each comparison involves accessing the text at random positions on the disk. This will severely degrade the performance of the sorting process.

We explain an algorithm especially designed for large texts. Split the text into blocks that can be sorted in main memory. Then, for each block, build its suffix array in main memory and merge it with the rest of the array already built for the previous text. That is:

- build the suffix array for the first block,
- build the suffix array for the second block,
- merge both suffix arrays,
- build the suffix array for the third block,
- merge the new suffix array with the previous one,
- build the suffix array for the fourth block,
- merge the new suffix array with the previous one,
- ... and so on.

The difficult part is how to merge a large suffix array (already built) with the small suffix array (just built). The merge needs to compare text positions which are spread in a large text, so the problem persists. The solution is to first determine how many elements of the large array are to be placed between each pair of elements in the small array, and later use that information to merge the arrays without accessing the text. Hence, the information that we need is how many suffixes of the large text lie between each pair of positions of the small suffix array. We compute counters that store this information.

The counters are computed without using the large suffix array. The text corresponding to the large array is sequentially read into main memory. Each suffix of that text is *searched* in the small suffix array (in main memory). Once we find the inter-element position where the suffix lies, we just increment the appropriate counter. Figure 8.10 illustrates this process.

We analyze this algorithm now. If there is $O(M)$ main memory to index, then there will be $O(n/M)$ text blocks. Each block is merged against an array of size $O(n)$, where all the $O(n)$ suffixes of the large text are binary searched in the small suffix array. This gives a total CPU complexity of $O(n^2 \log(M)/M)$.

Notice that this same algorithm can be used for index maintenance. If a new text of size n' is added to the database, it can be split into blocks as before and merged block-wise into the current suffix array. This will take $O(nn' \log(M)/M)$. To delete some text it suffices to perform an $O(n)$ pass over the array eliminating all the text positions which lie in the deleted areas.

As can be seen, the construction process is in practice more costly for suffix arrays than for inverted files. The construction of the supra-index consists of a fast final sequential pass over the suffix array.

Indexing times for 250 Mb of text are close to 0.8 Mb/min on the reference machine. This is five to ten times slower than the construction of inverted indices.

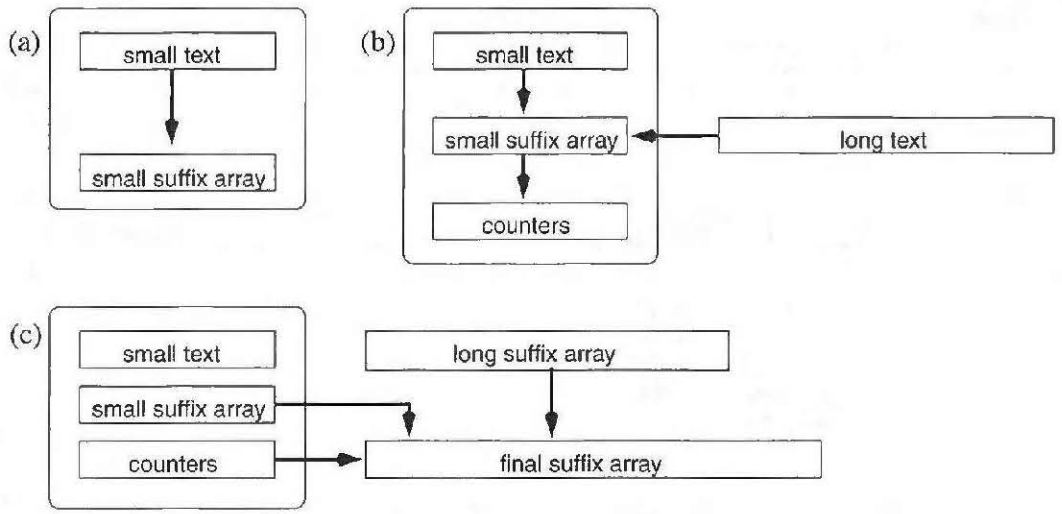


Figure 8.10 A step of the suffix array construction for large texts: (*a*) the local suffix array is built, (*b*) the counters are computed, (*c*) the suffix arrays are merged.

8.3.2 Signature Files

Signature files are word-oriented index structures based on hashing. They pose a low overhead (10% to 20% over the text size), at the cost of forcing a sequential search over the index. However, although their search complexity is linear (instead of sublinear as with the previous approaches), its constant is rather low, which makes the technique suitable for not very large texts. Nevertheless, inverted files outperform signature files for most applications.

Structure

A signature file uses a hash function (or 'signature') that maps words to bit masks of B bits. It divides the text in blocks of b words each. To each text block of size b, a bit mask of size B will be assigned. This mask is obtained by bitwise ORing the signatures of all the words in the text block. Hence, the signature file is no more than the sequence of bit masks of all blocks (plus a pointer to each block). The main idea is that if a word is present in a text block, then all the bits set in its signature are also set in the bit mask of the text block. Hence, whenever a bit is set in the mask of the query word and not in the mask of the text block, then the word is not present in the text block. Figure 8.11 shows an example.

However, it is possible that all the corresponding bits are set even though the word is not there. This is called a *false drop*. The most delicate part of the design of a signature file is to ensure that the probability of a false drop is low enough while keeping the signature file as short as possible.

The hash function is forced to deliver bit masks which have at least ℓ bits set. A good model assumes that ℓ bits are randomly set in the mask (with possible repetition). Let $\alpha = \ell/B$. Since each of the b words sets ℓ bits at

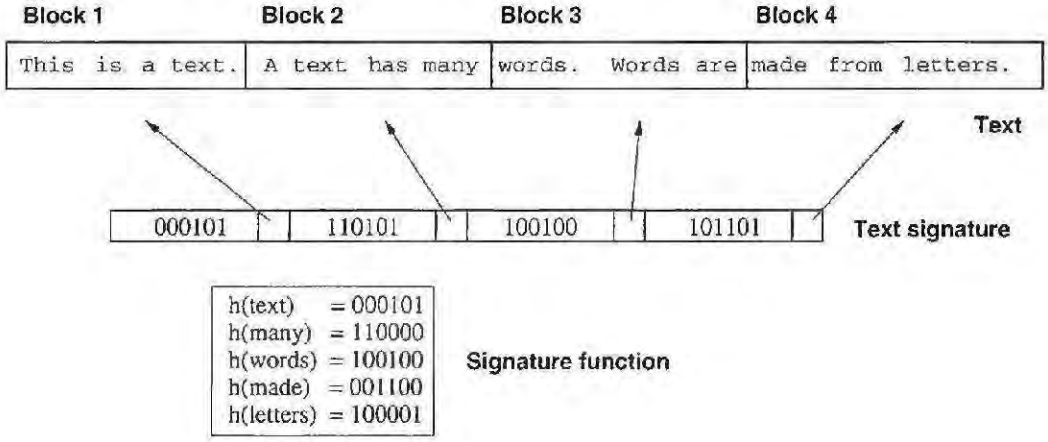


Figure 8.11 A signature file for our sample text cut into blocks.

random, the probability that a given bit of the mask is set in a word signature is $1-(1-1/B)^{b\ell} \approx 1-e^{-b\alpha}$. Hence, the probability that the ℓ random bits set in the query are also set in the mask of the text block is

$$(1-e^{-b\alpha})^{\alpha B}$$

which is minimized for $\alpha = \ln(2)/b$. The false drop probability under the optimal selection $\ell = B\ln(2)/b$ is $(1/2^{\ln(2)})^{B/b} = 1/2^{\ell}$.

Hence, a reasonable proportion B/b must be determined. The space overhead of the index is approximately $(1/80) \times (B/b)$ because B is measured in bits and b in words. Then, the false drop probability is a function of the overhead to pay. For instance, a 10% overhead implies a false drop probability close to 2%, while a 20% overhead errs with probability 0.046%. This error probability corresponds to the expected amount of sequential searching to perform while checking if a match is a false drop or not.

Searching

Searching a single word is carried out by hashing it to a bit mask W, and then comparing the bit masks B_i of all the text blocks. Whenever $(W \ \& \ B_i = W)$, where & is the bitwise AND, all the bits set in W are also set in B_i and therefore the text block *may* contain the word. Hence, for all candidate text blocks, an online traversal must be performed to verify if the word is actually there. This traversal cannot be avoided as in inverted files (except if the risk of a false drop is accepted).

No other types of patterns can be searched in this scheme. On the other hand, the scheme is more efficient to search phrases and reasonable proximity queries. This is because *all* the words must be present in a block in order for that block to hold the phrase or the proximity query. Hence, the bitwise OR of all the query masks is searched, so that *all* their bits must be present. This

reduces the probability of false drops. This is the only indexing scheme which improves in phrase searching.

Some care has to be exercised at block boundaries, however, to avoid missing a phrase which crosses a block limit. To allow searching phrases of j words or proximities of up to j words, consecutive blocks must overlap in j words.

If the blocks correspond to retrieval units, simple Boolean conjunctions involving words or phrases can also be improved by forcing all the relevant words to be in the block.

We were only able to find real performance estimates from 1992, run on a Sun 3/50 with local disk. Queries on a small 2.8 Mb database took 0.42 seconds. Extrapolating to today's technology, we find that the performance should be close to 20 Mb/sec (recall that it is linear time), and hence the example of 250 Mb of text would take 12 seconds, which is quite slow.

Construction

The construction of a signature file is rather easy. The text is simply cut in blocks, and for each block an entry of the signature file is generated. This entry is the bitwise OR of the signatures of all the words in the block.

Adding text is also easy, since it is only necessary to keep adding records to the signature file. Text deletion is carried out by deleting the appropriate bit masks.

Other storage proposals exist apart from storing all the bit masks in sequence. For instance, it is possible to make a different file for each bit of the mask, i.e. one file holding all the first bits, another file for all the second bits, etc. This reduces the disk times to search for a query, since only the files corresponding to the ℓ bits which are set in the query have to be traversed.

8.4 Boolean Queries

We now cover set manipulation algorithms. These algorithms are used when operating on sets of results, which is the case in Boolean queries. Boolean queries are described in Chapter 4, where the concept of *query syntax tree* is defined.

Once the leaves of the query syntax tree are solved (using the algorithms to find the documents containing the basic queries given), the relevant documents must be worked on by composition operators. Normally the search proceeds in three phases: the first phase determines which documents classify, the second determines the relevance of the classifying documents so as to present them appropriately to the user, and the final phase retrieves the exact positions of the matches to highlight them in those documents that the user actually wants to see.

This scheme avoids doing unnecessary work on documents which will not classify at last (first phase), or will not be read at last (second phase). However, some phases can be merged if doing the extra operations is not expensive. Some phases may not be present at all in some scenarios.

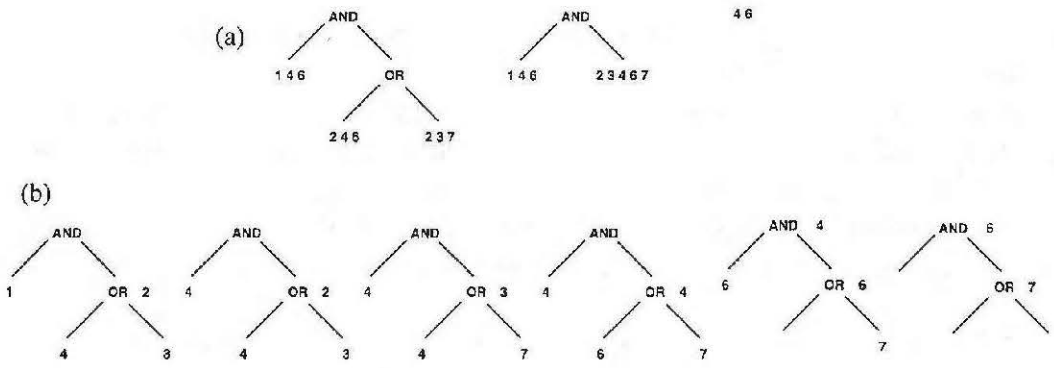


Figure 8.12 Processing the internal nodes of the query syntax tree. In (*a*) full evaluation is used. In (*b*) we show lazy evaluation in more detail.

Once the leaves of the query syntax tree find the classifying sets of documents, these sets are further operated by the internal nodes of the tree. It is possible to algebraically optimize the tree using identities such as $a\ OR\ (a\ AND\ b) = a$, for instance, or sharing common subexpressions, but we do not cover this issue here.

As all operations need to pair the same document in both their operands, it is good practice to keep the sets sorted, so that operations like intersection, union, etc. can proceed sequentially on both lists and also generate a sorted list. Other representations for sets not consisting of the list of matching documents (such as bit vectors) are also possible.

Under this scheme, it is possible to evaluate the syntax tree in *full* or *lazy* form. In the full evaluation form, both operands are first completely obtained and then the complete result is generated. In lazy evaluation, results are delivered only when required, and to obtain that result some data is recursively required to both operands.

Full evaluation allows some optimizations to be performed because the sizes of the results are known in advance (for instance, merging a very short list against a very long one can proceed by binary searching the elements of the short list in the long one). Lazy evaluation, on the other hand, allows the application to control when to do the work of obtaining new results, instead of blocking it for a long time. Hybrid schemes are possible, for example obtain all the leaves at once and then proceed in lazy form. This may be useful, for instance, to implement some optimizations or to ensure that all the accesses to the index are sequential (thus reducing disk seek times). Figure 8.12 illustrates this.

The complexity of solving these types of queries, apart from the cost of obtaining the results at the leaves, is normally linear in the total size of all the intermediate results. This is why this time may dominate the others, when there are huge intermediate results. This is more noticeable to the user when the final result is small.

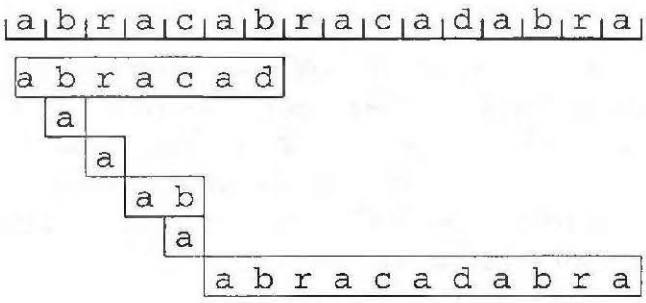


Figure 8.13 Brute-force search algorithm for the pattern 'abracadabra.' Squared areas show the comparisons performed.

8.5 Sequential Searching

We now cover the algorithms for text searching when no data structure has been built on the text. As shown, this is a basic part of some indexing techniques as well as the only option in some cases. We cover exact string matching in this section. Later we cover matching of more complex patterns. Our exposition is mainly conceptual and the implementation details are not shown (see the Bibliographic Discussion at the end of this chapter for more information).

The problem of exact string matching is: given a short pattern P of length m and a long text T of length n, find all the text positions where the pattern occurs. With minimal changes this problem subsumes many basic queries, such as word, prefix, suffix, and substring search.

This is a classical problem for which a wealth of solutions exists. We sketch the main algorithms, and leave aside a lot of the theoretical work that is not competitive in practice. For example, we do not include the Karp-Rabin algorithm, which is a nice application of hashing to string searching, but is not practical. We also briefly cover multipattern algorithms (that search many patterns at once), since a query may have many patterns and it may be more efficient to retrieve them all at once. Finally, we also mention how to do phrases and proximity searches.

We assume that the text and the pattern are sequences of characters drawn from an alphabet of size σ, whose first character is at position 1. The average-case analysis assumes random text and patterns.

