o_2, \ldots, o_k)$, 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 Table 6.12, we illustrate the logic of the query. Drawing out the first step of the join of INDEX_PROX to itself for an arbitrary document d_i yields tuples in which each term in INDEX_TERM is matched with all other terms. This table shows only those terms within document d_i that matched with other terms in document d_i. 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 6.12. 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 d_i 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 6.13).

Table 6.13. 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

6.3.5 Computing Relevance Using Unchanged SQL

Relevance ranking is critical for large document collections as a Boolean query frequently returns many thousands of documents. Recent World Wide Web search engines such as *Google* and *Yahoo!*, as well as commercial information retrieval systems such as Convera's *RetrievalWare* and Verity's *Topic*, all implement relevance ranking. Numerous algorithms exist to compute a measure of similarity between a query and a document. We have discussed many of these variations in Chapter 2.

As we previously mentioned in Section 2.1, 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 (see Section 2.1). The distance between the query vector Q and the document vector D_i 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 w_{qj} is weight of the jth term in the query q, and d_{ij} is the weight of the jth term in the ith 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 (see Section 2.2.1). 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. Many variations of this coefficient are found in the literature [Salton, 1989]. 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 QRY_WT
    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, DOC_WT 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_WT and QRY_WT. 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 $a \leq b \Rightarrow a^2 \leq b^2$ for $a, b \geq 0$. 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.

SQL for the pivoted normalization measure described in Section 2.1.2 and for the probabilistic measure described in Section 2.2.3 is given in [McCabe et al., 1999]. 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. We will describe the use of a combination of similarity measures in more detail in Section 8.3.

6.3.6 Relevance Feedback in the Relational Model

Relevance feedback can be supported using the relational model [Lundquist et al., 1997]. 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 (see Section 3.1). In this work, 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.

6.3.7 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 [Frieder et al., 2000b, Frieder et al., 2003]. 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 SQL/DS, and even mySQL, system portability was demonstrated. Efficiency was enhanced using several optimization approaches including some described earlier (see Chapter 5) 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. Additional implementation details, including a query-processing framework that supported query and result caching, are found in [Frieder et al., 2000b].

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 on [Lundquist et al., 1999]. A later effort at the University of Tokyo [Goda et al., 2001] demonstrated further scalability using the SIRE approach on a 100+ node PC cluster. At ETH-Zurich, researchers showed that the SIRE approach can be used to improve throughput for document insertion and update as well as simple retrieval [Grabs et al., 2001].

6.4 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) [Schmidt et al., 2000, Shanmugasundaram et al., 1999]. An approach which uses a static relational schema was described in [Florescu and Kossman, 1999] and additional support for a full implementation of an XML query language XML-QL is also described [Deutsch et al., 1999]. More recently, algorithms that translate XQuery expressions to SQL were presented [DeHaan et al., 2003].

6.4.1 Background

XML has become the standard for platform-independent data exchange [Buneman et al., 1996, Goldman et al., 1999]. There were a variety of methods proposed for storing XML data and accessing them efficiently [Abiteboul, 1997].

One approach is a customized tree file structure, but this lacks portability and does not leverage existing database technology [Kanne and Moerkotte, 2000]. Other approaches include building a database system specifically tailored to storing semi-structured data from the ground up [McHugh et al., 1997, Quass et al., 1996] or using a full inverted index [Shin, 1998].

There are several popular XML query languages [Luk et al., 2002, Deutsch et al., 1999, Bonifati and Ceri, 2000]. In August 1997, initial work on XPath, touted as a basic path-based query language, was submitted to the W3C (World Wide Web Consortium). In 1998, XML-QL, a query language developed at AT&T [Deutsch et al., 1999], was designed to meet the requirements of a full-featured XML query language set out by the W3C. The specification describing XPath as it is known today was released in 1999. In December 2001, XPath 2.0 was released. One of the newest semi-structured query languages, XQuery, is also among the most powerful. It borrows many ideas from prior work on other semi-structured query languages such as XML-QL and XPath, as well as from relational query languages like SQL. The first public draft of the XQuery 1.0 specification was released in June 2001 and has a current update as of May 2003. [Boag et al., 2003].

6.4.2 Static Relational Schema to support XML-QL

We now briefly describe a static relational schema that supports arbitrary XML schemas. This was first proposed in [Florescu and Kossman, 1999] to provide support for XML query processing. Later, in the IIT Information Retrieval Laboratory (**www.ir.iit.edu**), it was 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 described in [Florescu and Kossman, 1999], 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 (<u>*TagId*</u>, *tag*) ATTRIBUTE (<u>*AttributeId*</u>, *attribute*)
TAG_PATH (<u>*TagId*</u>, *path*) DOCUMENT (<u>*DocId*</u>, *fileName*)
INDEX (<u>*Id, parent, path, type, tagId, attrId, pos*</u>, *value*)

For the remainder of this section, consider once again our sample text.

<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>

6.4.3 Storing XML Metadata

These tables store the metadata (data about the data) of the XML files. TAG_NAME (see Table 6.14) and TAG_PATH (see Table 6.15) 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 (see Table 6.16) 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 *tagId* 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.

6.4.4 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, it

Table 6.14. TAG_NAME

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

Table 6.15. TAG_PATH

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

Table 6.16. ATTRIBUTE

AttributeId	attribute
7	LANGUAGE

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. An example of this relation is shown in Table 6.17. Other relevant attributes might include the length of the document – or the normalized length [Kamps et al., 2004].

Table 6.17. DOCUMENT

docId	fileName
2	doc_0.xml
3	doc_1.xml

6.4.5 INDEX

The INDEX table (see Table 6.18) 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 *TagId* and *AttrId* is a foreign key to the *TagId* 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.

Table 6.18. INDEX

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

6.5 Multi-dimensional Data Model

In the preceding two sections, we described methods to use the relational model as the core storage and retrieval components for an information retrieval system or an XML-IR system. It is also possible to use a multi-dimensional data model to provide similar functionality. Such a model inherently supports hierarchical dimensions and is well suited for natural hierarchies that occur in some documents. A DOCUMENT dimension might include the hierarchy of a document (e.g.; document, chapter, section, paragraph, sentence). Two additional dimensions are LOCATION and TIME. The location of a document and the publication date of a document are both common in information retrieval applications. The corresponding hierarchies of LOCATION (e.g.; country, region, state, city) and TIME (e.g.; year, month, day) are well suited to multi-dimensional modeling.

Processing of data in a multi-dimensional data model is referred to as OLAP (On-Line Analytical Processing). ROLAP (Relational OLAP) refers to a multi-dimensional model that is implemented with a relational database system. MOLAP refers to a multi-dimensional model that is implemented with a multi-dimensional database system. For our discussion it is not important whether or not ROLAP or MOLAP is used. The key is that we are now able to more easily represent hierarchical dimensions that naturally occur in text.

Typically, a star schema is used for OLAP applications. Our dimensions: DOCUMENT, TIME, and LOCATION are arranged in a star around a single *fact* table. The fact table is analogous to the INDEX relation we described in Section 6.3. This table maps a given term to its corresponding dimensions of DOCUMENT, TIME, and LOCATION.

Once data are represented with a star schema, it can then be migrated to a ROLAP or MOLAP system. These multi-dimensional systems offer the added advantage that they more naturally represent hierarchical data often found in a document collection. Relevance ranking using a MOLAP system is described in [McCabe et al., 2000].

6.6 Mediators

At this point, we have described a means of integrating structured data, semi-structured data and text via the relational model. This approach is able to harness the power of existing relational systems and provides a means by which data can be easily integrated when it is practical to store it in a centralized repository.

The reality is that some applications require the storage of data in disparate locations. For text collections in the multi-terabyte range that already have search engines that access them, it is not realistic to expect that a redundant copy of these collections will be made in a relational system. Instead, a *me-*

diator that resides between a user and the data determines which data sources are most relevant to query, submits the query to those engines that search the determined relevant sources, and then consolidate the results. If all the sources are text, this becomes a relatively straightforward metasearch. Metasearch is simply a search of a set of search engines. If some sources are text, others are XML, and still others are relational, then a mediator is needed to run on top of these sources and mediate between the different sources and the user.

In spite of much research, a recent survey of data integration agreed that integration of structured, semi-structured, and unstructured data remains a key research problem [Raghavan and Garcia-Molina, 2002]. All of the work done on mediators is directly focused on this problem. Two types of mediators exist: Internet mediators and intranet mediators.