8.5.1 Brute Force

The brute-force (BF) algorithm is the simplest possible one. It consists of merely trying all possible pattern positions in the text. For each such position, it verifies whether the pattern matches at that position. See Figure 8.13.

Since there are $O(n)$ text positions and each one is examined at $O(m)$ worst-case cost, the worst-case of brute-force searching is $O(mn)$. However, its average

case is $O(n)$ (since on random text a mismatch is found after $O(1)$ comparisons on average). This algorithm does not need any pattern preprocessing.

Many algorithms use a modification of this scheme. There is a *window* of length m which is slid over the text. It is *checked* whether the text in the window is equal to the pattern (if it is, the window position is reported as a match). Then, the window is *shifted* forward. The algorithms mainly differ in the way they check and shift the window.

8.5.2 Knuth-Morris-Pratt

The KMP algorithm was the first with linear worst-case behavior, although on average it is not much faster than BF. This algorithm also slides a window over the text. However, it does not try all window positions as BF does. Instead, it reuses information from previous checks.

After the window is checked, whether it matched the pattern or not, a number of pattern letters were compared to the text window, and they all matched except possibly the last one compared. Hence, when the window has to be shifted, there is a *prefix* of the pattern that matched the text. The algorithm takes advantage of this information to avoid trying window positions which can be deduced not to match.

The pattern is preprocessed in $O(m)$ time and space to build a table called *next*. The *next* table at position j says which is the longest proper prefix of $P_{1..j-1}$ which is also a suffix and the characters following prefix and suffix are different. Hence $j - next[j] + 1$ window positions can be safely skipped if the characters up to $j-1$ matched, and the j-th did not. For instance, when searching the word 'abracadabra,' if a text window matched up to 'abracab,' five positions can be safely skipped since $next[7] = 1$. Figure 8.14 shows an example.

The crucial observation is that this information depends only on the pattern, because if the text in the window matched up to position $j-1$, then that text is equal to the pattern.

The algorithm moves a window over the text and a pointer inside the window. Each time a character matches, the pointer is advanced (a match is reported if the pointer reaches the end of the window). Each time a character is not matched, the window is shifted forward in the text, to the position given by *next*, but the pointer position in the text does not change. Since at each text comparison the window or the pointer advance by at least one position, the algorithm performs at most $2n$ comparisons (and at least n).

The Aho-Corasick algorithm can be regarded as an extension of KMP in matching a set of patterns. The patterns are arranged in a trie-like data structure. Each trie node represents having matched a prefix of some pattern(s). The *next* function is replaced by a more general set of *failure* transitions. Those transitions go between nodes of the trie. A transition leaving from a node representing the prefix x leads to a node representing a prefix y, such that y is the longest prefix in the set of patterns which is also a proper suffix of x. Figure 8.15 illustrates this.

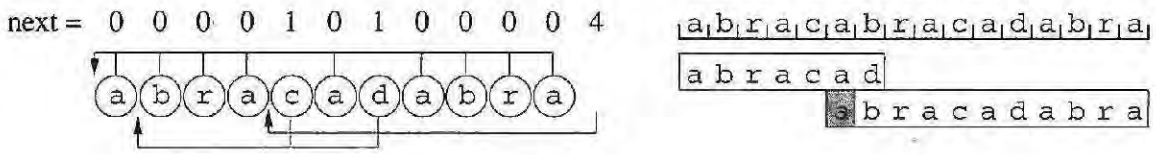


Figure 8.14 KMP algorithm searching 'abracadabra.' On the left, an illustration of the *next* function. Notice that after matching 'abracada' we do not try to match the last 'a' with the first one since what follows cannot be a 'b.' On the right, a search example. Grayed areas show the prefix information reused.

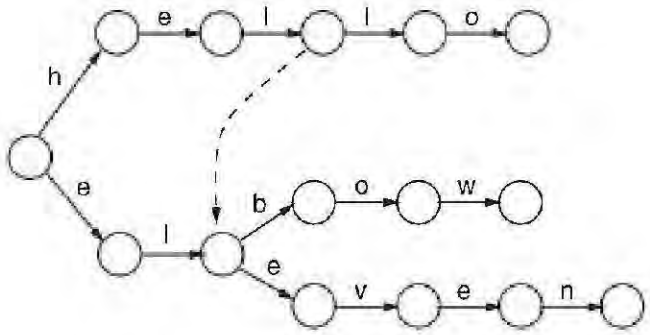


Figure 8.15 Aho-Corasick trie example for the set 'hello,' 'elbow' and 'eleven' showing only one of all the failure transitions.

This trie, together with its failure transitions, is built in $O(m)$ time and space (where m is the total length of all the patterns). Its search time is $O(n)$ no matter how many patterns are searched. Much as KMP, it makes at most $2n$ inspections.

8.5.3 Boyer-Moore Family

BM algorithms are based on the fact that the check inside the window can proceed backwards. When a match or mismatch is determined, a *suffix* of the pattern has been compared and found equal to the text in the window. This can be used in a way very similar to the *next* table of KMP, i.e. compute for every pattern position j the next-to-last occurrence of $P_{j..m}$ inside P. This is called the 'match heuristic.'

This is combined with what is called the 'occurrence heuristic.' It states that the text character that produced the mismatch (if a mismatch occurred) has to be aligned with the same character in the pattern after the shift. The heuristic which gives the longest shift is selected.

For instance, assume that 'abracadabra' is searched in a text which starts with 'abracababra.' After matching the suffix 'abra' the underlined text character 'b' will cause a mismatch. The match heuristic states that since 'abra' was matched a shift of 7 is safe. The occurrence heuristic states that since the underlined 'b' must match the pattern, a shift of 5 is safe. Hence, the pattern is

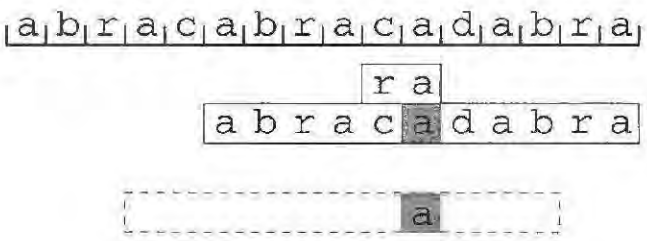


Figure 8.16 BM algorithm searching 'abracadabra.' Squared areas show the comparisons performed. Grayed areas have already been compared (but the algorithm compares them again). The dashed box shows the match heuristic, which was not chosen.

shifted by 7. See Figure 8.16.

The preprocessing time and space of this algorithm is $O(m+\sigma)$. Its search time is $O(n \log(m)/m)$ on average, which is 'sublinear' in the sense that not all characters are inspected. On the other hand, its worst case is $O(mn)$ (unlike KMP, the old suffix information is not kept to avoid further comparisons).

Further simplifications of the BM algorithm lead to some of the fastest algorithms on average. The Simplified BM algorithm uses only the occurrence heuristic. This obtains almost the same shifts in practice. The BM-Horspool (BMH) algorithm does the same, but it notices that it is not important any more that the check proceeds backwards, and uses the occurrence heuristic on the *last* character of the window instead of the one that caused the mismatch. This gives longer shifts on average. Finally, the BM-Sunday (BMS) algorithm modifies BMH by using the character *following* the last one, which improves the shift especially on short patterns.

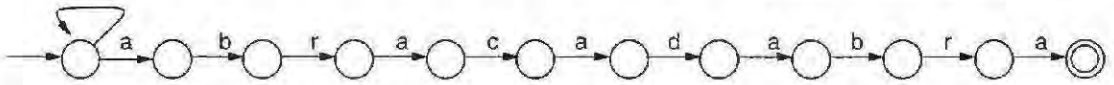
The Commentz-Walter algorithm is an extension of BM to multipattern search. It builds a trie on the reversed patterns, and instead of a backward window check, it enters into the trie with the window characters read backwards. A shift function is computed by a natural extension of BM. In general this algorithm improves over Aho-Corasick for not too many patterns.

8.5.4 Shift-Or

Shift-Or is based on *bit-parallelism*. This technique involves taking advantage of the intrinsic parallelism of the bit operations inside a computer word (of w bits). By cleverly using this fact, the number of operations that an algorithm performs can be cut by a factor of at most w. Since in current architectures w is 32 or 64, the speedup is very significant in practice.

The Shift-Or algorithm uses bit-parallelism to simulate the operation of a non-deterministic automaton that searches the pattern in the text (see Figure 8.17). As this automaton is simulated in time $O(mn)$, the Shift-Or algorithm achieves $O(mn/w)$ worst-case time (optimal speedup).

The algorithm first builds a table B which for each character stores a bit mask $b_m...b_1$. The mask in $B[c]$ has the i-th bit set to zero if and only if



$B[a] =$	1	0	0	1	0	1	0	1	0	0	1
$B[b] =$	0	1	0	0	0	0	0	0	1	0	0
$B[r] =$	0	0	1	0	0	0	0	0	0	1	0
$B[c] =$	0	0	0	0	1	0	0	0	0	0	0
$B[d] =$	0	0	0	0	0	0	1	0	0	0	0
$B[*] =$	0	0	0	0	0	0	0	0	0	0	0

Figure 8.17 Non-deterministic automaton that searches 'abracadabra,' and the associated B table. The initial self-loop matches any character. Each table column corresponds to an edge of the automaton.

$p_i = c$ (see Figure 8.17). The state of the search is kept in a machine word $D = d_m...d_1$, where d_i is zero whenever the state numbered i in Figure 8.17 is active. Therefore, a match is reported whenever d_m is zero. In the following, we use '|' to denote the bitwise OR and '&' to denote the bitwise AND.

D is set to all ones originally, and for each new text character T_j, D is updated using the formula

$$D' \leftarrow (D << 1) \mid B[T_j]$$

(where '<<' means shifting all the bits in D one position to the left and setting the rightmost bit to zero). It is not hard to relate the formula to the movement that occurs in the non-deterministic automaton for each new text character.

For patterns longer than the computer word (i.e., $m > w$), the algorithm uses $\lceil m/w \rceil$ computer words for the simulation (not all them are active all the time). The algorithm is $O(n)$ on average and the preprocessing is $O(m+\sigma)$ time and $O(\sigma)$ space.

It is easy to extend Shift-Or to handle classes of characters by manipulating the B table and keeping the search algorithm unchanged.

This paradigm also can search a large set of extended patterns, as well as multiple patterns (where the complexity is the same as before if we consider that m is the total length of all the patterns).

8.5.5 Suffix Automaton

The Backward DAWG matching (BDM) algorithm is based on a suffix automaton. A *suffix automaton* on a pattern P is an automaton that recognizes all the suffixes of P. The non-deterministic version of this automaton has a very regular structure and is shown in Figure 8.18.

The BDM algorithm converts this automaton to deterministic. The size and construction time of this automaton is $O(m)$. This is basically the preprocessing effort of the algorithm. Each path from the initial node to any internal

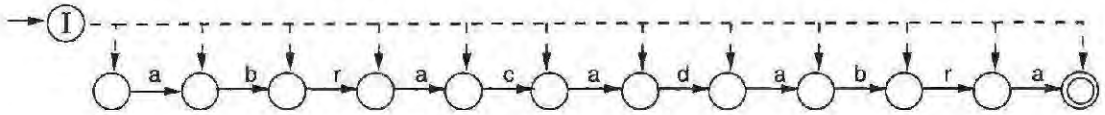


Figure 8.18 A non-deterministic suffix automaton. Dashed lines represent ε-transitions (i.e., they occur without consuming any input). I is the initial state of the automaton.

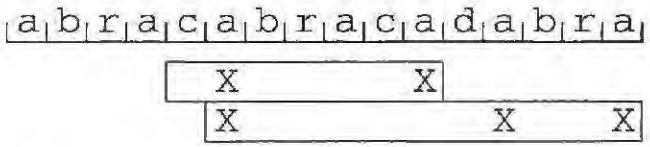


Figure 8.19 The BDM algorithm for the pattern 'abracadabra.' The rectangles represent elements compared to the text window. The Xs show the positions where a pattern prefix was recognized.

node represents a substring of the pattern. The final nodes represent pattern suffixes.

To search a pattern P, the suffix automaton of P^r (the reversed pattern) is built. The algorithm searches backwards inside the text window for a substring of the pattern P using the suffix automaton. Each time a terminal state is reached before hitting the beginning of the window, the position inside the window is remembered. This corresponds to finding a *prefix* of the pattern equal to a suffix of the window (since the reverse suffixes of P^r are the prefixes of P). The last prefix recognized backwards is the *longest* prefix of P in the window. A match is found if the complete window is read, while the check is abandoned when there is no transition to follow in the automaton. In either case, the window is shifted to align with the longest prefix recognized. See Figure 8.19.

This algorithm is $O(mn)$ time in the worst case and $O(n \log(m)/m)$ on average. There exists also a multipattern version of this algorithm called MultiBDM, which is the fastest for many patterns or very long patterns.

BDM rarely beats the best BM algorithms. However, a recent bit-parallel implementation called BNDM improves over BM in a wide range of cases. This algorithm simulates the non-deterministic suffix automaton using bit-parallelism. The algorithm supports some extended patterns and other applications mentioned in Shift-Or, while keeping more efficient than Shift-Or.

8.5.6 Practical Comparison

Figure 8.20 shows a practical comparison between string matching algorithms run on our reference machine. The values are correct within 5% of accuracy with a 95% confidence interval. We tested English text from the TREC collection, DNA (corresponding to 'h.influenzae') and random text uniformly generated over 64 letters. The patterns were randomly selected from the text except for random

text, where they were randomly generated. We tested over 10 Mb of text and measured CPU time. We tested short patterns on English and random text and long patterns on DNA, which are the typical cases.

We first analyze the case of random text, where except for very short patterns the clear winners are BNDM (the bit-parallel implementation of BDM) and the BMS (Sunday) algorithm. The more classical Boyer-Moore and BDM algorithms are also very close. Among the algorithms that do not improve with the pattern length, Shift-Or is the fastest, and KMP is much slower than the naive algorithm.

The picture is similar for English text, except that we have included the Agrep software in this comparison, which worked well only on English text. Agrep turns out to be much faster than others. This is not because of using a special algorithm (it uses a BM-family algorithm) but because the code is carefully optimized. This shows the importance of careful coding as well as using good algorithms, especially in text searching where a few operations per text character are performed.

Longer patterns are shown for a DNA text. BNDM is the fastest for moderate patterns, but since it does not improve with the length after $m > w$, the classical BDM finally obtains better times. They are much better than the Boyer-Moore family because the alphabet is small and the suffix automaton technique makes better use of the information on the pattern.

We have not shown the case of extended patterns, that is, where flexibility plays a role. For this case, BNDM is normally the fastest when it can be applied (e.g., it supports classes of characters but not wild cards), otherwise Shift-Or is the best option. Shift-Or is also the best option when the text must be accessed sequentially and it is not possible to skip characters.

8.5.7 Phrases and Proximity

If a sequence of words is searched to appear in the text exactly as in the pattern (i.e., with the same separators) the problem is similar to that of exact search of a single pattern, by just forgetting the fact that there are many words. If any separator between words is to be allowed, it is possible to arrange it using an extended pattern or regular expression search.

The best way to search a phrase element-wise is to search for the element which is less frequent or can be searched faster (both criteria normally match). For instance, longer patterns are better than shorter ones; allowing fewer errors is better than allowing more errors. Once such an element is found, the neighboring words are checked to see if a complete match is found.

A similar algorithm can be used to search a proximity query.

8.6 Pattern Matching

We present in this section the main techniques to deal with complex patterns. We divide it into two main groups: searching allowing errors and searching for extended patterns.

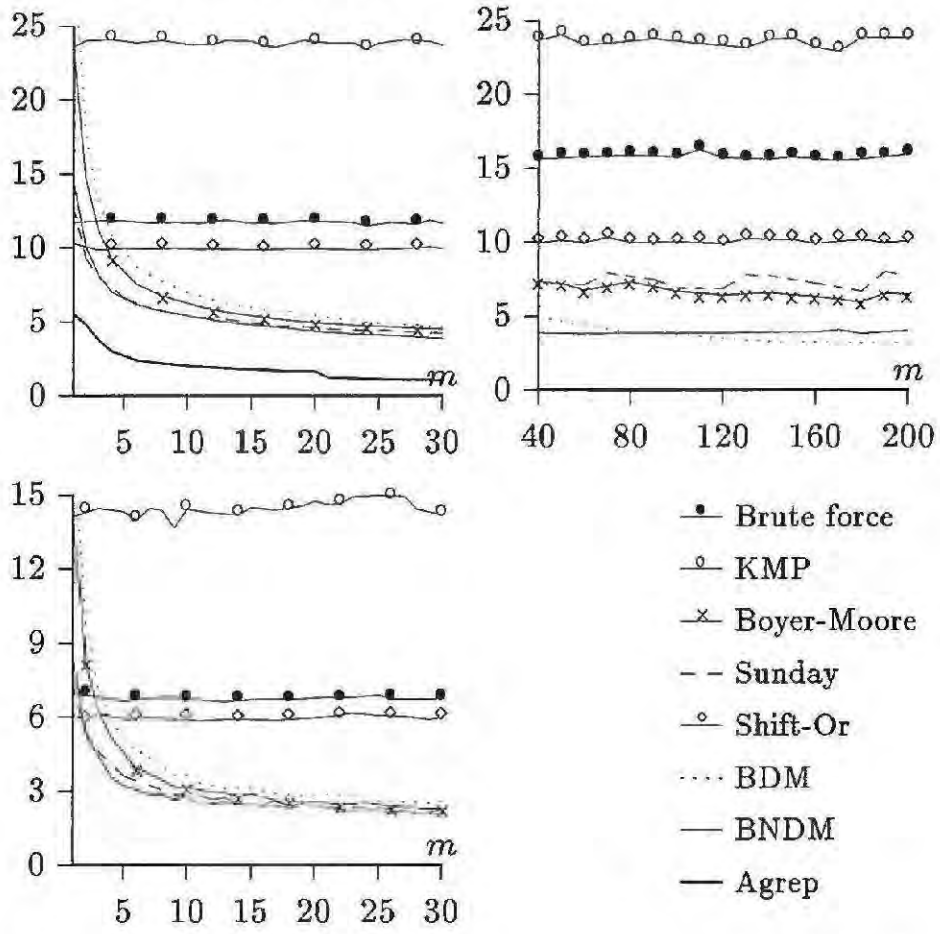


Figure 8.20 Practical comparison among algorithms. The upper left plot is for short patterns on English text. The upper right one is for long patterns on DNA. The lower plot is for short patterns on random text (on 64 letters). Times are in tenths of seconds per megabyte.

8.6.1 String Matching Allowing Errors

This problem (called 'approximate string matching') can be stated as follows: given a short pattern P of length m, a long text T of length n, and a maximum allowed number of errors k, find all the text positions where the pattern occurs with at most k errors. This statement corresponds to the Levenshtein distance. With minimal modifications it is adapted to searching whole words matching the pattern with k errors.

This problem is newer than exact string matching, although there are already a number of solutions. We sketch the main approaches.

Dynamic Programming

The classical solution to approximate string matching is based on dynamic programming. A matrix $C[0..m, 0..n]$ is filled column by column, where $C[i, j]$

represents the minimum number of errors needed to match $P_{1..i}$ to a suffix of $T_{1..j}$. This is computed as follows

$$\begin{aligned} C[0,j] &= 0 \\ C[i,0] &= i \\ C[i,j] &= \text{if } (P_i = T_j) \text{ then } C[i-1,j-1] \\ &\quad \text{else } 1 + \min(C[i-1,j], C[i,j-1], C[i-1,j-1]) \end{aligned}$$

where a match is reported at text positions j such that $C[m,j] \leq k$ (the final positions of the occurrences are reported).

Therefore, the algorithm is $O(mn)$ time. Since only the previous column of the matrix is needed, it can be implemented in $O(m)$ space. Its preprocessing time is $O(m)$. Figure 8.21 illustrates this algorithm.

In recent years several algorithms have been presented that achieve $O(kn)$ time in the worst case or even less in the average case, by taking advantage of the properties of the dynamic programming matrix (e.g., values in neighbor cells differ at most by one).

Automaton

It is interesting to note that the problem can be reduced to a non-deterministic finite automaton (NFA). Consider the NFA for $k = 2$ errors shown in Figure 8.22. Each row denotes the number of errors seen. The first one 0, the second one 1, and so on. Every column represents matching the pattern up to a given position. At each iteration, a new text character is read and the automaton changes its states. Horizontal arrows represent matching a character, vertical arrows represent insertions into the pattern, solid diagonal arrows represent replacements, and dashed diagonal arrows represent deletions in the pattern (they are ε-transitions). The automaton accepts a text position as the end of a match

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Figure 8.21 The dynamic programming algorithm search 'survey' in the text 'surgery' with two errors. Bold entries indicate matching positions.

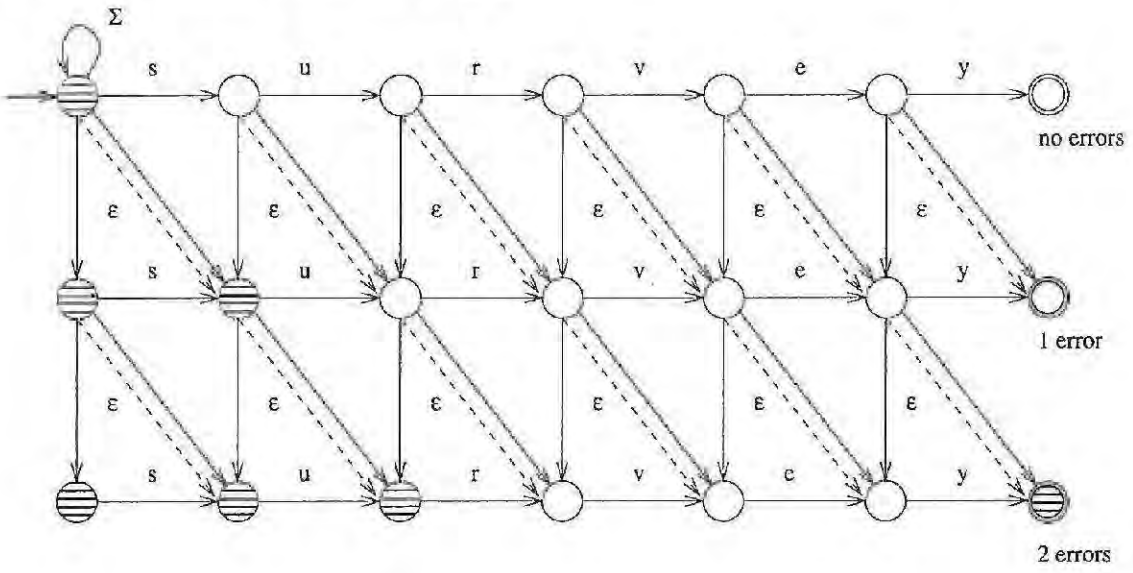


Figure 8.22 An NFA for approximate string matching of the pattern 'survey' with two errors. The shaded states are those active after reading the text 'surgery'. Unlabelled transitions match any character.

with k errors whenever the $(k+1)$-th rightmost state is active.

It is not hard to see that once a state in the automaton is active, all the states of the same column and higher rows are active too. Moreover, at a given text character, if we collect the smallest active rows at each column, we obtain the current column of the dynamic programming algorithm. Figure 8.22 illustrates this (compare the figure with Figure 8.21).

One solution is to make this automaton deterministic (DFA). Although the search phase is $O(n)$, the DFA can be huge. An alternative solution is based on bit-parallelism and is explained next.

Bit-Parallelism

Bit-parallelism has been used to parallelize the computation of the dynamic programming matrix (achieving average complexity $O(kn/w)$) and to parallelize the computation of the NFA (without converting it to deterministic), obtaining $O(kmn/w)$ time in the worst case. Such algorithms achieve $O(n)$ search time for short patterns and are currently the fastest ones in many cases, running at 6 to 10 Mb per second on our reference machine.

Filtering

Finally, other approaches first filter the text, reducing the area where dynamic programming needs to be used. These algorithms achieve 'sublinear' expected time in many cases for low error ratios (i.e., not all text characters are inspected,

$O(kn \log_\sigma(m)/m)$ is a typical figure), although the filtration is not effective for more errors. Filtration is based on the fact that some portions of the pattern must appear with no errors even in an approximate occurrence.

The fastest algorithm for low error levels is based on filtering: if the pattern is split into $k+1$ pieces, any approximate occurrence must contain at least one of the pieces with no errors, since k errors cannot alter all the $k+1$ pieces. Hence, the search begins with a multipattern exact search for the pieces and it later verifies the areas that may contain a match (using another algorithm).

8.6.2 Regular Expressions and Extended Patterns

General regular expressions are searched by building an automaton which finds all their occurrences in a text. This process first builds a non-deterministic finite automaton of size $O(m)$, where m is the length of the regular expression. The classical solution is to convert this automaton to deterministic form. A deterministic automaton can search any regular expression in $O(n)$ time. However, its size and construction time can be exponential in m, i.e. $O(m2^m)$. See Figure 8.23. Excluding preprocessing, this algorithm runs at 6 Mb/sec in the reference machine.

Recently the use of bit-parallelism has been proposed to avoid the construction of the deterministic automaton. The non-deterministic automaton is simulated instead. One bit per automaton state is used to represent whether the state is active or not. Due to the algorithm used to build the non-deterministic automaton, all the transitions move forward except for ε-transitions. The idea is that for each text character two steps are carried out. The first one moves forward, and the second one takes care of all the ε-transitions. A function E from bit masks to bit masks is precomputed so that all the corresponding bits are moved according to the ε-transitions. Since this function is very large (i.e., 2^m entries) its domain is split in many functions from 8- or 16-bit submasks to m-bit masks. This is possible because $E(B_1...B_j) = E(B_1)|...|E(B_j)$, where B_i

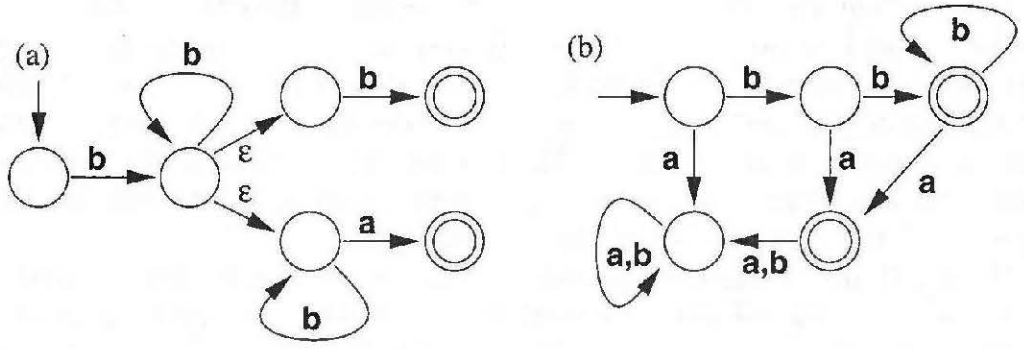


Figure 8.23 The non-deterministic (a) and deterministic (b) automata for the regular expression `b b*  (b | b*a)`.

are the submasks. Hence, the scheme performs $\lceil m/8 \rceil$ or $\lceil m/16 \rceil$ operations per text character and needs $\lceil m/8 \rceil 2^8 \lceil m/w \rceil$ or $\lceil m/16 \rceil 2^{16} \lceil m/w \rceil$ machine words of memory.

Extended patterns can be rephrased as regular expressions and solved as before. However, in many cases it is more efficient to give them a specialized solution, as we saw for the extensions of exact searching (bit-parallel algorithms). Moreover, extended patterns can be combined with approximate search for maximum flexibility. In general, the bit-parallel approach is the best equipped to deal with extended patterns.

Real times for regular expressions and extended pattern searching using this technique are between 2–8 Mb/sec.

8.6.3 Pattern Matching Using Indices

We end this section by explaining how the indexing techniques we presented for simple searching of words can in fact be extended to search for more complex patterns.

Inverted Files

As inverted files are word-oriented, other types of queries such as suffix or substring queries, searching allowing errors and regular expressions, are solved by a *sequential* (i.e., online) search over the vocabulary. This is not too bad since the size of the vocabulary is small with respect to the text size.

After either type of search, a list of vocabulary words that matched the query is obtained. All their lists of occurrences are now *merged* to retrieve a list of documents and (if required) the matching text positions. If block addressing is used and the positions are required or the blocks do not coincide with the retrieval unit, the search must be completed with a sequential search over the blocks.

Notice that an inverted index is word-oriented. Because of that it is not surprising that it is not able to efficiently find approximate matches or regular expressions that span many words. This is a restriction of this scheme. Variations that are not subject to this restriction have been proposed for languages which do not have a clear concept of word, like Finnish. They collect text *samples* or *n-grams*, which are fixed-length strings picked at regular text intervals. Searching is in general more powerful but more expensive.

In a full-inverted index, search times for simple words allowing errors on 250 Mb of text took our reference machine from 0.6 to 0.85 seconds, while very complex expressions on extended patterns took from 0.8 to 3 seconds. As a comparison, the same collection cut in blocks of 1 Mb size takes more than 8 seconds for an approximate search with one error and more than 20 for two errors.

Suffix Trees and Suffix Arrays

If the suffix tree indexes all text positions it can search for words, prefixes, suffixes and substrings with the same search algorithm and cost described for word search. However, indexing all positions makes the index 10 to 20 times the text size for suffix trees.

Range queries are easily solved too, by just searching both extremes in the trie and then collecting all the leaves which lie in the middle. In this case the cost is the height of the tree, which is $O(\log n)$ on average (excluding the tasks of collecting and sorting the leaves).

Regular expressions can be searched in the suffix tree. The algorithm simply simulates sequential searching of the regular expression. It begins at the root, since any possible match starts there too. For each child of the current node labeled by the character c, it assumes that the next text character is c and recursively enters into that subtree. This is done for each of the children of the current node. The search stops only when the automaton has no transition to follow. It has been shown that for random text only $O(n^{\alpha}\text{polylog}(n))$ nodes are traversed (for $0 < \alpha < 1$ dependent on the regular expression). Hence, the search time is sublinear for regular expressions *without* the restriction that they must occur inside a word. Extended patterns can be searched in the same way by taking them as regular expressions.

Unrestricted approximate string matching is also possible using the same idea. We present a simplified version here. Imagine that the search is online and traverse the tree recursively as before. Since all suffixes start at the root, any match starts at the root too, and therefore do not allow the match to start later. The search will automatically stop at depth $m + k$ at most (since at that point more than k errors have occurred). This implies constant search time if n is large enough (albeit exponential on m and k). Other problems such as approximate search of extended patterns can be solved in the same way, using the appropriate online algorithm.

Suffix trees are able to perform other complex searches that we have not considered in our query language (see Chapter 4). These are specialized operations which are useful in specific areas. Some examples are: find the longest substring in the text that appears more than once, find the most common substring of a fixed size, etc.