6.6.1 Internet Mediators

Internet mediators respond to user queries by issuing a plurality of queries to search engines on remote Internet sites, consolidate the results returned from the remote search engines and present these results to the user. For example, a user might have a query about books for sale on the Internet and choose to query both *Waldenbooks* and *Daltonbooks*. The Internet mediator would send a request to both sites and come up with a consolidated answer. In the trivial case, all the sites follow a common schema, but an Internet mediator addresses the more challenging problem of reconciling disparate schemas at the time of the query.

At present, there are a small number of existing mediator research projects in the academic world. All of these focus on different areas of the data mediation problem. The MIX project, from the University of California at San Diego, concentrates on schema integration [Baru et al., 1998]. Large data collections are viewed as one large distributed database, wherein all data are represented as XML documents for which there is a well-formed schema. These data are queried using XML query languages. A key disadvantage to this approach is that any legacy data must be made to conform to this schema. This may be reasonable for some small applications, but for multi-terabyte applications, this is not a viable process. The Tukwila Data Integration system under development at the University of Washington adopts a similar approach [Ives et al., 1999], but it too focuses on schema integration *only*; their primary goal is not to provide an answer to a user's question. Stanford's TSIMMIS project concentrates on being able to interface with large volumes of data and search them in a typical Web-search manner [Garcia-Molina et al., 1997].

A mediation infrastructure for digital libraries is described by Melnik, et. al [Melnik et al., 2000]. This infrastructure allows users to develop wrappers around various sources and to query all the sources using a common language. A high-level, query language SDLIP (Simple Digital Library Interoperability

Protocol) is used to query the sources. This requires the user to identify how to identify the correct sources for a given query.

The mediation infrastructure described by Melinik, et. al avoids many of the details of numerous unstructured sources. The query language assumes that users are querying a single document collection; nothing in the query language facilitates queries over structured or semi-structured data. In many respects, this mediation infrastructure is similar to the InfoBus in which CORBA was used to hide details of various unstructured services on the Web [Paepcke et al., 2000]. In each of these efforts, the core focus is on multiple site result integration solely. The problem of source selection, a key challenge for a mediator that must sit on top of numerous sources, is neglected.

Overall, it is difficult for an Internet mediator to accomplish schema reconciliation prior to query execution time. If one source lacks an attribute like *publisher_date* and another has this attribute, it is a non-trivial effort to identify this at the time of the query. Since these sources are completely outside the control of the mediator, it is not feasible to identify sufficient metadata prior to the query. With an intranet mediator, the situation is very different. Metadata can be defined well in advance of the query and schema reconciliation can be accomplished as well.

6.6.2 Intranet Mediator

The key to the architecture of an intranet mediator is that all of the schemas for the data sources are available long before the time of the query. A high-level sample mediator architecture is given in Figure 6.2.

Essentially, the sample mediator consists of the following key components: *Query Processor:* The query processor takes the natural language query and parses it into key grammatical constructs such as *subject, verb,* and *objects.* Additionally, the query parser performs a part-of-speech tagging operation on the query to identify the most likely part of speech for each term in the query. Finally, an entity tagger is used to identify top-level semantic concepts, such as *location, person, place, organization, etc.* in the query. These tools are often used in various question/answering systems, but they lack the efficiency required to work on enormous document collections. Hence, the mediator only uses these tools to parse the *query.* We make the assumption that time exists to do complex natural language processing on the query, but not the document collection.

Level 0 Rules: The first set of rules is referred to as 'level 0'. These rules take the syntactic elements in the query and existing metadata lists and identify higher-level semantic concepts in the query. Consider a course number like "*CS 522*". This might be recognized by the two character prefix "*CS*", and then the three digit sequence.

A level 0 rule might be of the form:

If *subject or object = [list of course prefixes]* [3 digits]
then *subject or object = [COURSE_NUMBER].*

Level 1 Rules: 'Level 1' rules take output from the query processor and semantic concepts identified by level 0 rules and map to one or more *retrieval functions* which are then used to obtain the actual data that comprise the answer to the query.

Retrieval Functions: These are small functions, key to the mediator, which contain the code needed to actually retrieve data from a source. These speak to the source in the language of the source. For example, a relational source will communicate using an SQL script, and an XML source might be sent XML-QL or some other XML query language. This flexibility is a key feature, as it allows the mediator to easily connect to virtually *any* type of data source. Simply stated, if a source can be queried, then the mediator can access it via a retrieval function. One might note that the whole game of retrieval from multiple heterogeneous sources is simply one of taking the English query and, from it, choosing the right retrieval functions. The idea is that the combined efforts of the rules framework (both level 0 and level 1 rules) enable the selection of the correct retrieval functions.

Dispatchers: The various source-type dispatchers handle the task of asynchronously invoking the appropriate retrieval functions for the sources that are deemed appropriate to the query.

Results Manager: The results manager combines the results from the various sources with some sources weighted higher than others.

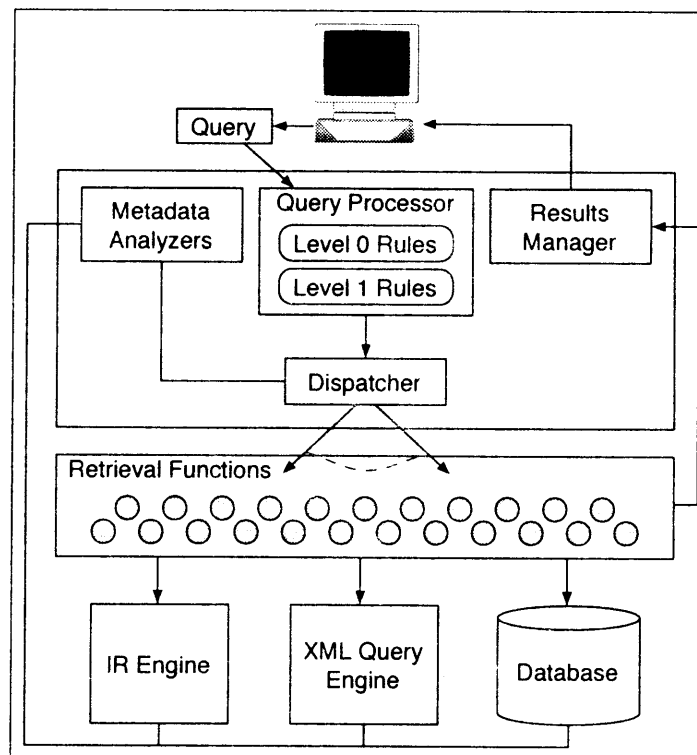
Metadata Analyzers: The mediator also contains analyzers that examine new input sources and identify key aspects of the source. The analyzers assume that a source is the actual data. Analyzers exist for structured, unstructured, and semi-structured data.

6.7 Summary

We discussed several approaches focused on the integration of structured and text data. To aid the reader, we initially provided a limited review of the relational database model and continued with a historical progression of the data integration field. We discussed the key concerns involved in data integration—namely data integrity, portability, and performance—and noted that maintaining and coordinating two separate systems was difficult and expensive to do.

Having motivated the integration of traditional relational database management features with traditional information retrieval functionality, we described

Figure 6.2. Intranet Mediator Architecture



early efforts that extended relational database management systems with user-defined operators. These extensions provided information retrieval functionality, but also potentially incurred performance and portability penalties. We then provided a detailed illustration of the integration of both information retrieval and relation database functionality using standard, unchanged SQL. Next, we described an actual implementation, SIRE, of this approach. Subsequently, the ability to treat XML retrieval as an application of the relational model was explored. The chapter concluded with a brief discussion of data integration using multi-dimensional data models and mediators.

6.8 Exercises

1 Using *Alice in Wonderland* develop a utility to output a file that is suitable for populating the INDEX relation described in this chapter.

2 Load the output obtained in the preceding exercise into the relational DBMS of your choice.

3 Implement a simple dot product SQL query to query the data you have just loaded. Implement ten different queries.

4 Notice that the term *Alice* is replicated numerous times. Implement a Huffman encoding compression algorithm to reduce the space stored for each term. Reload the INDEX relation and compute the amount of storage overhead.

5 Show how the probabilistic approach developed by Robertson and Sparck Jones described in Section 2.2.1 can be implemented as an application of a relational database system. Repeat this exercise for the approach developed by Kwok described in section 2.2.4.