If a suffix array indexes all text positions, *any* algorithm that works on suffix trees at $C(n)$ cost will work on suffix arrays at $O(C(n) \log n)$ cost. This is because the operations performed on the suffix tree consist of descending to a child node, which is done in $O(1)$ time. This operation can be simulated in the suffix array in $O(\log n)$ time by binary searching the new boundaries (each suffix tree node corresponds to a string, which can be mapped to the suffix array interval holding all suffixes starting with that string). Some patterns can be searched directly in the suffix array in $O(\log n)$ total search time without simulating the suffix tree. These are: word, prefix, suffix and subword search, as well as range search.

However, again, indexing all text positions normally makes the suffix array

size four times or more the text size. A different alternative for suffix arrays is to index only word beginnings and to use a vocabulary supra-index, using the same search algorithms used for the inverted lists.

8.7 Structural Queries

The algorithms to search on structured text (see Chapter 4) are largely dependent on each model. We extract their common features in this section.

A first concern about this problem is how to store the structural information. Some implementations build an ad hoc index to store the structure. This is potentially more efficient and independent of any consideration about the text. However, it requires extra development and maintenance effort.

Other techniques assume that the structure is marked in the text using 'tags' (i.e., strings that identify the structural elements). This is the case with HTML text but not the case with C code where the marks are implicit and are inherent to C. The technique relies on the same index to query content (such as inverted files), using it to index and search those tags as if they were words. In many cases this is as efficient as an ad hoc index, and its integration into an existing text database is simpler. Moreover, it is possible to define the structure dynamically, since the appropriate tags can be selected at search time. For that goal, inverted files are better since they naturally deliver the results in text order, which makes the structure information easier to obtain. On the other hand, some queries such as direct ancestry are hard to answer without an ad hoc index.

Once the content and structural elements have been found by using some index, a set of answers is generated. The models allow further operations to be applied on those answers, such as 'select all areas in the left-hand argument which contain an area of the right-hand argument.' This is in general solved in a way very similar to the set manipulation techniques already explained in section 8.4. However, the operations tend to be more complex, and it is not always possible to find an evaluation algorithm which has linear time with respect to the size of the intermediate results.

It is worth mentioning that some models use completely different algorithms, such as exhaustive search techniques for tree pattern matching. Those problems are NP-complete in many cases.

8.8 Compression

In this section we discuss the issues of searching compressed text directly and of searching compressed indices. Compression is important when available storage is a limiting factor, as is the case of indexing the Web.

Searching and compression were traditionally regarded as exclusive operations. Texts which were not to be searched could be compressed, and to search

a compressed text it had to be decompressed first. In recent years, very efficient compression techniques have appeared that allow searching directly in the compressed text. Moreover, the search performance is *improved*, since the CPU times are similar but the disk times are largely reduced. This leads to a win-win situation.

Discussion on how common text and lists of numbers can be compressed has been covered in Chapter 7.

8.8.1 Sequential Searching

A few approaches to directly searching compressed text exist. One of the most successful techniques in practice relies on Huffman coding taking words as symbols. That is, consider each different text word as a symbol, count their frequencies, and generate a Huffman codefor the words. Then, compress the text by replacing each word with its code. To improve compression/decompression efficiency, the Huffman code uses an alphabet of bytes instead of bits. This scheme compresses faster and better than known commercial systems, even those based on Ziv-Lempel coding.

Since Huffman coding needs to store the codes of each symbol, this scheme has to store the whole vocabulary of the text, i.e. the list of all different text words. This is fully exploited to efficiently search complex queries. Although according to Heaps' law the vocabulary (i.e., the alphabet) grows as $O(n^\beta)$ for $0 < \beta < 1$, the generalized Zipf's law shows that the distribution is skewed enough so that the entropy remains constant (i.e., the compression ratio will not degrade as the text grows). Those laws are explained in Chapter 6.

Any single-word or pattern query is first searched in the vocabulary. Some queries can be binary searched, while others such as approximate searching or regular expression searching must traverse sequentially all the vocabulary. This vocabulary is rather small compared to the text size, thanks to Heaps' law. Notice that this process is exactly the same as the vocabulary searching performed by inverted indices, either for simple or complex pattern matching.

Once that search is complete, the list of different words that match the query is obtained. The Huffman codes of all those words are collected and they are searched in the compressed text. One alternative is to traverse byte-wise the compressed text and traverse the Huffman decoding tree in synchronization, so that each time that a leaf is reached, it is checked whether the leaf (i.e., word) was marked as 'matching' the query or not. This is illustrated in Figure 8.24. Boyer-Moore filtering can be used to speed up the search.

Solving phrases is a little more difficult. Each element is searched in the vocabulary. For each word of the vocabulary we define a bit mask. We set the i-th bit in the mask of all words which match with the i-th element of the phrase query. This is used together with the Shift-Or algorithm. The text is traversed byte-wise, and only when a leaf is reached, does the Shift-Or algorithm consider that a new text symbol has been read, whose bit mask is that of the leaf (see Figure 8.24). This algorithm is surprisingly simple and efficient.

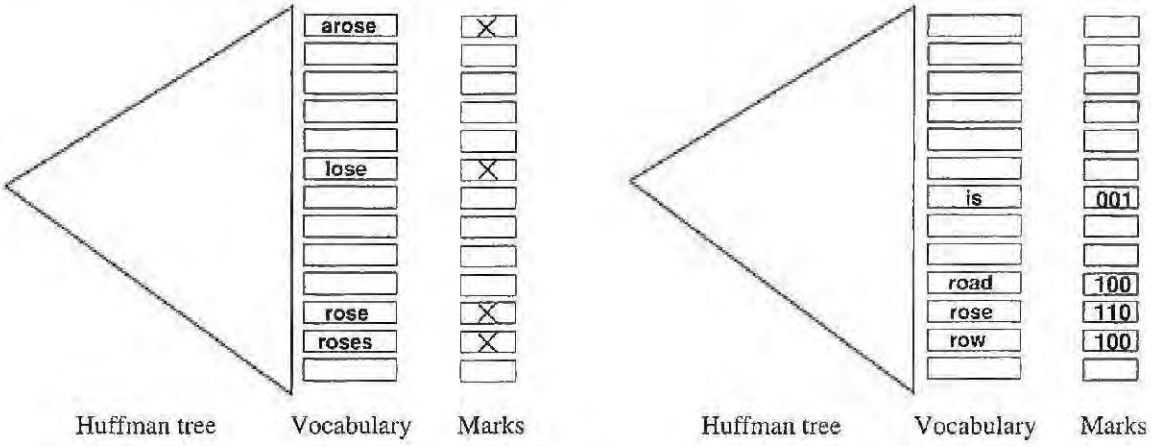


Figure 8.24 On the left, searching for the simple pattern '`rose`' allowing one error. On the right, searching for the phrase '`ro* rose is`,' where '`ro*`' represents a prefix search.

This scheme is especially fast when it comes to solving a complex query (regular expression, extended pattern, approximate search, etc.) that would be slow with a normal algorithm. This is because the complex search is done only in the small vocabulary, after which the algorithm is largely insensitive to the complexity of the originating query. Its CPU times for a simple pattern are slightly higher than those of Agrep (briefly described in section 8.5.6). However, if the I/O times are considered, compressed searching is faster than all the online algorithms. For complex queries, this scheme is unbeaten by far.

On the reference machine, the CPU times are 14 Mb/sec for any query, while for simple queries this improves to 18 Mb/sec if the speedup technique is used. Agrep, on the other hand, runs at 15 Mb/sec on simple searches and at 1–4 Mb/sec for complex ones. Moreover, I/O times are reduced to one third on the compressed text.

8.8.2 Compressed Indices

Inverted Files

Inverted files are quite amenable to compression. This is because the lists of occurrences are in increasing order of text position. Therefore, an obvious choice is to represent the differences between the previous position and the current one. These differences can be represented using less space by using techniques that favor small numbers (see Chapter 7). Notice that, the longer the lists, the smaller the differences. Reductions in 90% for block-addressing indices with blocks of 1 Kb size have been reported.

It is important to notice that compression does not necessarily degrade time performance. Most of the time spent in answering a query is in the disk transfer. Keeping the index compressed allows the transference of less data, and it may be worth the CPU work of decompressing. Notice also that the lists of

occurrences are normally traversed in a sequential manner, which is not affected by a differential compression. Query times on compressed or decompressed indices are reported to be roughly similar.

The text can also be compressed independently of the index. The text will be decompressed only to display it, or to traverse it in case of block addressing. Notice in particular that the online search technique described for compressed text in section 8.8.1 uses a vocabulary. It is possible to integrate both techniques (compression and indexing) such that they share the same vocabulary for both tasks and they do not decompress the text to index or to search.

Suffix Trees and Suffix Arrays

Some efforts to compress suffix trees have been pursued. Important reductions of the space requirements have been obtained at the cost of more expensive searching. However, the reduced space requirements happen to be similar to those of uncompressed suffix arrays, which impose much smaller performance penalties.

Suffix arrays are very hard to compress further. This is because they represent an almost perfectly random permutation of the pointers to the text.

However, the subject of building suffix arrays on compressed text has been pursued. Apart from reduced space requirements (the index plus the compressed text take less space than the uncompressed text), the main advantage is that both index construction and querying almost double their performance. Construction is faster because more compressed text fits in the same memory space, and therefore fewer text blocks are needed. Searching is faster because a large part of the search time is spent in disk seek operations over the text area to compare suffixes. If the text is smaller, the seeks reduce proportionally.

A compression technique very similar to that shown in section 8.8.1 is used. However, the Huffman code on words is replaced by a Hu-Tucker coding. The Hu-Tucker code respects the lexicographical relationships between the words, and therefore direct binary search over the compressed text is possible (this is necessary at construction and search time). This code is suboptimal by a very small percentage (2–3% in practice, with an analytical upper bound of 5%).

Indexing times for 250 Mb of text on the reference machine are close to 1.6 Mb/min if compression is used, while query times are reduced to 0.5 seconds in total and 0.3 seconds for the text alone. Supra-indices should reduce the total search time to 0.15 seconds.

Signature Files

There are many alternative ways to compress signature files. All of them are based on the fact that only a few bits are set in the whole file. It is then possible

to use efficient methods to code the bits which are not set, for instance run-length encoding. Different considerations arise if the file is stored as a sequence of bit masks or with one file per bit of the mask. They allow us to reduce space and hence disk times, or alternatively to increase B (so as to reduce the false drop probability) keeping the same space overhead. Compression ratios near 70% are reported.

8.9 Trends and Research Issues

In this chapter we covered extensively the current techniques of dealing with text retrieval. We first covered indices and then online searching. We then reviewed set manipulation, complex pattern matching and finally considered compression techniques. Figure 8.25 summarizes the tradeoff between the space needed for the index and the time to search one single word.

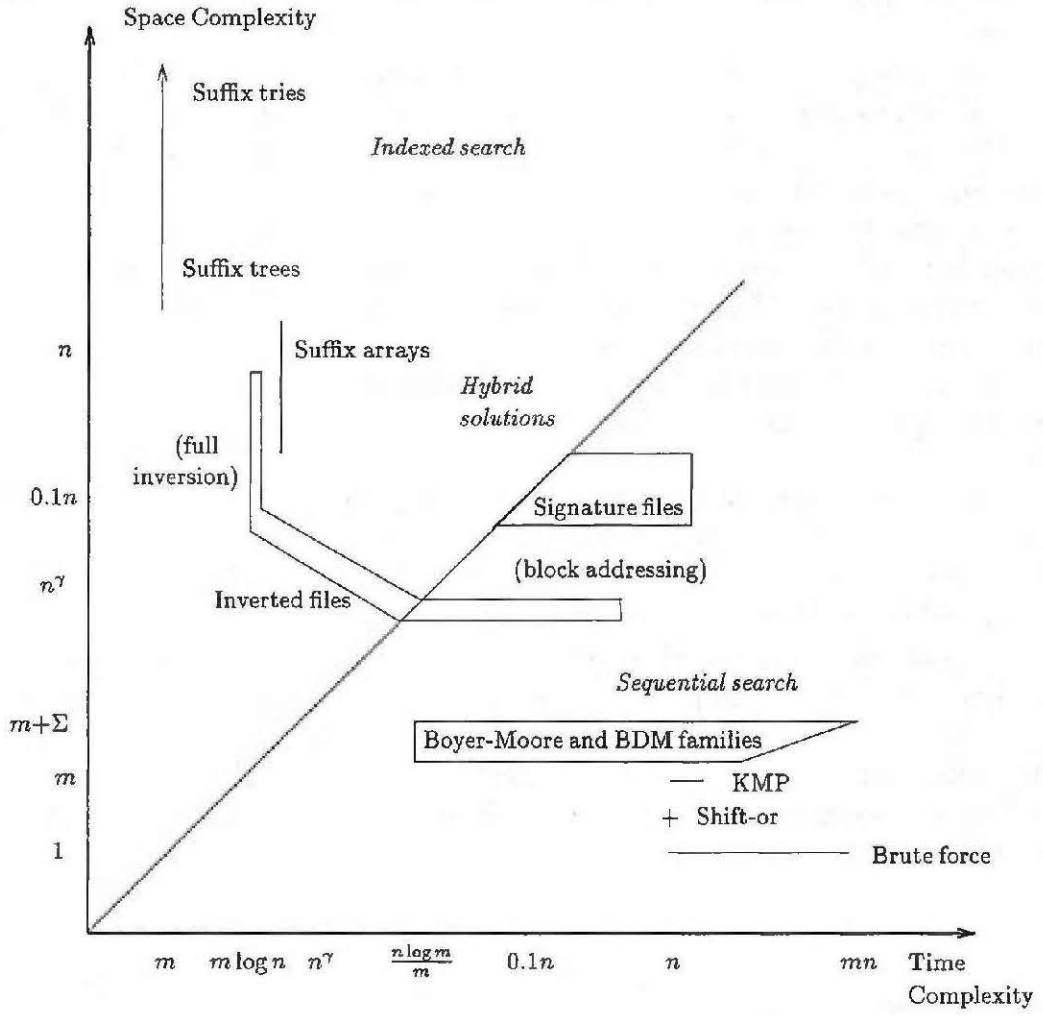


Figure 8.25 Tradeoff of index space versus word searching time.

Probably the most adequate indexing technique in practice is the inverted file. As we have shown throughout the chapter, many hidden details in other structures make them harder to use and less efficient in practice, as well as less flexible for dealing with new types of queries. These structures, however, still find application in restricted areas such as genetic databases (for suffix trees and arrays, for the relatively small texts used and their need to pose specialized queries) or some office systems (for signature files, because the text is rarely queried in fact).

The main trends in indexing and searching textual databases today are

- **Text collections are becoming huge**. This poses more demanding requirements at all levels, and solutions previously affordable are not any more. On the other hand, the speed of the processors and the relative slowness of external devices have changed what a few years ago were reasonable options (e.g., it is better to keep a text compressed because reading less text from disk and decompressing in main memory pays off).
- **Searching is becoming more complex**. As the text databases grow and become more heterogeneous and error-prone, enhanced query facilities are required, such as exploiting the text structure or allowing errors in the text. Good support for extended queries is becoming important in the evaluation of a text retrieval system.
- **Compression is becoming a star in the field**. Because of the changes mentioned in the time cost of processors and external devices, and because of new developments in the area, text retrieval and compression are no longer regarded as disjoint activities. Direct indexing and searching on compressed text provides better (sometimes much better) time performance and less space overhead at the same time. Other techniques such as block addressing trade space for processor time.