Chapter 7

PARALLEL INFORMATION RETRIEVAL

Parallel architectures are often described based on the number of instruction and data streams, namely single and multiple data and instruction streams. A complete taxonomy of different combinations of instruction streams and data was given in [Flynn, 1972]. To evaluate the performance delivered by these architectures on a given computation, *speedup* is defined as $\frac{T_s}{T_p}$, where T_s is the time taken by the *best* sequential algorithm, and T_p is the time taken by the parallel algorithm under consideration. The higher the speedup, the better the performance. The motivation for measuring speedup is that it indicates whether or not an algorithm scales. An algorithm that has near linear speedup on sixteen processors may not exhibit similar speedup on hundreds of processors. However, an algorithm that delivers very little or no speedup on only two processors will certainly not scale to large numbers of processors.

Multiple Instruction Multiple Data (MIMD) implies that each processing element is potentially executing a different instruction stream. This is the case in most of the modern parallel engines. Synchronization is more difficult with this approach, as compared to a Single Instruction Multiple Data (SIMD) system, because one processor can still be running some code while another is waiting for a message.

In SIMD architectures, all processors execute the same instruction concurrently. A controlling master processor sends an instruction to a collection of slave processors, and they all execute it at the same time on different sequences of data. SIMD systems are effective when all processors work on different pieces of data with the same instruction. In such cases, large speedups using SIMD engines are possible. Some image processing applications, where each pixel or set of pixels is assigned to a processor, are solved efficiently by SIMD solutions.

In this chapter, we include only algorithms written for a parallel processor. We distinguish these algorithms from distributed algorithms since they are fundamentally different. A distributed information retrieval algorithm is designed to satisfy the need to store data in many physically disparate locations. The most known example of a distributed information retrieval system is the World Wide Web (WWW). We discuss this and other distributed information retrieval systems in Chapter 8. However, with parallel systems, the processing elements are close to one another—often on the same circuit board.

7.1 Parallel Text Scanning

In parallel pattern match, the text collection consisting of n documents is partitioned into p partitions (p is typically the number of available processors) [Evans and Ghanemi, 1988]. Each available processor receives a partition of the text and a copy of the query. A sequential algorithm executes on each $\lceil \frac{n}{p} \rceil$ sized portion of text. Once this is done, all of the hits are returned to the controlling processor. Since it is possible for a pattern to span across two or more partitions, an additional step is required to check for matches that span partitions. This extra checking results in additional overhead for the parallel algorithm.

Parallel string matching is a simpler case of parallel text scanning, in that string matching assumes that the text to be searched is completely memory resident. A survey of parallel string matching algorithms is found in [Breslauer and Galil, 1991]. This survey describes several different parallel algorithms that search a text string of size l for a pattern of size k.

The parallel pattern matching algorithm has a key flaw in that patterns which span partitions result in considerable overhead. For some cases, the parallel algorithm yields no speedup at all. The parallel signature file approach yields linear speedup over the sequential file, but run time for this algorithm is not better than the time required to implement a sequential indexing algorithm. This fact was pointed out by Salton when he implemented this algorithm on a Connection Machine and a SUN3 [Salton, 1988]. Additionally, Stone used an analytical model to compute that a sequential machine will outperform a parallel machine with 32K processors. This occurs if an inverted index is used for the sequential matching and the file scan is used on the 32K processor machine [Stone, 1987]. Another repetition of the theme that information retrieval does not require enough processing to enable good parallel processing algorithms is given in [Cockshott, 1989].

Generally, parallel algorithms start with the best sequential algorithm. Comparing a parallel scanner to a sequential scanner is not an accurate measure of speedup, as it is well known that the best sequential algorithms use an inverted index.

7.1.1 Scanning Hardware

Numerous special purpose hardware machines were built to scan text. A survey of these is found in [Hurson et al., 1990]. We briefly review two of these as they serve to illustrate the need for this portion of our taxonomy.

7.1.1.1 Utah Retrieval System

The Utah Retrieval System (URS) is implemented as a non-deterministic Finite State Automata (FSA) with a series of special purpose comparators [Hollaar and Haskins, 1984, Hollaar and Haskins, 1991]. The FSA is constructed so that many non-deterministic paths are explored at the same time; therefore, it never requires a backward look. The URS is essentially a smart disk controller, as the hardware is placed close to the disk controller so that only data that match the retrieval criteria are returned to the calling processor. As always, the motivation behind the special purpose algorithm is run-time performance. While proximity searching can be done, it is not clear that the URS can be used to store weights of terms in a document. Hence, this approach has some of the same problems as a signature-based approach.

Subsequent work with the URS employed an index and a simplified posting list. This posting list does not contain proximity information so the index is used simply to identify which documents should be scanned. The FSA is used to scan the documents to obtain the required response to the query. This scanning step is needed to determine the locations of terms within the document.

7.1.1.2 A Data Parallel Pattern Matching Approach

To avoid precomputation of a FSA and to search large blocks of text simultaneously a data parallel pattern matching (DPPM) algorithm was developed [Mak et al., 1991]. In the DPPM algorithm, a block of data is compared against a sequential serial portion of the pattern. Sequentially characters of the search pattern are compared individually against an entire block of text. Given the high degree of mismatch between the pattern and the block of text an "early-out" mismatch detection scheme flushes the entire block of text. This occurs once a match with the pattern is no longer possible. This early mismatch detection mechanism greatly reduces the total search processing time as redundant comparisons are avoided.

An architecture that relied on multiple individual DPPM search engines to identify document offsets where the pattern matches were found was outlined. Based on simple computations and the predetermined offsets, the required information retrieval operators proposed for the Utah Retrieval System were supported. A VLSI realization of the DPPM engine, and a corresponding analysis of the global architecture, was presented. The analysis demonstrated a potential search rate of one gigabyte of text per second.

Bailey et al. examined parallel search on the PADRE system, a distributed in-memory pattern matching system, and explored the issues of scaling it to one terabyte [Bailey and Hawking, 1996]. Additionally, their scaling approach partitioned data to many CPUs of a virtual machine – an approach that has since been shown to not always be reasonable [Chowdhury and Pass, 2003].

7.1.2 Parallel Signature Files

Several approaches have been used to parallelize the scanning of signature files. Since a signature file can be scanned in an arbitrary order, this structure is inherently parallelizable.

7.1.2.1 Connection Machine

An early algorithm developed for the Connection Machine used signature files to represent documents [Stanfill and Kahle, 1986]. Details of sequential algorithms that use text signatures are described in Section 5.3.

Several signatures are stored in each processing element. Each processing element is assigned signatures only for a single document. The reason for this is that it was assumed that a document would not expand beyond a single processing element. A query is processed by generating the bitmap for a term in the query and broadcasting it to all processors. Each processor checks the bitmap against the list of signatures. When a match occurs, the mailbox in the processing element that corresponds to the document is updated with the weight of the query term. Document weights due to repetition within a document are lost because the signature does not capture the number of occurrences of a word in a document. However, a global weight across the document collection is used.

Once all the signatures are scanned, the documents are ranked by doing a global maximum of all mailboxes. The processors whose mailboxes contain the global maximum are then zeroed, and the global maximum is repeated. This continues until the number of documents that should be retrieved is obtained.

The commercial implementation of this algorithm contained several refinements [Sherman, 1994]. The actual values for the signatures for the CM were:

w = 30 words per signature

s = 1024 bits in a signature

i = 10 hash functions used to code a word

Fifty-five signatures were placed in a single processing element. The assumption that one processing element maps to a corresponding signature is re-

moved. Additionally, weights are not done by document. They are computed for signature pairs. The idea being that a sixty word radius is a better document segment to rank than an entire document. Hence, a weight is maintained for each signature.

To resolve a query, the top one hundred query weights were used. The bitmap for the query was generated as before, and rapid microcode was used to quickly check the corresponding i bits in the signature. Whenever a query term appeared to match a signature, the corresponding weight was updated appropriately. Once the signature match was complete, the signature pairs were then combined using a proprietary scoring algorithm that averaged the weights of the two signatures. The use of signature pairs made it possible to incorporate proximity information into the relevance ranking. Some interprocessor communication occurs to obtain the single signature part that crosses a document boundary (both above and below the processing element). However, the overhead for these two processor "sends" is low because it occurs only between adjacent processors.

The algorithm was completely memory resident, as the application queried a news wire service in which only the most recent news articles were used in retrieval. As documents aged, they were exported from the CM to make room for new documents. Several million documents could then be stored in memory and run-time performance was routinely within one to three seconds.