8.10 Bibliographic Discussion

A detailed explanation of a full inverted index and its construction and querying process can be found in [26]. This work also includes an analysis of the algorithms on inverted lists using the distribution of natural language. The in-place construction is described in [572]. Another construction algorithm is presented in [341].

The idea of block addressing inverted indices was first presented in a system called Glimpse [540], which also first exposed the idea of performing complex pattern matching using the vocabulary of the inverted index. Block addressing indices are analyzed in [42], where some performance improvements are proposed. The variant that indexes sequences instead of words has been implemented in a system called Grampse, which is described in [497].

Suffix arrays were presented in [538] together with the algorithm to build them in $O(n \log n)$ character comparisons. They were independently discovered

by [309] under the name of 'PAT arrays.' The algorithm to build large suffix arrays is presented in [311]. The use of supra-indices over suffix array is proposed in [37], while the modified binary search techniques to reduce disk seek time are presented in [56]. The linear-time construction of suffix trees is described in [780].

The material on signature files is based on [243]. The different alternative ways of storing the signature file are explained in [242].

The original references for the sequential search algorithms are: KMP [447], BM [110], BMH [376], BMS [751], Shift-Or [39], BDM [205] and BNDM [592]. The multipattern versions are found in [9, 179], and MultiBDM in [196]. Many enhancements of bit-parallelism to support extended patterns and allow errors are presented in [837]. Many ideas from that paper were implemented in a widely distributed software for online searching called Agrep [836].

The reader interested in more details about sequential searching algorithms may look for the original references or in good books on algorithms such as [310, 196].

One source for the classical solution to approximate string matching is [716]. An $O(kn)$ worst-case algorithm is described in [480]. The use of a DFA is proposed in [781]. The bit-parallel approach to this problem started in [837], although currently the fastest bit-parallel algorithms are [583] and [43]. Among all the filtering algorithms, the fastest one in practice is based on an idea presented in [837], later enhanced in [45], and finally implemented in [43].

A good source from which to learn about regular expressions and building a DFA is [375]. The bit-parallel implementation of the NFA is explained in [837]. Regular expression searching on suffix trees is described in [40], while searching allowing errors is presented in [779].

The Huffman coding was first presented in [386], while the word-oriented alternative is proposed in [571]. Sequential searching on text compressed using that technique is described in [577]. Compression used in combination with inverted files is described in [850], with suffix trees in [430], with suffix arrays in [575], and with signature files in [243, 242]. A good general reference on compression is [78].

Chapter 9
Parallel and Distributed IR

by Eric Brown

9.1 Introduction

The volume of electronic text available online today is staggering. By many accounts, the World Wide Web alone contains over 200 million pages of text, comprising nearly 500 gigabytes of data. Moreover, the Web (see Chapter 13) has been growing at an exponential rate, nearly doubling in size every six months. Large information service providers, such as LEXIS-NEXIS (see Chapter 14), have amassed document databases that reach into the terabytes. On a slightly smaller scale, the largest corporate intranets now contain over a million Web pages. Even private collections of online documents stored on personal computers are growing larger as disk space becomes cheaper and electronic content becomes easier to produce, download, and store.

As document collections grow larger, they become more expensive to manage with an information retrieval system. Searching and indexing costs grow with the size of the underlying document collection; larger document collections invariably result in longer response times. As more documents are added to the system, performance may deteriorate to the point where the system is no longer usable. Furthermore, the economic survival of commercial systems and Web search engines depends on their ability to provide high query processing rates. In fact, most of a Web search company's gross income comes from selling 'advertising impressions' (advertising banners displayed at the user's screen) whose number is proportional to the number of query requests attended.

To support the demanding requirements of modern search environments, we must turn to alternative architectures and algorithms. In this chapter we explore parallel and distributed information retrieval techniques. The application of parallelism can greatly enhance our ability to scale traditional information retrieval algorithms and support larger and larger document collections.

We continue this introduction with a review of parallel computing and parallel program performance measures. In section 9.2 we explore techniques for

implementing information retrieval algorithms on parallel platforms, including inverted file and signature file methods. In section 9.3, we turn to distributed information retrieval and approaches to collection partitioning, source selection, and distributed results merging (often called collection fusion). We discuss future trends in section 9.4, and conclude with a bibliographic discussion in section 9.5.

9.1.1 Parallel Computing

Parallel computing is the simultaneous application of multiple processors to solve a single problem, where each processor works on a different part of the problem. With parallel computing, the overall time required to solve the problem can be reduced to the amount of time required by the longest running part. As long as the problem can be further decomposed into more parts that will run in parallel, we can add more processors to the system, reduce the time required to solve the problem, and scale up to larger problems.

Processors can be combined in a variety of ways to form parallel architectures. Flynn [259] has defined a commonly used taxonomy of parallel architectures based on the number of the instruction and data streams in the architecture. The taxonomy includes four classes:

- **SISD** single instruction stream, single data stream
- **SIMD** single instruction stream, multiple data stream
- **MISD** multiple instruction stream, single data stream
- **MIMD** multiple instruction stream, multiple data stream.

The SISD class includes the traditional von Neumann [134] computer running sequential programs, e.g., uniprocessor personal computers. SIMD computers consist of N processors operating on N data streams, with each processor executing the same instruction at the same time. Machines in this class are often massively parallel computers with many relatively simple processors, a communication network between the processors, and a control unit that supervises the synchronous operation of the processors, e.g., the Thinking Machines CM-2. The processors may use shared memory, or each processor may have its own local memory. Sequential programs require significant modification to make effective use of a SIMD architecture, and not all problems lend themselves to a SIMD implementation.

MISD computers use N processors operating on a single data stream in shared memory. Each processor executes its own instruction stream, such that multiple operations are performed simultaneously on the same data item. MISD architectures are relatively rare. Systolic arrays are the best known example.

MIMD is the most general and most popular class of parallel architectures. A MIMD computer contains N processors, N instruction streams, and N data streams. The processors are similar to those used in a SISD computer; each

processor has its own control unit, processing unit, and local memory.† MIMD systems usually include shared memory or a communication network that connects the processors to each other. The processors can work on separate, unrelated tasks, or they can cooperate to solve a single task, providing a great deal of flexibility. MIMD systems with a high degree of processor interaction are called *tightly coupled*, while systems with a low degree of processor interaction are *loosely coupled*. Examples of MIMD systems include multiprocessor PC servers, symmetric multiprocessors (SMPs) such as the Sun HPC Server, and scalable parallel processors such as the IBM SP2.

Although MIMD typically refers to a single, self-contained parallel computer using two or more of the same kind of processor, MIMD also characterizes *distributed computing* architectures. In distributed computing, multiple computers connected by a local or wide area network cooperate to solve a single problem. Even though the coupling between processors is very loose in a distributed computing environment, the basic components of the MIMD architecture remain. Each computer contains a processor, control unit, and local memory, and the local or wide area network forms the communication network between the processors.

The main difference between a MIMD parallel computer and a distributed computing environment is the cost of interprocessor communication, which is considerably higher in a distributed computing environment. As such, distributed programs are usually coarse grained, while programs running on a single parallel computer tend to be finer grained. Granularity refers to the amount of computation relative to the amount of communication performed by the program. Coarse-grained programs perform large amounts of computation relative to communication; fine-grained programs perform large amounts of communication relative to computation. Of course, an application may use different levels of granularity at different times to solve a given problem.

9.1.2 Performance Measures

When we employ parallel computing, we usually want to know what sort of performance improvement we are obtaining over a comparable sequential program running on a uniprocessor. A number of metrics are available to measure the performance of a parallel algorithm. One such measure is the speedup obtained with the parallel algorithm relative to the best available sequential algorithm for solving the same problem, defined as:

$$S = \frac{\text{Running time of best available sequential algorithm}}{\text{Running time of parallel algorithm}}$$

† The processors used in a MIMD system may be identical to those used in SISD systems, or they may provide additional functionality, such as hardware cache coherence for shared memory.

Ideally, when running a parallel algorithm on N processors, we would obtain perfect speedup, or $S = N$. In practice, perfect speedup is unattainable either because the problem cannot be decomposed into N equal subtasks, the parallel architecture imposes control overheads (e.g., scheduling or synchronization), or the problem contains an inherently sequential component. Amdahl's law [18] states that the maximal speedup obtainable for a given problem is related to f, the fraction of the problem that must be computed sequentially. The relationship is given by:

$$S \leq \frac{1}{f + (1-f)/N} \leq \frac{1}{f}$$

Another measure of parallel algorithm performance is efficiency, given by:

$$\phi = \frac{S}{N}$$

where S is speedup and N is the number of processors. Ideal efficiency occurs when $\phi = 1$ and no processor is ever idle or performs unnecessary work. As with perfect speedup, ideal efficiency is unattainable in practice.

Ultimately, the performance improvement of a parallel program over a sequential program should be viewed in terms of the reduction in real time required to complete the processing task combined with the additional monetary cost associated with the parallel hardware required to run the parallel program. This gives the best overall picture of parallel program performance and cost effectiveness.

9.2 Parallel IR

9.2.1 Introduction

We can approach the development of parallel information retrieval algorithms from two different directions. One possibility is to develop new retrieval strategies that directly lend themselves to parallel implementation. For example, a text search procedure can be built on top of a neural network. Neural networks (see Chapter 2) are modeled after the human brain and solve problems using a large number of nodes (neurons), each of which has a set of inputs, a threshold, and an output. The output of one node is connected to the input of one or more other nodes, with the boundaries of the network defining the initial input and final output of the system. A node's output value is determined by a weighted function of the node's inputs and threshold. A training procedure is used to learn appropriate settings for the weights and thresholds in the network. Computation proceeds by applying input values to the network, computing each active node's output value, and conditioning these values through the network until the final output values are obtained. Neural networks naturally lend themselves to parallel implementation on SIMD hardware. The challenge with this approach is to

define the retrieval task in such a way that it maps well onto the computational paradigm.

The other possibility is to adapt existing, well studied information retrieval algorithms to parallel processing. This is the approach that we will consider throughout the rest of this chapter. The modifications required to adapt an existing algorithm to parallel implementation depend on the target parallel platform. We will investigate techniques for applying a number of retrieval algorithms to both MIMD and SIMD architectures. Since parallel information retrieval is still very much an active research area, few approaches have fallen out as accepted standard techniques. We will, therefore, present a sampling of the work that has been done and avoid preferring one technique over another.

9.2.2 MIMD Architectures

MIMD architectures offer a great deal of flexibility in how parallelism is defined and exploited to solve a problem. The simplest way in which a retrieval system can exploit a MIMD computer is through the use of *multitasking*. Each of the processors in the parallel computer runs a separate, independent search engine. The search engines do not cooperate to process individual queries, but they may share code libraries and data cached by the file system or loaded into shared memory. The submission of user queries to the search engines is managed by a broker, which accepts search requests from the end users and distributes the requests among the available search engines. This is depicted in Figure 9.1. As more processors are added to the system, more search engines may be run and more search requests may be processed in parallel, increasing the throughput of the system. Note, however, that the response time of individual queries remains unchanged.

In spite of the simplicity of this approach, care must be taken to properly balance the hardware resources on the system. In particular, as the number of processors grows, so must the number of disks and I/O channels. Unless the entire retrieval index fits in main memory, the search processes running on the different processors will perform I/O and compete for disk access. A bottleneck at the disk will be disastrous for performance and could eliminate the throughput gains anticipated from the addition of more processors.

In addition to adding more disks to the computer, the system administrator must properly distribute the index data over the disks. Disk contention will remain as long as two search processes need to access index data stored on the same disk. At one extreme, replicating the entire index on each disk eliminates disk contention at the cost of increased storage requirements and update complexity. Alternatively, the system administrator may partition and replicate index data across the disks according to profile information; heavily accessed data is replicated while less frequently accessed data is distributed randomly. Yet another approach is to install a disk array, or RAID [165], and let the operating system handle partitioning the index. Disk arrays can provide low latency and high throughput disk access by striping files across many disks.

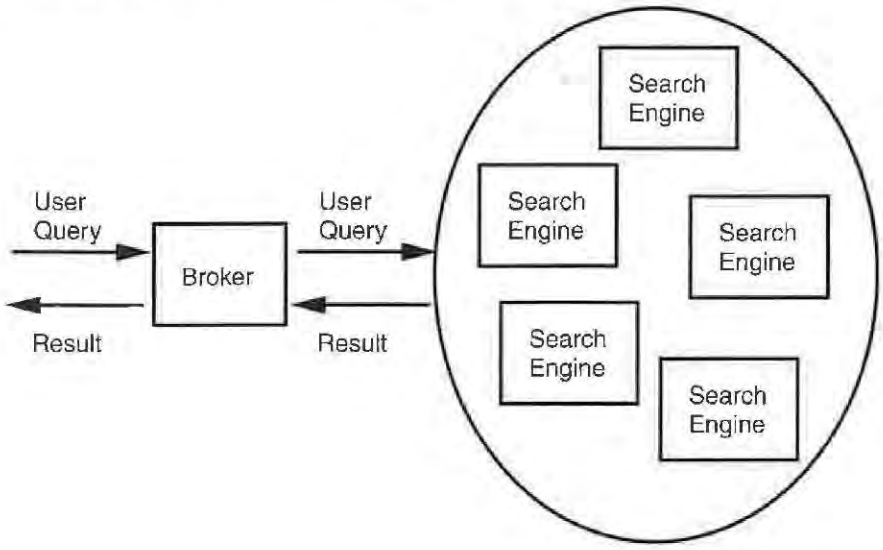


Figure 9.1 Parallel multitasking on a MIMD machine.

To move beyond multitasking and improve query response time, the computation required to evaluate a single query must be partitioned into subtasks and distributed among the multiple processors, as shown in Figure 9.2. In this configuration the broker and search processes run in parallel on separate processors as before, but now they all cooperate to evaluate the same query. High level processing in this system proceeds as follows. The broker accepts a query from the end user and distributes it among the search processes. Each of the search processes then evaluates a portion of the query and transmits an intermediate result back to the broker. Finally, the broker combines the intermediate results into a final result for presentation to the end user.

Since IR computation is typically characterized by a small amount of processing per datum applied to a large amount of data, how to partition the computation boils down to a question of how to partition the *data*. Figure 9.3 presents a high level view of the data processed by typical search algorithms (see Chapter 8). Each row represents a document, d_j, and each column represents an indexing item, k_i. Here, k_i may be a term, phrase, concept, or a more abstract indexing item such as a dimension in an LSI vector or a bit in a document signature. The entries in the matrix, $w_{i,j}$, are (possibly binary) weights, indicating if and to what degree indexing item i is assigned to document j. The indexing item weights associated with a particular document form a vector, $\vec{d_j} = (w_{1,j}, \ldots, w_{t,j})$. During search, a query is also represented as a vector of indexing item weights, $\vec{q} = (w_{1,q}, \ldots, w_{t,q})$, and the search algorithm scores each document by applying a matching function $F(\vec{d_j}, \vec{q}) = sim(d_j, q)$.

This high level data representation reveals two possible methods for partitioning the data. The first method, *document partitioning*, slices the data matrix horizontally, dividing the documents among the subtasks. The N documents in the collection are distributed across the P processors in the system,

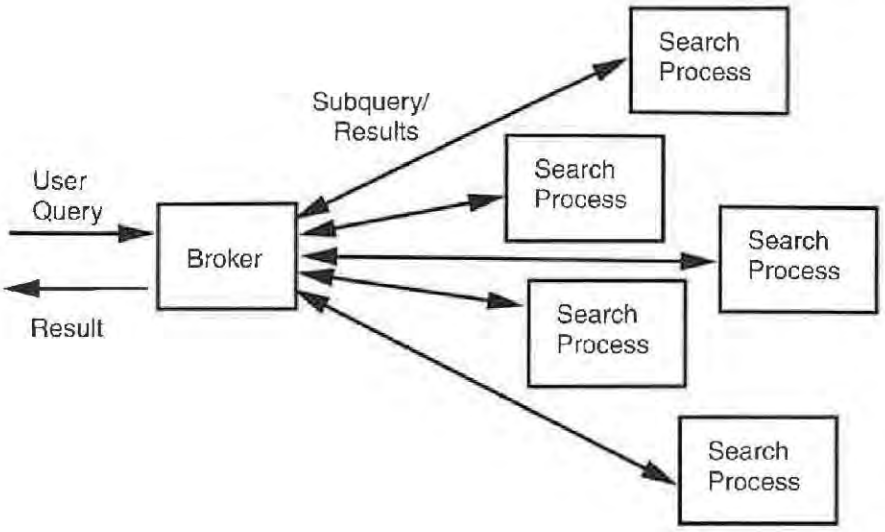


Figure 9.2 Partitioned parallel processing on a MIMD machine.

		Indexing Items					
		k_1	k_2	...	k_i	...	k_t
Documents	d_1	$w_{1,1}$	$w_{2,1}$	...	$w_{i,1}$	...	$w_{t,1}$
	d_2	$w_{1,2}$	$w_{2,2}$	...	$w_{i,2}$	...	$w_{t,2}$
	...	...	...	...	...	...	...
	d_j	$w_{1,j}$	$w_{2,j}$	...	$w_{i,j}$	...	$w_{t,j}$
	...	...	...	...	...	...	...
	d_N	$w_{1,N}$	$w_{2,N}$	...	$w_{i,N}$	...	$w_{t,N}$

Figure 9.3 Basic data elements processed by a search algorithm.

creating P subcollections of approximately N/P documents each. During query processing, each parallel process (one for each processor) evaluates the query on the subcollection of N/P documents assigned to it, and the results from each of the subcollections are combined into a final result list. The second method, *term partitioning*, slices the data matrix vertically, dividing the indexing items among the P processors such that the evaluation procedure for each document is spread over multiple processors in the system. Below we consider both of these partitioning schemes for each of the three main index structures.

Inverted Files

We first discuss inverted files for systems that employ document partitioning. Following that, we cover systems that employ term partitioning. There are two approaches to document partitioning in systems that use inverted files, namely, logical document partitioning and physical document partitioning.

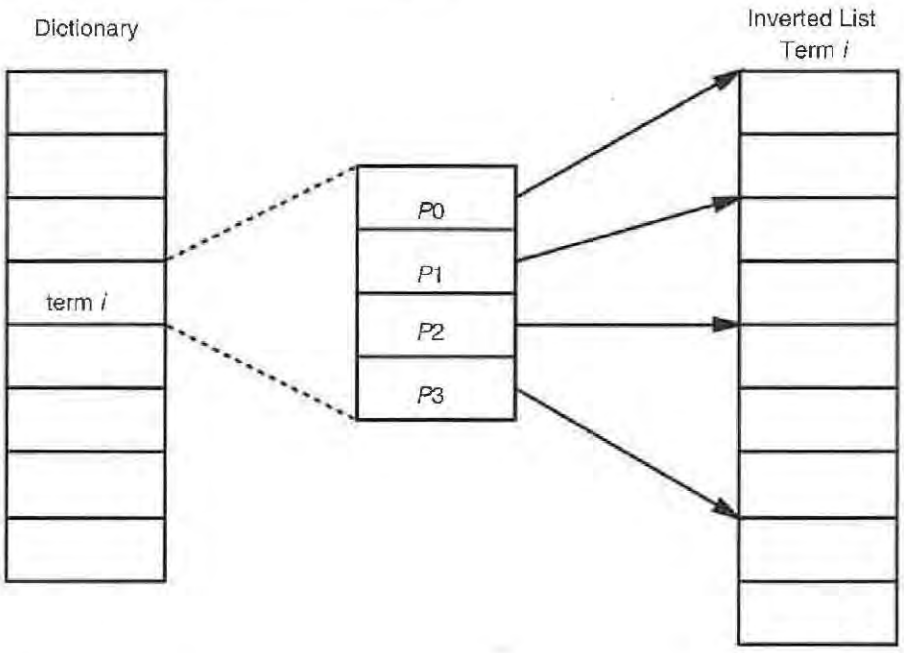


Figure 9.4 Extended dictionary entry for document partitioning.

Logical Document Partitioning

In this case, the data partitioning is done logically using essentially the same basic underlying inverted file index as in the original sequential algorithm (see Chapter 8). The inverted file is extended to give each parallel process (one for each processor) direct access to that portion of the index related to the processor's subcollection of documents. Each term dictionary entry is extended to include P pointers into the corresponding inverted list, where the j-th pointer indexes the block of document entries in the inverted list associated with the subcollection in the j-th processor. This is shown in Figure 9.4, where the dictionary entry for term i contains four pointers into term i's inverted list, one for each parallel process ($P = 4$).

When a query is submitted to the system, the broker (from Figure 9.2) first ensures that the necessary term dictionary and inverted file entries are loaded into shared memory, where all of the parallel processes can access a single shared copy. The broker then initiates P parallel processes to evaluate the query. Each process executes the same document scoring algorithm on its document subcollection, using the extended dictionary to access the appropriate entries in the inverted file. Since all of the index operations during query processing are read-only, there is no lock contention among the processes for access to the shared term dictionary and inverted file. The search processes record document scores in a single shared array of document score accumulators and notify the broker when they have completed. Updates to the accumulator array do not produce lock contention either since the subcollections scored by the different search processes are mutually exclusive. After all of the search processes have finished, the broker sorts the array of document score accumulators and produces the final ranked list of documents.

At inverted file construction time, the indexing process for logically partitioned documents can exploit the parallel processors using a variant of the indexing scheme described by Brown [123] (see Chapter 8). First, the indexer partitions the documents among the processors. Next, it assigns document identifiers such that all identifiers in partition i are less than all identifiers in partition $i+1$. The indexer then runs a separate indexing process on each processor in parallel. Each indexing process generates a batch of inverted lists, sorted by indexing item. After all of the batches have been generated, a merge step is performed to create the final inverted file. Since the inverted lists in each batch are sorted the same way, a binary heap-based priority queue is used to assemble the inverted list components from each batch that correspond to the current indexing item. The components are concatenated in partition number order to produce a final inverted list and a dictionary entry for the indexing item is created that includes the additional indexing pointers shown in Figure 9.4.

Physical Document Partitioning

In this second approach to document partitioning, the documents are physically partitioned into separate, self-contained subcollections, one for each parallel processor. Each subcollection has its own inverted file and the search processes share nothing during query evaluation. When a query is submitted to the system, the broker distributes the query to all of the parallel search processes. Each parallel search process evaluates the query on its portion of the document collection, producing a local, intermediate hit-list. The broker then collects the intermediate hit-lists from all of the parallel search processes and merges them into a final hit-list.

The P intermediate hit-lists can be merged efficiently using a binary heap-based priority queue [188]. A priority queue of n elements has the property that element i is greater than elements $2i$ and $2i+1$, where i ranges from 1 to n. A priority queue is not fully sorted, but the maximal element is always immediately available (i.e., in $\Theta(1)$ time) and can be extracted in $O(\log n)$ time. Inserting an element into a priority queue can be done in $O(\log n)$ time as well. To merge the intermediate hit-lists, a priority queue of P elements is created with the first entry from each intermediate hit-list inserted into the queue in $O(P \log P)$ time. To generate the final (and global) hit-list with the top k retrieved documents (in a global ranking), k elements are extracted from the priority queue. As each element is extracted from the priority queue, the intermediate hit-list from which the element was originally drawn inserts a new element into the priority queue. The P intermediate hit-lists can be merged into a final hit-list of k elements in $O((P+k)\log P)$ time.

The merge procedure just described assumes that the parallel search processes produce globally consistent document scores, i.e., document scores that can be merged directly. Depending on the ranking algorithm in use, each parallel search process may require global term statistics in order to produce globally consistent document scores. There are two basic approaches to collect information on global term statistics. The first approach is to compute global term statistics at indexing time and store these statistics with each of the subcollec-

tions. The second approach is for the query processing to proceed in two phases. During the first phase, the broker collects subcollection term statistics from each of the search processes and combines them into global term statistics. During the second phase, the broker distributes the query and global term statistics to the search processes and query evaluation proceeds as before. The first solution offers better query processing performance at the expense of more complex indexing, while the second solution allows subcollections to be built and maintained independently at the expense of doubling communication costs during query evaluation.

To build the inverted files for physically partitioned documents, each processor creates, in parallel, its own complete index corresponding to its document partition. If global collection statistics are stored in the separate term dictionaries, then a merge step must be performed that accumulates the global statistics for all of the partitions and distributes them to each of the partition dictionaries.

Logical document partitioning requires less communication than physical document partitioning with similar parallelization, and so is likely to provide better overall performance. Physical document partitioning, on the other hand, offers more flexibility (e.g., document partitions may be searched individually) and conversion of an existing IR system into a parallel IR system is simpler using physical document partitioning. For either document partitioning scheme, threads provide a convenient programming paradigm for creating the search processes, controlling their operation, and communicating between them. Threads are natively supported in some modern programming languages (e.g., Java [491]) and well supported in a standard way in others (e.g., POSIX threads in C/C++). Thread packages allow programmers to develop parallel programs using high level abstractions of concurrent execution, communication, and synchronization. The compiler and runtime system then map these abstractions to efficient operating system services and shared memory operations.

Term Partitioning

When term partitioning is used with an inverted file-based system, a single inverted file is created for the document collection (using the parallel construction technique described above for logical document partitioning) and the inverted lists are spread across the processors. During query evaluation, the query is decomposed into indexing items and each indexing item is sent to the processor that holds the corresponding inverted list. The processors create hit-lists with partial document scores and return them to the broker. The broker then combines the hit-lists according to the semantics of the query. For Boolean queries, the hit-lists are unioned, intersected, or subtracted as appropriate. For ranked free text queries, the hit-lists contain term scores that must be combined according to the semantics of the ranking formula.

In comparison, document partitioning affords simpler inverted index construction and maintenance than term partitioning. Their relative performance during query processing was shown by Jeong and Omiecinski [404] to depend on term distributions. Assuming each processor has its own I/O channel and disks, when term distributions in the documents and the queries are more skewed,

document partitioning performs better. When terms are uniformly distributed in user queries, term partitioning performs better. For instance, using TREC data, Ribeiro-Neto and Barbosa [673, 57] have shown that term partitioning might be twice as fast with long queries and 5–10 times faster with very short (Web-like) queries.

Suffix Arrays

We can apply document partitioning to suffix arrays in a straightforward fashion. As with physical document partitioning for inverted files, the document collection is divided among the P processors and each partition is treated as an independent, self-contained collection. The system can then apply the suffix array construction techniques described in Chapter 8 to each of the partitions, with the enhancement that all of the partitions are indexed concurrently. During search, the broker broadcasts the query to all of the search processes, collects the intermediate results, and merges the intermediate results into a final hit-list.

If all of the documents will be kept in a single collection, we can still exploit the parallel processors to reduce indexing time. An interesting property of the suffix array construction algorithm for large texts (described in Chapter 8) is that each of the merges of partial indices is independent. Therefore all of the $O((n/M)^2)$ merges may be run in parallel on separate processors. After all merges are complete, the counters for each partial index must be accumulated and the final index merge may be performed.

Term partitioning for a suffix array amounts to distributing a single suffix array over multiple processors such that each processor is responsible for a lexicographical interval of the array. During query processing, the broker distributes the query to the processors that contain the relevant portions of the suffix array and merges the results. Note that when searching the suffix array, all of the processors require access to the entire text. On a single parallel computer with shared memory (e.g., an SMP system), this is not a problem since the text may be cached in shared memory. This may be a problem, however, if shared memory is not available and communication costs are high, as is the case in a distributed system (e.g., a network of workstations).

Signature Files

To implement document partitioning in a system that uses signature files, the documents are divided among the processors as before and each processor generates signatures for its document partition. At query time, the broker generates a signature for the query and distributes it to all of the parallel processors. Each processor evaluates the query signature locally as if its document partition was a separate, self-contained collection. Then the results are sent to the broker, which combines them into a final hit-list for the user. For Boolean queries, the final result is simply a union of the results returned from each processor. For

ranked queries, the ranked hit-lists are merged as described above for inverted file implementations.

To apply term partitioning in a signature file-based system, we would have to use a bit-sliced signature file [627] and partition the bit slices across the processors. The amount of sequential work required to merge the intermediate results from each of the processors and eliminate false drops, however, severely limits the speedup S available with this organization. Accordingly, this organization is not recommended.

9.2.3 SIMD Architectures

SIMD architectures lend themselves to a more restricted domain of problems than MIMD architectures. As such, SIMD computers are less common than MIMD computers. Perhaps the best known example of the SIMD architecture is the Thinking Machines CM-2, which has been used to support both signature file- and inverted file-based information retrieval algorithms. Each processing element in the CM-2 has a 1 bit arithmetic logic unit (ALU) and a small amount of local memory. The processing elements execute local and non-local parallel instructions. A local parallel instruction causes each processing element to perform the same operation in unison on data stored in the element's local memory. A non-local parallel instruction involves communication between the processing elements and includes operations such as summing the components of a vector or finding a global maximum.

The CM-2 uses a separate front-end host computer to provide an interface to the back-end parallel processing elements. The front-end controls the loading and unloading of data in the back-end and executes serial program instructions, such as condition and iteration statements. Parallel macro instructions are sent from the front-end to a back-end microcontroller, which controls the simultaneous execution of the instruction on a set of back-end processing elements.

The CM-2 provides a layer of abstraction over the back-end processors, called *virtual processors*. One or more virtual processors map to a single physical processor. Programs express their processing needs in terms of virtual processors, and the hardware maps virtual processor operations onto physical processors. A physical processor must sequentially perform the operations for each of its virtual processors. The ratio of virtual to physical processors is called the *virtual processing ratio*, VP. As VP increases, an approximately linear increase in running time occurs.

Signature Files

The most natural application of a SIMD computer in IR is to support signature files. Recall from Chapter 8 the basic search process for signature files. First, the search system constructs a signature for the query terms. Next, the system compares the query signature with the signature of every document in the collection and marks documents with matching signatures as potentially relevant.

```
probe_doc (P_bit Doc_sig[], char *term)
{
  int   i;
  P_int Doc_match;

  Doc_match = 1;
  for (i = 0; i < num_hashes; i++) {
    Doc_match &= Doc_sig[hash (i, term)];
  }
  return Doc_match;
}
```

Figure 9.5 probe_doc.

Finally, the system scans the full text of potentially relevant documents to eliminate false drops, ranks the matching documents, and returns the hit-list to the user. If the probability of false drops is acceptably low, the full text scan may be eliminated. Also, if the system is processing Boolean queries, it may need to generate more than one signature for the query and combine the intermediate results of each signature according to the operators used in the query.