7.1.2.2 Digital Array Processor (DAP)

Signatures were used initially in the Digital Array Processor (DAP) by using a two-phased search. Many of the parallel algorithms are based on a bit serial implementation of the sequential algorithms given in [Mohan and Willett, 1985]. In this algorithm, signatures are assigned by using a term dictionary and setting a single bit in the signature for each term. The 1024 bit-long signatures are distributed to each processor (4096 processors). Hence, 4096 documents are searched in parallel. The query is broadcast to each processor. Since only one bit is set per term, the number of matching bits is used as a measure of relevance. This uses the assumption that a bit match with the query indicates a match with the term. Since several terms can map to the same bit in the signature, this is not always true.

To verify that a match really occurs, a second phase begins. In this phase, a pattern matching algorithm is implemented, and the document is examined to compute the number of terms that really match. This is done sequentially and only for the documents that ranked highly during the first phase. Performance of the algorithm is claimed to be "good," but no specific results are presented [Pogue and Willett, 1987].

7.1.2.3 HYTERM

Another approach by Lee, HYbrid TExt Retrieval Machine (HYTERM), uses a hybrid approach between special-purpose search hardware and partitioned signature files, which can be done via hardware or software [Lee, 1995]. This architecture employs a signature file using superimposed signatures to identify possible matches to a Boolean request. Once this is done, the false hits are removed either by a software scan or a special-purpose pattern match device.

Signatures are partitioned such that each partition has a certain key or portion of the signatures. This saves memory as the signatures in a given partition need not store the partition key. The key is checked quickly to determine whether or not the entire partition must be searched. The partitions are stored in memory, and are spread across the processors or, as Lee calls them signature modules, as they are filled. Initially, only one signature module is used. Once it is full, a single-bit key is used, and the signatures are partitioned across two processors. The process continues until all free processors are full, then new ones can be added and the process can continue indefinitely.

The actual text is stored across numerous small texts. Once the signature modules have identified candidate documents to be checked for false drops, the text processing modules retrieve the document from a disk. It is noted that by spreading the documents across numerous disks, the resilience to failure improves. When a disk is down, it only means a few documents will be inaccessible. The overall query will still work, it will just have lower precision and recall than if the disk had been working. Documents are uniformly distributed to the disks, either by hashing or round-robin allocation.

7.1.2.4 Transputers

Two algorithms were developed with transputers in the early 1990's [Cringean et al., 1990, Cringean et al., 1991]. The first was a term-based algorithm in which only full terms were encoded in a signature, the second algorithm uses trigrams (overlapping three character sequences—see Section 3.4) to respond to wildcard searches.

Another signature-based algorithm was implemented on a transputer network. Transputers essentially serve as building blocks to an arbitrary parallel interconnection network, and are often distributed as a chip in which *links* are present that can be connected to other transputers. For this approach, different interconnection networks were tested, but ultimately a *triple chain* network was used in which a master processor sends messages to three separate linear arrays. Using only a single linear array, data transmission requires on the order of p steps, where p is the number of processors.

A two-phased algorithm is again used. In this first phase, a master processor sequentially scans signatures for each document. The documents that

correspond to signature matches are then distributed to the p processors, and a sequential string matching algorithm is implemented on each of the p processors. In this work, a modified Boyer-Moore algorithm by Horspool is used for sequential string matching [Horspool, 1983].

During one performance test, the signatures were eliminated and only string matching was done. For this test with fifteen processors, a speedup of 12.6 was obtained. Additional tests with varying signature lengths were conducted. For larger signatures, fewer false hits occur. Thus, less string matching in parallel is needed. With 512 bit signatures, fifteen processors only obtained a speedup of 1.4 because only thirteen percent of the original document collection were searched.

Additional tests were done with signatures based on trigrams instead of terms. Each trigram was hashed to a single bit. The pattern-matching algorithm implemented on each processor was modified to search for wildcards. These searches include symbols that indicate one or more characters will satisfy the search (e.g., a search for "st*" will find all strings with a prefix of "st"). Initial speedups for trigram-based signatures were only 2.1 for fifteen processors. This poor speedup was caused by the sequential signature match in the first phase of the algorithm. To alleviate this, additional transputer components were added so that two and then four processors were used to scan the signature file in parallel. With four transputers for the parallel signature match, speedup improved to 4.5 for twelve processors.

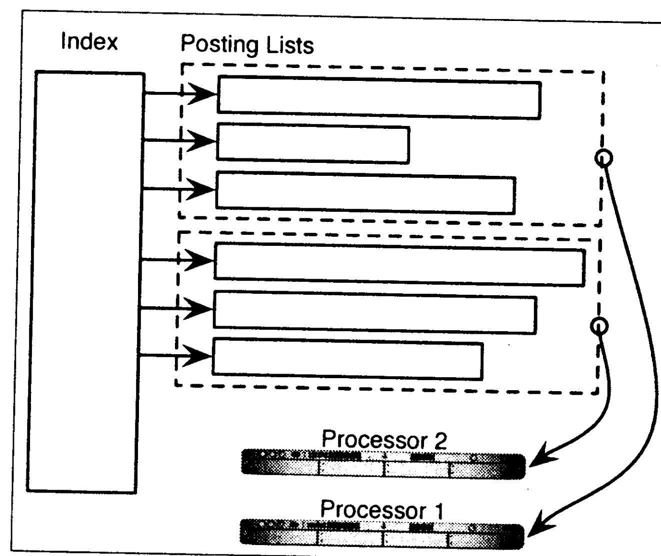
Another term-based transputer algorithm is found in [Walden and Sere, 1988]. In this work, a master processor sends the query to each of the processors where all "relevant" documents are identified and returned to the master for final ranking. "Relevant" is defined as having matched one of the terms in the query. Document signatures are used to save storage, but no work is done to avoid false hits. The interesting aspect of this work is that three different interconnection networks were investigated: ring, linear array, and tree. Speedups for a 10 megabyte document collection with a ring interconnection network were almost linear for up to fifteen processing elements, but fell to only 6.14 for sixty-three processing elements. Essentially, the test collection was too small to exploit the work of the processing elements. In a tree structure, sixty-three processing elements yielded a speedup of 7.33.

7.2 Parallel Indexing

Yet another approach is to parallelize the inverted index. The idea is to partition the index such that portions of the index are processed by different processors. Figure 7.1 illustrates an inverted index that was partitioned between two processors. This is intrinsically more difficult, in that simply partitioning the index and sending an equal number of terms to each of the p processors does not always result in equal amounts of work. Skew in posting list size

poses a difficult problem. Nevertheless, parallel index algorithms were developed for the Connection Machine, the DAP, and some others. We discuss these algorithms in this section.

Figure 7.1. Partitioning an Inverted Index



7.2.1 Parallel Indexing on a Connection Machine

The signature-based algorithm did not improve on the *best* sequential algorithm, so a new approach based on an inverted index was developed. Both an entirely memory resident index and a disk-based index were constructed [Stanfill et al., 1989]. The posting lists were sequences of document identifiers that were placed in a two dimensional array. Mapping between posting list entries, and placement within a two dimensional array was defined. The entries of the posting were allocated one at a time, starting with the first row and moving to the second row only after the first row was full. This has had the effect of allocating data to the different processors (since each column is processed by an individual processor) in a round-robin fashion.

Consider a posting list with terms t_1, t_2, and t_3. Assume t_1 occurs in documents d_1 and d_2. The posting list for this term will be stored in the first two positions of row zero, in an array stored in memory. Assume t_2 occurs in documents d_1 and d_3, and t_3 occurs in documents d_1, d_2, and d_3. For these three terms, the 2×4 dimensional array shown in Table 7.1 is populated.

Table 7.1. Parallel Storage of Posting Lists

1	2	1	3
1	2	3	

Using this approach a row of 1024 postings can be processed in a single step if all processors are used. A full row is referred to as a *stripe*. Since the terms have been scattered across the array, it is necessary to track which entries map to a given posting list. A second array is used for this purpose. It holds an entry for the term followed by a start row, a start column, and a length for the posting list. The start row and column indicate the first location in the posting list that corresponds to the term. Continuing our example, the index table for the terms 0, 1, and 2 is given in Table 7.2.

The first row of this entry indicates that term t_1 contains a posting list that starts at position [0,0] and continues to position [0,1] of the two dimensional posting list array given in Table 7.2. This can be inferred because the row-at-a-time allocation scheme is used.

A query of n terms is processed in n steps. Essentially, the algorithm is:

```
do i = 1 to n
   curr_row = index(i)
   for j = 1 to row_length do in parallel
      curr_doc_id = doc_id(curr_row)
      score(curr_doc_id) = score(curr_doc_id) + weight(curr_row)
   end
end
```

This is only a sketch of the algorithm. Extra work must be done to deactivate processors when an entire row is not required (a bit mask can be used to deactivate a processor). Additionally, for long posting lists, or posting lists that start at the end of a stripe, more than one row must be processed during a single iteration of the inner loop.

Table 7.2. Mapping of Index Terms to Posting List Entries

Term	Start Row	Start Column	Length
t_1	0	0	2
t_2	0	2	2
t_3	1	0	3

For each query term, a lookup in the index is done to determine which stripe of the posting list is to be processed. Each processor looks at its entry for the stripe in the *doc_id*. It is this *doc_id* whose score must be updated, as the entry in the posting list implies that the term in the query is matched by a term in this document. A one-dimensional score array (referred to as a "mailbox" in the original algorithm) is updated. This one-dimensional array is distributed as one element to each processor, so each processor corresponds to a document. For a document collection with more documents than processors a "virtual processor" must be used. (Note: There will never be more than one update to the score element as the posting list only contains one entry for each term-doc appearance). The final ranking is done with a global maximum to find the highest-ranked element of the score array and to successively mask that element. This can be repeated until all documents have been retrieved.

This algorithm obtains good speedup when a posting list uses an entire stripe. The problem is that often a stripe is only being used for one or two entries. A posting list containing one entry results in 1023 processors doing nothing while one processor issues the update to the score array. The posting table can be partitioned by node [Stanfill, 1990, Stanfill and Thau, 1991] to accommodate clusters of processors (or nodes). This facilitates the movement of data from disk into the posting array.

One node consists of thirty-two processors. The problem is that if the posting list entries are arbitrarily assigned to nodes based on a document range (i.e., node one receives postings for documents zero and two, while node two receives postings for documents one and three) it is conceivable that one node can have substantially more postings than another. To avoid this problem, the nodes are partitioned such that each partition contains a range of term identifiers. Occasionally, empty space occurs in a partition as there may be no data for a given range of terms in a particular range of documents. It can be shown that for typical document collections, eighty to ninety percent of the storage is used. This partitioned posting list yields improved speedup as the number of idle processors is reduced.

7.2.2 Inverted Index on the Connection Machine

The previous algorithm processed one query term at a time. Parallel processing was done to update the ranking, but even with optimal processor utilization (fully used stripes), the time taken for a query of t terms is on the order of $O(t)$. An approach that allows logarithmic time is given in [Asokan et al., 1990]. The algorithm consists of the following steps:

Step 1: Partition the set of processors into clusters. Each cluster works on a single query term.

Step 2: Each cluster simultaneously references the index to determine the posting list that corresponds to its own cluster. Hence, cluster 1 obtains the posting list for term 1, cluster 2 obtains the posting list for term 2, etc.

Step 3: Use all p processors to merge the $\lceil \frac{n}{p} \rceil$ posting lists, where n is the number of documents. This effectively produces a sorted list of all documents that are referenced by the terms in the query. Since the posting list contains the weight, a document weight appears as well. Hence, a merged posting list might appear as:

$< D1, 0.5 >< D1, 0.3 >< D2, 0.5 >< D2, 0.9 >.$

This posting list occurs if document one contains two query terms with weights of 0.5 and 0.3, respectively, and document two contains two terms with weights of 0.5 and 0.9 respectively.

Step 4: Use all processors to eliminate duplicates from this list and generate a total score. After this step, our posting list appears as:

$< D1, 0.8 >< D2, 1.4 >$

Step 5: Sort the posting list again based on the score assigned to each document. Our posting list will now appear as:

$< D2, 1.4 >< D1, 0.8 >$

A modified bitonic sort can be done to merge the lists so the complexity of the algorithm is $O(\log_2 t)$ time. This appears superior to the O(t) time, but it should be noted that the algorithm requires $O(\lceil \frac{n}{p} \rceil)$ processors assigned to a cluster to process a single posting list for a given term. If too many terms exist, it may be necessary to overlay some of the operations.

7.2.3 Parallel Indexing on a Digital Array Processor (DAP)

As with the Connection Machine, an earlier scanning algorithm that uses term signatures on the DAP, was replaced with an indexing algorithm [Reddaway, 1991, Bond and Reddaway, 1993].

The key difference between the DAP algorithm and the CM algorithm is that a compressed posting list is used. Additionally, the algorithm running on the DAP is claimed to be more efficient as the DAP uses a simpler interconnection network (a mesh instead of a hypercube) and the global operations such as global maximum are substantially faster. Since no really remote "send" operations are done, the authors of the DAP approach claim that it is not necessary to have a hypercube.

The compression scheme is based on the observation that large hit lists often have the same leading bits. Consider a hit list that contains documents 8, 9, 10,

11, 12, 13, 14, and 15. The binary values all have a leading bit of 1 (1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111). By allocating one bit as a *block indicator*, the hits within the block can be stored in three bits. Hence, block 1 would contain the references (000, 001, 010, 011, 100, 101, 110, and 111). The total bits for the representation changes from $(8)(4) = 32$ to $1 + (8)(3) = 25$. Clearly, the key to this representation is the number of hits within a block. For the DAP, a 24-bit code (no compression) is used for rare terms (those that occur only once in every 50,000 documents). For terms that appear more frequently, an 8-bit block code with a 16-bit offset within the block (the block holds up to 64K) references entries in the posting list. Finally, for the most frequent terms, a 64K document block is treated as 256 separate sub-blocks. A key difference in the parallel algorithm for the DAP is that the expansion of the posting list into an uncompressed form is done in parallel. Performance of the DAP-based system is claimed to be 200 times faster than the previous sequential work. Other experiments using 4096 processors indicate the DAP 610 yields a significant (over one hundred times faster) improvement over a VAX 6000 [Manning, 1989, Reddaway, 1991].

7.2.4 Partitioning a Parallel Index

An analytical model for determining the best means of partitioning an inverted index in a shared nothing environment is given in [Tomasic and Garcia-Molina, 1993]. Three approaches were studied. The first, referred to as the *system* approach, partitioned the index based on terms. The entire posting list for term *a* was placed on disk 1, the posting list for term *b* was placed on disk 2, etc. The posting lists were assigned to disks in a round-robin fashion.

Partitioning based on documents was referred to as the *disk* strategy. In this approach, all posting list entries corresponding to document 1 are placed on disk 1, document 2 on disk 2, etc. Documents were assigned to disks in a round-robin fashion. Hence, to retrieve an entire posting list for term *a*, it is necessary to retrieve the partial posting lists from each disk for term *a* and merge them. Although the merge takes more time than the system entry, the retrieval can take place in parallel.

The *host* strategy partitioned posting list entries for each document and placed them on separate processors. Hence, document 1 is sent to processor 1, document 2 to processor 2, etc.

An analytical model was also developed by fitting a frequency distribution to some text (a more realistic approach than blindly following Zipf's Law). The results of the analytical simulation were that the *host* and *disk* strategy perform comparably, but the *system* strategy does not perform as well. This is because the *system* strategy requires sequential reading of potentially long posting lists and transmission of those lists. The *system* strategy becomes competitive when the communication costs were dramatically reduced.

7.2.5 A Parallel Inverted Index Algorithm on the CM-5

Another algorithm on the CM-5 is described in [Masand and Stanfill, 1994]. In this work, the documents were distributed to sixty-four different processors where a compressed inverted index was built for each of the processors. Construction of the inverted index was extremely fast. In twenty minutes, a 2.1 gigabyte document collection was indexed, and the size of the index file was only twenty-four percent of the size of the raw text.

Queries were processed by sending the query to each of the processors and obtaining a relevance ranking for each processor. Once obtained, a global maximum was computed to determine the highest ranked document among all the processors. This document was ranked first. The global maximum computation was repeated until the number of documents that were to be retrieved was reached.

7.2.6 Computing Boolean Operations on Posting Lists

Another area in which parallel processing is used in conjunction with an inverted index is the computation of Boolean operations on two posting lists, A and B [Bataineh et al., 1989, Bataineh et al., 1991]. The posting lists are partitioned so that the total number of elements in each partition is of equal size. The Boolean computation is obtained by computing the Boolean result for each partition. There is no need to compare values located in one partition with another because the partitions are constructed such that each partition contains values within a specific range and no other partition contains values that overlap within that range. This partitioning process can be done in parallel.