Stanfill [741] shows how this procedure can be adapted to the CM-2 (or any similar SIMD machine). The core of the procedure is the subroutine shown in Figure 9.5.‡ This routine probes the document signature `Doc_sig` for the given query word `term` by applying each of the signature hash functions to `term` and ANDing together the corresponding bits in `Doc_sig`. The result of the AND operation is stored in `Doc_match`. If `Doc_match` is 1, `term` is present in `Doc_sig`; if `Doc_match` is 0, `term` is absent. Both `Doc_sig` and `Doc_match` are parallel variables, such that each virtual processor operates in parallel on its own copy of the variables. By loading the entire signature file onto the back-end virtual processors, all of the document signatures can be searched in parallel.

This procedure must be enhanced under the following condition. If the number of words in a document $|d|$ exceeds the number of words W that can be inserted into a document signature, then the document must be segmented into $|d|/W$ segments and represented by $|d|/W$ signatures. In this case, the `probe_doc` routine is applied to all signatures for a document and an OR is taken over the individual signature results to obtain the final result for the document. If the false drop probability warrants scanning the full text of the documents, only those segments with matching signatures need be scanned. As soon as a qualifying segment is found, the entire document is marked as a match for the query.

‡ The algorithms shown in this chapter are presented using a C-like pseudo-code. Parallel data type names begin with a capital 'P'.

```
bool_search (P_bit Doc_sig[], bquery_t query)
{
  switch (query.op) {
    case AND:
      return (bool_search (Doc_sig, query.arg1)
              && bool_search (Doc_sig, query.arg2));
    case OR:
      return (bool_search (Doc_sig, query.arg1)
              || bool_search (Doc_sig, query.arg2));
    case NOT:
      return (!bool_search (Doc_sig, query.arg1));
    case WORD:
      return (probe_doc (Doc_sig, query.arg1));
  }
}
```

Figure 9.6 `bool_search`.

A general Boolean retrieval system can be implemented on top of `probe_doc` with the recursive procedure shown in Figure 9.6. Here `bquery_t` is a recursive data type that contains two arguments and an operator. If the operator is `NOT` or `WORD`, then the second argument in the `bquery_t` is empty. The final return value is stored in a parallel Boolean variable, which indicates for each document whether or not that document satisfies the Boolean query. Again, if the probability of false drops associated with the signature scheme is acceptably low, the set of matching documents may be returned immediately. Otherwise, the system must perform further processing on the text of each matching document to eliminate false drops.

If weights are available for the query terms, it is possible to build a ranking retrieval system on top of the parallel signature file search process. Query term weights could be supplied by the end-user when the query is created, or they could be assigned by the system using a collection statistic such as *idf* (see Chapter 2). The algorithm in Figure 9.7 shows how to use `probe_doc` to build a ranking system.

In `rank_search`, the `wquery_t` data type contains an array of query terms and an array of weights associated with those terms. First, all documents that contain the current term are identified with `probe_doc`. Next, the score for each of those documents is updated by adding the weight associated with the current query term (the `where` clause tests a parallel variable expression and activates only those processors that satisfy the expression). After all query terms have been processed, the parallel variable `Doc_score` contains the rank scores for all of the documents.

The final step in the processing of a weighted query is to rank the scored documents by sorting and returning the top k hits. This can be accomplished in

```
rank_search (P_bit Doc_sig[], wquery_t query)
{
  int     i;
  P_float Doc_score;
  P_bool  Doc_match;

  Doc_score = 0;
  for (i = 0; i < query.num_terms; i++) {
    Doc_match = probe_doc (Doc_sig, query.terms[i]);
    where (Doc_match) {
      Doc_score += query.weights[i];
    }
  }
  return (Doc_score);
}
```

Figure 9.7 `rank_search`.

a number of ways. One possibility is to use the global ranking routine provided by the CM-2, which takes a parallel variable and returns 0 for the largest value, 1 for the next largest value, etc. Applying this routine to `Doc_score` yields the ranked documents directly. If the number of hits returned is much less than the number of documents in the collection ($k << N$), the global ranking function performs more work than necessary. An alternative is for the retrieval system to use the global maximum routine in an iterative process of identification and extraction. During each iteration, the system applies the global maximum routine to `Doc_score` to identify the current top ranked document. The document is added to the hit-list and its score in `Doc_score` is set to -1. After k iterations, the top k hits will have been entered on the hit-list.

The techniques just described assume that the entire signature file fits in main memory. If this is not the case, additional steps must be taken to process the entire document collection. A straightforward approach is to process the collection in batches. A batch consists of as many document signatures as will fit in main memory at one time. Each batch is read into memory and scored using one of the above algorithms. The intermediate results are saved in an array of document scores. After all batches have been processed, the array of document scores is ranked and the final hit-list is generated.

In general, processing the collection in batches performs poorly due to the I/O required to read in each batch. The performance penalty imposed by the I/O can be reduced by processing multiple queries on each batch, such that the I/O costs are amortized over multiple queries. This helps query processing throughput, but does nothing to improve query processing response time.

An alternative to processing in batches is to use a parallel *bit-sliced signature file*, proposed by Panagopoulos and Faloutsos [627] (see Chapter 8).

doc_1	0	1	1	0	1	1
doc_2	1	0	0	1	0	0
doc_3	1	1	1	0	1	0
doc_4	0	1	0	0	0	0
doc_5	1	1	0	0	0	1

Figure 9.8 Document signatures.

Figure 9.8 shows a matrix representation of the signatures for a small document collection ($N = 5$). In a traditional signature file, each row of the matrix, or document signature, is stored contiguously. In a bit-sliced signature file, each column of the matrix, or bit-slice, is stored contiguously. A bit-slice is a vertical slice through the matrix, such that bit-slice i contains the i-th bit from every document signature. With this organization, the retrieval system can load just those bit-slices required by the query terms in question. Note that the file offset of bit-slice i (starting with 0) is $i*N$ bits, and the length of each bit-slice is N bits.

When using a bit-sliced signature file, each virtual processor is still responsible for scoring a single document. A virtual processor's local memory is used to store the bits from each bit-slice that correspond to the processor's document. A bit-slice, therefore, is distributed across the virtual processors with one bit at each processor. The set of bits across the virtual processors that corresponds to a single bit-slice is called a *frame*. The total number of frames is $F = M/N$, where M is the size of memory in bits available for storing bit-slices. When $F < W$ (W is the number of bit-slices in the file), the system employs a frame replacement policy to determine which bit-slices must be resident to process the query. The frame replacement policy may simply fetch all of the bit-slices that correspond to the query terms, or it may analyze the query and identify a subset of bit-slices that, when evaluated, still provides an acceptably low false drop probability.

To search the bit-sliced signature file, we must make a few modifications to our basic query processing procedures. First, the frame replacement routine must be run at the start of processing a query to insure that the required bit-slices are resident. Second, the signature hash functions must be updated to return a frame index rather than a signature bit index. The frame index is the index of the frame that contains the bit-slice corresponding to the previously computed signature bit index. Finally, the parallel bit array, `Doc_sig`, passed into `probe_doc` is replaced with the parallel bit array `Frames`, which provides each virtual processor access to its frames.

Panagopoulos and Faloutsos [627] analyze the performance of the parallel bit-sliced signature file and show that query response times of under 2 seconds can be achieved on a 128 Gb database on the CM-2. Although this technique addresses the issue of query response time on large document collections, it defeats one of the often claimed advantages of the signature file organization, namely, that indexing new documents is straightforward. In a traditional signature file

organization, new document signatures may simply be appended to the signature file. With a bit-sliced signature file, the signature file must be inverted, resulting in update costs similar to that of an inverted file.

Inverted Files

While the adaptation of signature file techniques to SIMD architectures is rather natural, inverted files are somewhat awkward to implement on SIMD machines. Nevertheless, Stanfill *et al.* [744, 740] have proposed two adaptations of inverted files for the CM-2. Recall from Chapter 8 the structure of an inverted list. In its simplest form, an inverted list contains a *posting* for each document in which the associated term appears. A posting is a tuple of the form $\langle k_i, d_j \rangle$, where k_i is a term identifier and d_j is a document identifier. Depending on the retrieval model, postings may additionally contain weights or positional information. If positional information is stored, then a posting is created for each occurrence of k_i in d_j.

The first parallel inverted file implementation for the CM-2 uses two data structures to store the inverted file: a postings table and an index. The postings table contains the document identifiers from the postings and the index maps terms to their corresponding entries in the postings table. Before the postings are loaded into these structures, they are sorted by term identifier. The document identifiers are then loaded into the postings table in this sorted order, filling in a series of rows of length P, where P is the number of processors in use. The postings table is treated as a parallel array, where the array subscript selects a particular row, and each row is spread across the P processors. For each term, the index stores the locations of the first and last entries in the postings table for the set of document identifiers associated with the term. Figure 9.9 shows a small document collection, the raw postings, and the resulting postings table and index. For example, to find the documents that contain the term 'piggy,' we look up 'piggy' in the index and determine that the postings table entries from row 1, position 3, to row 2, position 1, contain the corresponding document identifiers, or 0, 1, and 2.

At search time these data structures are used to rank documents as follows. First, the retrieval system loads the postings table onto the back-end processors. Next, the system iterates over the query terms. For each query term, an index lookup returns the range of postings table entries that must be processed. The search system then iterates over the rows included in this range. For each row, the processors that contain entries for the current term are activated and the associated document identifiers are used to update the scores of the corresponding documents.

Document scores are built up in accumulators (called *mailboxes* by Stanfill), which are allocated in a parallel array similar to the postings table. To update the accumulator for a particular document, we must determine the accumulator's row and position within the row. For convenience, we'll assume that this information (rather than document identifiers) is stored in the postings table. Furthermore,

Documents

This little piggy went to market.	This little piggy stayed home.	This little piggy had roast beef.

Postings

beef	2
had	2
home	1
little	0
little	1
little	2
market	0
piggy	0
piggy	1
piggy	2
roast	2
stayed	1
this	0
this	1
this	2
to	0
went	0

Index

	First		Last	
Term	Row	Pos.	Row	Pos.
beef	0	0	0	0
had	0	1	0	1
home	0	2	0	2
little	0	3	1	1
market	1	2	1	2
piggy	1	3	2	1
roast	2	2	2	2
stayed	2	3	2	3
this	3	0	3	2
to	3	3	3	3
went	4	0	4	0

Postings Table

2	2	1	0
1	2	0	0
1	2	2	1
0	1	2	0
0			

Figure 9.9 Parallel inverted file.

we'll assume that weights have been associated with each posting and stored in the postings table. The complete algorithm for scoring a weighted term is shown in Figure 9.10.

The `score_term` routine assumes that the index lookup for the query term has been done and the results were stored in `term`. The routine iterates over each row of postings associated with the term and determines which positions to process within the current row. `Position` is a parallel integer constant where the first instance contains 0, the second instance contains 1, etc., and the last instance contains $N\_PROCS - 1$. It is used in the `where` clause to activate the appropriate processors based on the positions of interest in the current row. The left-indexing performed on `Doc_score` at the end of the routine provides access to a particular instance of the parallel variable. This operation is significant because it involves communication between the processors. Posting weights must be shipped from the processor containing the posting to the processor containing the accumulator for the corresponding document. After the system has processed all of the query terms with `score_term`, it ranks the documents based on their scores and returns the top k documents.

It is expensive to send posting weights to accumulators on different processors. To address this problem, Stanfill [740] proposed the *partitioned postings*

```
score_term (P_float Doc_score[], P_posting Posting[],
            term_t term)
{
  int     i;
  int     first_pos;
  int     last_pos;
  P_int   Doc_row;
  P_int   Doc_pos;
  P_float Weight;

  for (i = term.first_row; i <= term.last_row; i++) {
    first_pos = (i == term.first_row ?
                 term.first_pos : 0);
    last_pos = (i == term.last_row ?
                term.last_pos : N_PROCS - 1);
    where (Position >= first_pos
           && Position <= last_pos) {
      Doc_row = Posting[i].row;
      Doc_pos = Posting[i].pos;
      Weight = term.weight * Posting[i].weight;
      [Doc_pos]Doc_score[Doc_row] += Weight;
    }
  }
}
```

Figure 9.10 score_term.

file, which eliminates the communication required in the previous algorithm by storing the postings and accumulator for a given document on the same processor. There are two tricks to accomplishing this. First, as the postings are loaded into the postings table, rather than working left to right across the rows and filling each row before starting with the next one, the postings are added to the column that corresponds to the processor where the associated document will be scored. This ensures that all of the postings associated with a document will be loaded onto the same processor as the document's accumulator. Figure 9.11(a) shows how the postings from Figure 9.9 would be loaded into a table for two processors, with documents 0 and 1 assigned to processor 0 and document 2 assigned to processor 1.

Figure 9.11(a) also demonstrates a problem with this scheme. The postings for the term *this* are skewed and no longer span consecutive rows. To handle this situation, we apply the second trick of the partitioned postings file, which is to segment the postings such that every term in segment i is lexicographically less than or equal to every term in segment $i + 1$. This is shown in Figure 9.11(b) using segments of three rows. Note how some segments may need to be padded with blank space in order to satisfy the partitioning constraints.

home	1	beef	2
little	0	had	2
little	1	little	2
market	0	piggy	2
piggy	0	roast	2
piggy	1	this	2
stayed	1		
this	0		
this	1		
to	0		
went	0		

(a)

home	1	beef	2
little	0	had	2
little	1	little	2
market	0	piggy	2
piggy	0	roast	2
piggy	1		
stayed	1	this	2
this	0		
this	1		
to	0		
went	0		

(b)

Figure 9.11 Skewed and partitioned postings.

Index

Term	First Partition	Last Partition	Tag
beef	0	0	0
had	0	0	1
home	0	0	2
little	0	0	3
market	1	1	0
piggy	1	1	1
roast	1	1	2
stayed	2	2	0
this	2	2	1
to	3	3	0
went	3	3	1

Postings Table

2	1	0	0
3	0	1	0
3	1	3	0
0	0	1	0
1	0	2	0
1	1		
0	1	1	0
1	0		
1	1		
0	0		
1	0		

Figure 9.12 Partitioned postings file.

The postings table and index undergo a few more modifications before reaching their final form, shown in Figure 9.12. First, term identifiers in the postings are replaced by term tags. The system assigns tags to terms such that no two terms in the same partition share the same tag. Second, document identifiers in the postings are replaced by document row numbers, where the row number identifies which row contains the accumulator for the document. Since the accumulator is at the same position (i.e., processor) as the posting, the row number is sufficient to identify the document. Finally, the index is modified to record the starting partition, ending partition, and tag for each term.

```
ppf_score_term (P_float Doc_score[], P_posting Posting[],
                term_t term)
{
  int     i;
  P_int   Doc_row;
  P_float Weight;

  for (i = term.first_part * N_ROWS;
       i < (term.last_part + 1) * N_ROWS; i++) {
    where (Posting[i].tag == term.tag) {
      Doc_row = Posting[i].row;
      Weight = term.weight * Posting[i].weight;
      Doc_score[Doc_row] += Weight;
    }
  }
}
```

Figure 9.13 `ppf_score_term`.

The modified term scoring algorithm is shown in Figure 9.13.

Here `N_ROWS` is the number of rows per partition. The algorithm iterates over the rows of postings that span the term's partitions and activates the processors with matching postings. Each active processor extracts the document row from the posting, calculates the term weight, and updates the document's score. After all query terms have been processed, the system ranks the documents and returns the top k. Stanfill [740] shows that the partitioned postings file imposes a space overhead of approximately 1/3 the original text (of which 10–20% is wasted partition padding) and can support sub 2-second query response times on a terabyte of text using a 64K processor CM-2.