Once the partitions are identified, each one is sent to a separate processing element. Subsequently, each processing element individually computes the Boolean result for its values. Finally, the results are obtained and stored on disk. The algorithm was originally implemented on the NCUBE/4 and the Intel iPSC/2. For posting lists corresponding to term *human* and *English* of 520,316 and 115,831 postings, respectively (the MEDLINE database was used), speedups of five for an eight processor NCUBE were observed and a speedup of seven for a sixteen processor IPSC were obtained. It was noted that the parallel algorithm began to degrade as the number of processors increased. As this happens, the amount of work per processor is reduced and communication overhead is increased.

7.2.7 Parallel Retrieval as an Application of an RDBMS

One of the motivating factors behind the development of an information retrieval engine as an application of the relational database model (see Chapter 6) was the availability of commercial parallel database implementations. Exe-

cuting the SQL scripts that implement the information retrieval application on a parallel relational database engine results in a parallel implementation of an information retrieval system.

In [Grossman et al., 1997], the feasibility of implementing a parallel information retrieval application as a parallel relational database application was demonstrated. From these findings, it was hypothesized that it was possible to develop a scalable, parallel information retrieval system using parallel relational database technology.

To validate this hypothesis, scaling experiments were conducted using a twenty-four processor database engine [Lundquist et al., 1999]. The initial findings were, however, disappointing. Using the same relational table definitions described in [Grossman et al., 1997], only a forty percent processor efficiency was achieved.

The parallel hardware used (the NCR DBC/1012) for the experiments supports automatic load balancing. The hashing scheme used to implement the load balancing is based on the index structure in the defined relational schema. In the DBC/1012 architecture used to evenly distribute the load, a uniformly distributed set of attributes must be the input to the hashing function. In the initial implementation, the hashing function was based on terms, and thus, was nonuniform. Modifying the input to the hashing function to include document identifiers, as well as terms, resulted in a uniform distribution of load to the processors. In later experimentation, a balanced processor utilization of greater than 92% was demonstrated, and a speedup of roughly twenty-two using twenty-four nodes, as compared to a comparable uniprocessor implementation, was achieved.

7.2.8 Summary of Parallel Indexing

Parallel processing within information retrieval is becoming more applicable as the cost of parallel I/O is reduced. Previous algorithms had problems with memory limitations and expensive communication between processors. Signature files were popular, but have not been used recently due to their unnecessarily high I/O demand and their inability to compute more sophisticated measures of relevance. Parallel inverted index algorithms are becoming more popular, and with improved compression techniques, they are becoming substantially more economical.

7.3 Clustering and Classification

Parallel clustering and classification implementations were developed for the Intel Paragon [Ruocco and Frieder, 1997]. Using a production machine, the authors developed a parallel implementation for the single-pass clustering and single-link classification algorithms (see Section 3.2). Using the *Wall Street*

Journal portion of the TREC document collection, the authors evaluated the efficiency of their approach and noted near-linear scalability for sixteen nodes.

More recently, the popular and effective Buckshot clustering algorithm [Cutting et al., 1992] was also parallelized [Jensen et al., 2002]. This parallel algorithm was developed for use on a low-cost cluster of PC's, and uses MPI for communication. No specialized parallel hardware is needed. The authors tested this algorithm using larger, more modern TREC document collections and demonstrated near-linear scalability in terms of number of nodes and collection size.

To accurately compare the efficiency of the developed approaches, the results derived from both the parallel and serial implementations must be identical. Otherwise, an improvement in the efficiency of the algorithm (via parallelism) could come at the expense of accuracy.

The single-pass clustering algorithm is data, presentation, and order dependent. Namely, the order in which the data are presented as input directly affects the output produced. Thus, it was necessary to provide mechanisms in the parallel implementation that mimicked the order of the presentation of the documents as input to the algorithm. Guaranteeing the identical order of document presentation resulted in the formation of identical clusters in both the serial and parallel implementations. The authors noted that the size of the clusters varied dramatically and suggested measures to reduce the cluster size disparity. Since the size disparity is a consequence of the single-pass algorithm, no modification was made.

7.4 Large Parallel Systems

There has been some work towards developing large, general-purpose parallel IR systems. This section details efforts for some of the most prominent ones. An overview of some work on parallel information retrieval systems can be found in [MacFarlane et al., 1997].

7.4.1 PADRE - A Parallel Document Retrieval Engine

The PADRE system for parallel document retrieval was initially developed for text retrieval on the Fujitsu AP1000 system. Over the years, it has been developed into a modern search system, and currently forms the core of CSIRO's Panoptic Enterprise search engine. At its core, PADRE is a distributed, inmemory pattern-matching system that also supports relevance ranking. The architecture has evolved over the years, first being developed for the AP1000 system as outlined in [Hawking, 1991, Hawking, 1994b, Hawking, 1994a, Bailey and Hawking, 1996], and later being applied to tasks as esoteric as XML retrieval in the recent INEX competition [Hawking et al., 2000, Craswell et al., 2002, Vercoustre et al., 2002].

Bailey et al. examined parallel search on the PADRE system and explored the issues of scaling it to one terabyte [Bailey and Hawking, 1996]. Additionally, their scaling approach partitioned data to many CPUs of a virtual machine – an approach that has since been shown to not always be reasonable [Chowdhury and Pass, 2003].

7.4.2 Frameworks for Parallel IR

Some work has been done investigating what frameworks would be most suitable to performing Parallel IR on Symmetrical Multiprocessors[Lu et al., 1997]. A mutithreaded, multitasking search engine was built to investigate the most efficient way to execute parallel queries. In addition, a simulator supports the varying of system parameters such as number of CPUs, threads, and disks, and compared the results of the simulator to those of their implementation. Using this approach, bottleneck points were identified. At this points, no additional resources such as threads, CPUs, and disks do not provide any further scalability. Generally, scalable retrieval performance was found for a variety of system configurations.

7.4.3 PLIERS - Portable Parallel IR using MPI

Researchers at Microsoft Research Cambridge have also developed a parallel information retrieval system called PLIERS [MacFarlane et al., 1999]. This system makes use of the MPI parallel application framework to create a portable parallel IR system [Gropp and Lusk, 1998]. The PLIERS system contains parallelized versions of the Indexing, Document Search, Document Update, and Information Filtering modules of the Okapi uniprocessor IR system [Robertson, 1997], made parallel via the use of standard MPI communication procedures. In addition, the system supported several modes of search, including Boolean and Proximity search, passage retrieval, and relevance-based search. Also, several different MPI implementations found that there were some implementation-specific differences, but nothing that impeded portability. Eventually, the system was ported to many different architectures such as a Network of Workstations (NOW), the Fujitsu AP1000 and AP3000 parallel machines, a cluster of PC's, and an Alpha farm. The authors conclude that there is a performance gain to be had by parallel IR systems designed in this way, and in particular they found that MPI collective operations improve query transactions for IR and term selection in information filtering.

7.5 Summary

As volumes of data available on-line continued to grow, information retrieval solutions needed to be developed that could cope with ever expanding collections. Towards addressing this data growth explosion, parallel solutions

were investigated. Initially, parallel information retrieval solutions focused on hardware-based full-text filtering. Eventually, these hardware solutions gave way to software implementations that roughly mirrored the hardware approaches. Recent parallel efforts are mostly algorithmic and architecturally independent.

We began our review by describing parallel text scanning techniques. We described two hardware solutions for full-text scanning, the Utah Retrieval System and the data parallel data matching system. Both systems supported hardware-level filtering to reduce the retrieved document sets. Although they did demonstrate significant improvements as compared to software full-text scanning, in general, full-text scanning introduces excessive I/O demands as all documents must be scanned. Later efforts using the Utah Retrieval System relied on indexing. However, most recent architectures use general purpose processors since they are able to more quickly incorporate enhancements. This has reduced the popularity of special purpose solutions.

Later efforts developed software supported text scanning. To reduce the I/O demands associated with full-text scanning, most efforts focused on signature analysis. Early studies relied on SIMD architectures, namely the DAP architecture and the Connection Machine. Results demonstrated limited scalability in terms of performance. Later signature analysis efforts were evaluated on MIMD systems such as the Inmos Transputers with somewhat better results.

The prohibitive I/O demands of text scanning approaches, both full-text and signature analysis, resulted in the development of parallel indexing approaches. The need for index-based approaches was clearly demonstrated in [Stone, 1987] where it was shown that serial computers using indexing techniques sustained faster retrieval speeds than parallel engines using a signature analysis approach. Parallel indexing approaches on both SIMD and MIMD architectures were developed, with some efforts resulting in near linear speedup.

We then described parallelizations of both clustering and classification algorithms. The approaches described were implemented on an Intel Paragon that was in production use. For all the algorithms studied, near linear speedup was noted.

We concluded this chapter with a brief discussion of Parallel Search Systems. Parallel information retrieval continues to be a relatively unexplored area. Parallel, scalable algorithms that efficiently support the strategies discussed in Chapter 2 and the utilities listed in Chapter 3 need to be developed. Currently, very few such algorithms are known, and even fewer, have been implemented and evaluated in a production environment.

A different approach to developing parallel information retrieval systems was addressed in [Lundquist et al., 1999]. In these efforts, a mapping from information retrieval operators onto parallel databases primitives was defined. Parallelism was achieved without requiring new parallel algorithms to be de-

veloped. Roughly a 22-fold speedup using twenty-four nodes was achieved. Such speedup is encouraging, and especially so, since it was unnecessary to implement new software.

Given the diversity of the commercially available parallel systems and the vast types of applications that constitute the realm of information retrieval, all that is clear is that it is still an open question of how best to support the domain of parallel information retrieval.

7.6 Exercises

1 Develop an average-case algorithmic analysis for a sequential inverted index for a *tf-idf* vector space query with t terms. Compare this to a parallel linear scan of a document collection with p processors.

2 Develop an algorithm to search an inverted index in parallel with a MIMD machine that will perform as well as or better than the sequential algorithm. Analyze your algorithm and clearly describe your analysis.

3 Design a simple parallel document clustering algorithm and analyze its performance. Compare this to a sequential document clustering algorithm.

Chapter 8

DISTRIBUTED INFORMATION RETRIEVAL

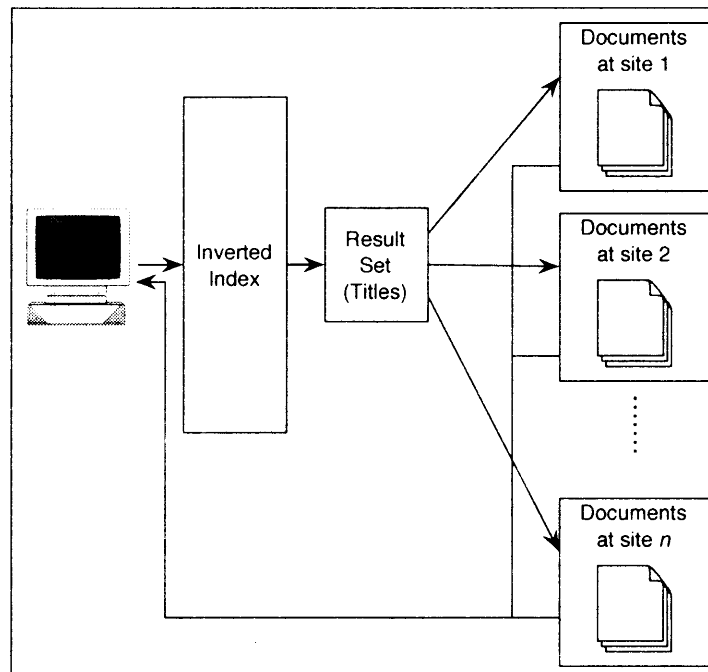
Until now, we focused strictly on the use of a single machine to provide an information retrieval service. In Chapter 7, we discussed the use of a single machine with multiple processors to improve performance. Although efficient performance is critical for user acceptance of the system, today, document 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. For a comprehensive discussion from a data engineering perspective on the engineering of data processing systems in a distributed environment, see [Shuey et al., 1997].

One of the latest popular processing infrastructures is the "Grid" [Foster et al., 2001, Foster, 2002, Alliance, 2004]. 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 modern 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 (see Figure 8.1).

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. We describe this model in Section 8.1.

In Section 8.2, we also briefly discuss Web search engines. We note that a Web search engine is really just an implementation, albeit a very popular one, of some of the algorithms and efficiency techniques already discussed in this

Figure 8.1. Distributed Document Retrieval



book. We do not include the specifics of any particular engine as such details are not only often proprietary but are constantly changing. Had we chosen to provide such details, they would be obsolete by the time our book appeared in press. Furthermore, by including such details we would turn this chapter into an endorsement of a particular engine, and that would be inappropriate and outside of the scope of this book. We do note, however, that a thorough listing of search engines is available at **www.searchenginewatch.com**. Several popular Web engines include *Google*, *Yahoo!*, *MSN Search*, and *AOL Search*.

Finally, Section 8.4 concludes this chapter with a brief description of Peer-to-Peer (P2P) efforts. Such efforts are currently only in their relative infancy. In future editions of this book, we hope to better describe the advancement in this now fledgling specialty.

8.1 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.

8.1.1 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 $\delta_j : R_j \rightarrow 2^{D_j}$ is a mapping assigning the $j^{\underline{th}}$ 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, it was proposed in [Turski, 1971] that the triple be expanded to a quadruple as:

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

where T is a set of distinct terms and the relation $\rho \subset T \times T$ such that $\rho(t_1, t_2)$ implies that t_1 is a synonym of t_2. Using the synonym relation, it is possible to represent documents as a set of *descriptors* and a set of *ascriptors*. Consider a document D_1, the set of descriptors d consists of all terms in D_1 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.

In addition to the thesaurus, a generalization relation over the sets of descriptors is defined as $\gamma \subset d \times d$ where $\gamma(t_1, t_2)$ implies that t_1 is a more general term than t_2. Hence, $\gamma(animal, dog)$ is an example of a valid generalization.

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 [Lenat and Guha, 1989], 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 $\preceq$ and let $t(d_i)$ indicate the list of descriptors for document d_i. Partial ordering, $\preceq$, 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 d_1 whose descriptors are all generalizations of the descriptors found in d_2 will have the ordering $d_1 \preceq d_2$. 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 q_1 must include (be a superset of) all documents that correspond to the documents that correspond to a more specific query q_2, where $q_1 \preceq q_2$. Formally:

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

The model described here was proven to be inclusive in [Turski, 1971]. This means that if two queries, q_1 and q_2, are presented to a system such that q_1 is more general than q_2, it is not necessary to retrieve from the entire document collection for each query. It is only necessary to obtain the answer set for q_1, $\delta(q_1)$, and then iteratively search $\delta(q_1)$ to obtain the $\delta(q_2)$.

8.1.2 Distributed Information Retrieval System Model

The centralized information retrieval system can be partitioned into n local information retrieval systems $S_1, S_2, \ldots, S_n$ [Mazur, 1984]. Each system S_j is of the form: $S_j = (T_j, D_j, R_j, \delta_j)$, where T_j is the thesaurus; D_j is the document collection; R_j the set of queries; and $\delta_j : R_j \rightarrow 2^{D_j}$ maps the queries to documents.

By taking the union of the local sites, it is possible to define the distributed information retrieval system as:

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

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 states 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 γ 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 T_1 is a superset of T_2.

$$D_1 \supset D_2$$

The document collection at site S_1 contains the collection D_2.

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

The queries at site S_1 contain those found in S_2.

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

The document collection returned by queries in S_1 will include all documents returned by queries in S_2. The following example illustrates that an arbitrary partition of a hierarchy may not yield valid subsystems.

Consider the people hierarchy:

γ(people, Harold), γ(people, Herbert), γ(people, Mary)

and the second animal hierarchy:

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

Assume that the hierarchy is split into sites S_1 and S_2. The hierarchy at S_1 is:

γ(people, Harold), γ(people, Mary)
γ(animal, cat), γ(animal, dog), γ(dog, doberman), γ(dog, poodle)

The hierarchy at S_2 is:

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

Consider a set of documents with the following descriptors:

D_1 = (Mary, Harold, Herbert)
D_2 = (Herbert, dog)
D_3 = (people, dog)
D_4 = (Mary, cheshire)
D_5 = (Mary, dog)
D_6 = (Herbert, black-cat, doberman)
D_7 = (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 S_1 will only retrieve documents D_3 and D_5, and S_2 will only retrieve documents D_6 and D_7. Hence, documents D_2 and D_4 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 S_1 and S_2 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 [Mazur, 1988].

8.2 Web Search

No chapter on distributed information retrieval would be complete without some mention of Web search engines. Search tools that access Web pages via the Internet are prime examples of the implementation of many of the algorithms and heuristics discussed in this book. 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 frequently (some vendors produce new releases or publish fixes in a week). 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.

8.2.1 Evaluation of Web Search Engines

As will be discussed in Section 9, in 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) is described in [Beitzel et al., 2003b]. 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.

8.2.2 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. For example, in [Ma et al., 2003], text is efficiently extracted from

template generated Web documents; the remainder 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. In [Beitzel et al., 2004a], 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.

8.2.3 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 [Beitzel et al., 2004b]. 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. What optimizations can you come up with using such logs? What measures would you use to demonstrate success? We are sure that in the next edition of the book, many such measures and optimizations will be described.

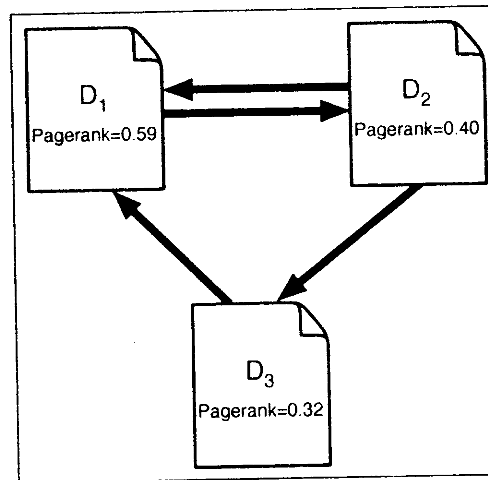
8.2.4 Page Rank

We close this section on Web search with the most popular algorithm for improving Web search. This PageRank algorithm (named after Page) was first described in [Brin and Page, 2000]. It extends the notion of hubs and authorities in the Web graph originally described in [Kleinberg, 1999]. PageRank is at the heart of the popular Web search engine, Google. Essentially, the PageRank 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 $D_1 \ldots D_n$ is defined as follows:

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

where $C(D_i)$ is the number of links out from page D_i 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. For the example in Figure 8.2, using the common dampening factor of 0.85 and initializing each PageRank to 1.0, it took 8 iterations before the scores converged.

Figure 8.2. Simple PageRank Calculation



8.2.5 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 indeed was proposed as early as TREC-4 . Later, in the CYBERosetta prototype [Desantis et al., 1996] 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.

8.3 Result Fusion

Fusing result sets as a means to improve accuracy became of wide academic research interest shortly after Lee [Lee, 1997] hypothesized on what conditions yielded successful fusion. That is, Lee used several common result fusing heuristics to merge results obtained from multiple independent search engines and demonstrated that relevant documents had a greater level of overlap than the set of non-relevant documents. His findings demonstrated that using the CombMNZ fusion heuristic resulted in higher retrieval accuracy than any of the individual search engines used. Aloui, et. al. expanded the work of Lee and concluded that to best capitalize on result fusion techniques, the individual search engines used had to greatly differ in their processing strategies and utilities [Alaoui et al., 1998].

Recently, in [Chowdhury et al., 2001, Beitzel et al., 2003a] a study revisiting the Lee fusion hypothesis was conducted. In this study, unlike in Lee's original study, all the system parameters, e.g., stemmers, parsers, stop-word list, etc., were kept constant and only the similarity measures used were varied. Furthermore, also unlike the Lee study, only the top existing similarity measures were used. The results demonstrated that the overlap difference between relevant and non-relevant documents that Lee described was far less significant when these new measures were used. Surprisingly, it was other system parameters that affected the overall fusion gains with greater impact than the initially postulated similarity measured used.

This observation relied upon using newly available highly accurate similarity measures. From the findings, it was postulated that over the years, given the accuracy improvement of the similarity measures, there remained only an insignificant difference between the result sets obtained using any of the better measures. Given only a slight difference between the result sets available, fusion, at least in that form, yielded only minimal improvements if not an actual reduction in retrieval accuracy. This makes sense when one considers that retrieval strategies have evolved over time to include useful features that were first generated for use with another strategy. Incorporation of term frequency and document length are now common across all retrieval strategies. Hence, as these strategies have evolved, it is not too surprising that they return very similar documents. This certainly reduces the likelihood that fusing retrieval strategies will yield improved effectiveness.

In terms of efficiency, in [Liu et al., 1996], an effort to enhance Web server performance improvements is described. They discussed the use of pre-started processes, or *cliettes*, to avoid the start-up costs of starting processes from a typical common gateway interface (CGI). This was used to implement a prototype system that provides search access to eight library collections.

Most current Web servers use a very detailed, full-text index, but if the Web continues to grow it may not be practical to use a single index. Early work in the area of Web-based distributed query processing was done by [Duda and Sheldon, 1994] in which a system that used the Wide Area Information Service (WAIS) only sent queries to certain servers based on an initial search of the content of those servers. The content was described by some specific fields in the documents that exist on each server such as *headline* of a news article or *subject* of an e-mail message. The use of a content index is the middle ground between sending the request to all of the servers, or providing a very detailed full-text index, and sending the request to only those servers that match the index.

More work done for the Glossary-of-Servers Server (GlOSS) builds a server that estimates the best server for a given query, based on the vector-space model [Gravano and Garcia-Molina, 1995]. The query vector is matched with a vector that characterizes each individual server. The top n servers are then ranked and searched. Several means of characterizing a server are explored. The simplest is to sum the *tf-idf* weights of each term on a given server and normalize based on the number of documents on the server. This yields a centroid vector for each server. A *tf-idf* vector space coefficient (as described in Section 2.1) can then be used to rank the servers for a given query. Different similarity coefficient thresholds at which a server is considered a possible source and assumptions used to estimate which databases are likely to contain all of the terms in the query are also used. It is estimated that the index on the GlOSS server is deemed to be only two percent of the size of a full-text index.

Query processing using a full-text index on a Web server can be done with any of the combination of strategies and utilities described in Chapters 2 and 3. However, an additional strategy based on the use of hypertext links found on Web pages has been investigated [Yuwono and Lee, 1996]. In this work, a strategy referred to as *vector spreading activation* was investigated in which documents were ranked based on a match with a term in a simple query. Additionally, documents that contained links to the original result set were added to the result set. The weight of the new *linked* documents was scaled to be less than the weight of the documents in the original result set. Experiments with scaling factors of zero and 0.5, with increments of 0.1, showed that 0.2, was the best scaling factor. Vector spreading activation was shown to be slightly better than *tf-idf* when average precision was measured for a small test collection of only 2,393 Web pages. Additionally, this system did not use a full-text

index. The indexer uses only HTML tokens such as terms in boldface or italics, the first sentence of every list item, titles, all-level headings, and anchor hypertexts.

What, by necessity, differs in the Web based search engine domain from the traditional information retrieval system environment is the means to evaluate individual systems. In conventional environments, a standard benchmark query and data mix is used to compare across systems. In the Web domain, however, each search engine indexes only a portion of the available data, and the portions do not necessarily overlap nor do they necessarily index even the same version (different time period) of the data where they do indeed overlap.

8.4 Peer-to-Peer Information Systems

We now turn our focus to an emerging field, a cross between the networking domain and the information retrieval discipline, namely Peer-to-Peer (P2P) architectures. By definition, Peer-to-Peer architectures are distributed environments where each node in the network is potentially a source for information (a server), a client in need of information (a client), and an intermediate router (a router) of information. Each node is independent and the system operates in a purely decentralized manner. In the realm of information retrieval systems, the provided resources are in the form of searchable data.

The main characteristics of P2P systems are their ad-hoc nature and durability. P2P systems can gracefully handle the joining and leaving of nodes from the system. The resources offered by these nodes are dynamically added or removed from the system as necessary. Furthermore, the failure of a single node does not destroy the overall system.

The origin of the P2P movement is often attributed to Napster, the music file sharing system, although Napster actually relied on a centralized implementation. That is, Napster was not decentralized, and hence, was not peer-to-peer in the pure sense of the definition. However, Napster did offer P2P functionality in that users could dynamically share files with others. Besides the inherent single point of contention in terms of performance and reliability, Napster's centralized implementation eventually doomed it to legal action, and today, Napster no longer exists in its original form. The demise of Napster taught enthusiasts a lesson. In response, they created the Gnutella protocol [V0.4, 2004], which is truly P2P, and serves as the basis of much of today's P2P research. (A later version of the Gnutella protocol [V0.6, 2004] also exists and extends P2P architectures to include hierarchies. This protocol and its applications are discussed later.)

Systems based on the Gnutella (Version 0.4) protocol generally provide only primitive search capability. That is, they generally rely on exact name search typically accomplished via sub-string matching. Specifically, a query matches a file if all the terms in the query are sub-strings of the file's metadata. Files that