9.3 Distributed IR

9.3.1 Introduction

Distributed computing is the application of multiple computers connected by a network to solve a single problem. A distributed computing system can be viewed as a MIMD parallel processor with a relatively slow inter-processor communication channel and the freedom to employ a heterogeneous collection of processors in the system. In fact, a single processing node in the distributed system could be a parallel computer in its own right. Moreover, if they all support the same public interface and protocol for invoking their services, the computers in the system may be owned and operated by different parties.

Distributed systems typically consist of a set of server processes, each running on a separate processing node, and a designated broker process responsible

for accepting client requests, distributing the requests to the servers, collecting intermediate results from the servers, and combining the intermediate results into a final result for the client. This computation model is very similar to the MIMD parallel processing model shown in Figure 9.2. The main difference here is that the subtasks run on different computers and the communication between the subtasks is performed using a network protocol such as TCP/IP [176] (rather than, for example, shared memory-based inter-process communication mechanisms). Another significant difference is that in a distributed system it is more common to employ a procedure for selecting a subset of the distributed servers for processing a particular request rather than broadcasting every request to every server in the system.

Applications that lend themselves well to a distributed implementation usually involve computation and data that can be split into coarse-grained operations with relatively little communication required between the operations. Parallel information retrieval based on *document partitioning* fits this profile well. In section 9.2.2 we saw how document partitioning can be used to divide the search task up into multiple, self-contained subtasks that each involve extensive computation and data processing with little communication between them. Moreover, documents are almost always grouped into collections, either for administrative purposes or to combine related documents into a single source. Collections, therefore, provide a natural granularity for distributing data across servers and partitioning the computation. Note that since term partitioning imposes greater communication overhead during query processing, it is rarely employed in a distributed system.

To build a distributed IR system, we need to consider both engineering issues common to many distributed systems and algorithmic issues specific to information retrieval. The critical engineering issues involve defining a search protocol for transmitting requests and results; designing a server that can efficiently accept a request, initiate a subprocess or thread to service the request, and exploit any locality inherent in the processing using appropriate caching techniques; and designing a broker that can submit asynchronous search requests to multiple servers in parallel and combine the intermediate results into a final end user response. The algorithmic issues include how to distribute documents across the distributed search servers, how to select which servers should receive a particular search request, and how to combine the results from the different servers.

The search protocol specifies the syntax and semantics of messages transmitted between clients and servers, the sequence of messages required to establish a connection and carry out a search operation, and the underlying transport mechanism for sending messages (e.g., TCP/IP). At a minimum, the protocol should allow a client to:

- obtain information about a search server, e.g., a list of databases available for searching at the server and possibly statistics associated with the databases;

- submit a search request for one or more databases using a well defined query language;
- receive search results in a well defined format;
- retrieve items identified in the search results.

For closed systems consisting of homogeneous search servers, a custom search protocol may be most appropriate, particularly if special functionality (e.g., encryption of requests and results) is required. Alternatively, a standard protocol may be used, allowing the system to interoperate more easily with other search servers. The Z39.50 [606] standard (see Chapter 4) for client/server information retrieval defines a widely used protocol with enough functionality to support most search applications. Another proposed protocol for distributed, heterogeneous search, called STARTS (Stanford Proposal for Internet Meta-Searching) [317], was developed at Stanford University in cooperation with a consortium of search product and service vendors. STARTS was designed from scratch to support distributed information retrieval and includes features intended to solve the algorithmic issues related to distributed IR, such as merging results from heterogeneous sources.

The other engineering issues related to building efficient client/server systems have been covered extensively in the literature (see, for example, Comer and Stevens [176] and Zomaya [852]). Rather than review them here, we continue with a more detailed look at the algorithmic issues involved in distributed IR.

9.3.2 Collection Partitioning

The procedure used to assign documents to search servers in a distributed IR system depends on a number of factors. First, we must consider whether or not the system is centrally administered. In a system comprising independently administered, heterogeneous search servers, the distributed document collections will be built and maintained independently. In this case, there is no central control of the document partitioning procedure and the question of how to partition the documents is essentially moot. It may be the case, however, that each independent search server is focused on a particular subject area, resulting in a semantic partitioning of the documents into distributed collections focused on particular subject areas. This situation is common in *meta* search systems that provide centralized access to a variety of back-end search service providers.

When the distributed system is centrally administered, more options are available. The first option is simple replication of the collection across all of the search servers. This is appropriate when the collection is small enough to fit on a single search server, but high availability and query processing throughput are required. In this scenario, the parallelism in the system is being exploited via multitasking (see Figure 9.1) and the broker's job is to route queries to the search servers and balance the loads on the servers.

Indexing the documents is handled in one of two ways. In the first method, each search server separately indexes its replica of the documents. In the second method, each server is assigned a mutually exclusive subset of documents to index and the index subsets are replicated across the search servers. A merge of the subsets is required at each search server to create the final indexes (which can be accomplished using the technique described under Document Partitioning in section 9.2.2). In either case, document updates and deletions must be broadcast to all servers in the system. Document additions may be broadcast, or they may be batched and partitioned depending on their frequency and how quickly updates must be reflected by the system.

The second option is random distribution of the documents. This is appropriate when a large document collection must be distributed for performance reasons but the documents will always be viewed and searched as if they are part of a single, logical collection. The broker broadcasts every query to all of the search servers and combines the results for the user.

The final option is explicit semantic partitioning of the documents. Here the documents are either already organized into semantically meaningful collections, such as by technical discipline, or an automatic clustering or categorization procedure is used to partition the documents into subject specific collections.

9.3.3 Source Selection

Source selection is the process of determining which of the distributed document collections are most likely to contain relevant documents for the current query, and therefore should receive the query for processing. One approach is to always assume that every collection is equally likely to contain relevant documents and simply broadcast the query to all collections. This approach is appropriate when documents are randomly partitioned or there is significant semantic overlap between the collections.

When document collections are partitioned into semantically meaningful collections or it is prohibitively expensive to search every collection every time, the collections can be ranked according to their likelihood of containing relevant documents. The basic technique is to treat each collection as if it were a single large document, index the collections, and evaluate the query against the collections to produce a ranked listing of collections. We can apply a standard cosine similarity measure using a query vector and collection vectors. To calculate a term weight in the collection vector using tf-idf style weighting (see Chapter 2), term frequency $tf_{i,j}$ is the total number of occurrences of term i in collection j, and the inverse document frequency idf_i for term i is $\log(N/n_i)$, where N is the total number of collections and n_i is the number of collections in which term i appears.

A danger of this approach is that although a particular collection may receive a high query relevance score, there may not be individual documents within the collection that receive a high query relevance score, essentially resulting in a false drop and unnecessary work to score the collection. Moffat and Zobel [574]

propose avoiding this problem by indexing each collection as a series of blocks, where each block contains B documents. When B equals 1, this is equivalent to indexing all of the documents as a single, monolithic collection. When B equals the number of documents in each collection, this is equivalent to the original solution. By varying B, a tradeoff is made between collection index size and likelihood of false drops.

An alternative to searching a collection index was proposed by Voorhees [792], who proposes using training queries to build a content model for the distributed collections. When a new query is submitted to the system, its similarity to the training queries is computed and the content model is used to determine which collections should be searched and how many hits from each collection should be returned.

9.3.4 Query Processing

Query processing in a distributed IR system proceeds as follows:

(1) Select collections to search.

(2) Distribute query to selected collections.

(3) Evaluate query at distributed collections in parallel.

(4) Combine results from distributed collections into final result.

As described in the previous section, Step 1 may be eliminated if the query is always broadcast to every document collection in the system. Otherwise, one of the previously described selection algorithms is used and the query is distributed to the selected collections. Each of the participating search servers then evaluates the query on the selected collections using its own local search algorithm. Finally, the results are merged.

At this point we have covered everything except how to merge the results. There are a number of scenarios. First, if the query is Boolean and the search servers return Boolean result sets, all of the sets are simply unioned to create the final result set. If the query involves free-text ranking, a number of techniques are available ranging from simple/naive to complex/accurate.

The simplest approach is to combine the ranked hit-lists using round robin interleaving. This is likely to produce poor quality results since hits from irrelevant collections are given status equal to that of hits from highly relevant collections. An improvement on this process is to merge the hit-lists based on relevance score. As with the parallel process described for Document Partitioning in section 9.2.2, unless proper global term statistics are used to compute the document scores, we may get incorrect results. If documents are randomly distributed such that global term statistics are consistent across all of the distributed collections, the merging based on relevance score is sufficient to maintain retrieval effectiveness. If, however, the distributed document collections are

semantically partitioned or maintained by independent parties, then reranking must be performed.

Callan [139] proposes reranking documents by weighting document scores based on their collection similarity computed during the source selection step. The weight for a collection is computed as $w = 1 + \mid C \mid \cdot(s - \bar{s})/\bar{s}$, where $\mid C \mid$ is the number of collections searched, s is the collection's score, and $\bar{s}$ is the mean of the collection scores.

The most accurate technique for merging ranked hit-lists is to use accurate global term statistics. This can be accomplished in one of a variety of ways. First, if the collections have been indexed for source selection, that index will contain global term statistics across all of the distributed collections. The broker can include these statistics in the query when it distributes the query to the remote search servers. The servers can then account for these statistics in their processing and produce relevance scores that can be merged directly. If a collection index is unavailable, query distribution can proceed in two rounds of communication. In the first round, the broker distributes the query and gathers collection statistics from each of the search servers. These statistics are combined by the broker and distributed back to the search servers in the second round.

Finally, the search protocol can require that search servers return global query term statistics and per-document query term statistics [317, 441]. The broker is then free to rerank every document using the query term statistics and a ranking algorithm of its choice. The end result is a hit-list that contains documents from the distributed collections ranked in the same order as if all of the documents had been indexed in a single collection.

9.3.5 Web Issues

Information retrieval on the World Wide Web is covered extensively in Chapter 13. For completeness, we briefly mention here how parallel and distributed information retrieval applies to the Web. The most direct application is to gather all of the documents on the Web into a single, large document collection. The parallel and distributed techniques described above can then be used directly as if the Web were any other large document collection. This is the approach currently taken by most of the popular Web search services.

Alternatively, we can exploit the distributed system of computers that make up the Web and spread the work of collecting, organizing, and searching all of the documents. This is the approach taken by the Harvest system [108]. Harvest comprises a number of components for gathering, summarizing, replicating, distributing, and searching documents. User queries are processed by *brokers*, which collect and refine information from *gatherers* and other brokers. The information at a particular broker is typically related to a restricted set of topics, allowing users to direct their queries to the most appropriate brokers. A central broker registry helps users find the best brokers for their queries (see Figure 13.4).

9.4 Trends and Research Issues

Parallel computing holds great potential for tackling the performance and scale issues associated with the large and growing document collections currently available online. In this chapter we have surveyed a number of techniques for exploiting modern parallel architectures. The trend in parallel hardware is the development of general MIMD machines. Coincident with this trend is the availability of features in modern programming languages, such as threads and associated synchronization constructs, that greatly facilitate the task of developing programs for these architectures. In spite of this trend, research in parallel IR algorithms on MIMD machines is relatively young, with few standard results to draw on.

Much of the early work in parallel IR was aimed at supporting signature files on SIMD architectures. Although SIMD machines are well suited to processing signature files, both SIMD machines and signature files have fallen out of favor in their respective communities. SIMD machines are difficult to program and are well suited to a relatively small class of problems. As Chapter 8 points out, signature files provide poor support for document ranking and hold few, if any, advantages over inverted files in terms of functionality, index size, and processing speed [851].

Distributed computing can be viewed as a form of MIMD computing with relatively high interprocessor communication costs. Most of the parallel IR algorithms discussed in this chapter, however, have a high ratio of computation to communication, and are well suited to both symmetric multiprocessor and distributed implementations. In fact, by using an appropriate abstraction layer for inter-process communication, we can easily implement a parallel system that works well on both multiprocessor and distributed architectures with relatively little modification.

Many challenges remain in the area of parallel and distributed text retrieval. While we have presented a number of approaches in this chapter, none stand out as the definitive solution for building parallel or distributed information retrieval systems. In addition to the continued development and investigation of parallel indexing and search techniques for systems based on inverted files and suffix arrays, two specific challenges stand out.

The first challenge is measuring retrieval effectiveness on large text collections. Although we can easily measure the speedup achieved by a given parallel system, measuring the quality of the results produced by that system is another story. This challenge, of course, is not unique to parallel IR systems. Large collections pose problems particularly when it comes to generating relevance judgments for queries. The pooling techniques used in TREC (see Chapter 3) may not work. There, ranked result lists are combined from multiple systems to produce a relatively small set of documents for human evaluation. The assumption is that most, if not all, of the relevant documents will be included in the pool. With large collections, this assumption may not hold. Moreover, it is unclear how important recall is in this context.

The second significant challenge is interoperability, or building distributed IR systems from heterogeneous components. The need for distributed systems

comprising heterogeneous back-end search servers is clear from the popularity of meta search services on the Web. The functionality of these systems is limited, however, due to the lack of term statistics from the back-end search servers, which would otherwise allow for accurate reranking and result list merging. Moreover, each search server employs its own, custom query language, opening up the possibility that the original intent of the query is lost when it is translated to the back-end query languages. Protocol standardization efforts, such as STARTS [317], attempt to address these problems, but commitment to these standards by the entire community of search providers is required.

9.5 Bibliographic Discussion

A thorough overview of parallel and distributed computing can be found in the *Parallel and Distributed Computing Handbook* [852], edited by Albert Zomaya. Many interesting research papers specific to parallel and distributed information systems can be found in the proceedings of the IEEE *International Conference on Parallel and Distributed Information Systems.*

Stanfill *et al.* [742, 744, 740] are responsible for much of the early work using massively parallel hardware (in particular, the Thinking Machines Connection Machine) to solve IR problems. Pogue and Willet [645] also explored massively parallel IR using the ICL Distributed Array Processor. Salton and Buckley [701] provide some interesting comments on the early implementations of parallel IR, challenging both their speed and effectiveness.

Lu *et al.* [524] analyze how to properly scale SMP hardware for parallel IR and emphasize the importance of proper hardware balance. Investigations into parallel and distributed inverted file implementations have been performed by Tomasic and García-Molina [762, 763, 764], Jeong and Omiecinski [404], and Ribeiro-Neto and Barbosa [673]. Parallel and distributed algorithms for suffix array construction and search have been explored by Navarro *et al.* [591]. Given P processors and total text of size n, they obtain average indexing times that are $O(n/P \log n)$ CPU time and $O(n/P)$ communication time.

Macleod *et al.* [535] offer a number of strategies and tips for building distributed information retrieval systems. Cahoon and McKinley [137] analyze the performance of the Inquery distributed information retrieval system.

Source selection and collection fusion issues have been investigated by Gravano *et al.* using the GlOSS system [318], Voorhees *et al.* [792], Callan *et al.* [139], Moffat and Zobel [574], Viles and French [787], and others.

Acknowledgements

The author gratefully acknowledges the support of IBM.

Chapter 10
User Interfaces and Visualization

by Marti A. Hearst

10.1 Introduction

This chapter discusses user interfaces for communication between human information seekers and information retrieval systems. Information seeking is an imprecise process. When users approach an information access system they often have only a fuzzy understanding of how they can achieve their goals. Thus the user interface should aid in the understanding and expression of information needs. It should also help users formulate their queries, select among available information sources, understand search results, and keep track of the progress of their search.

The human-computer interface is less well understood than other aspects of information retrieval, in part because humans are more complex than computer systems, and their motivations and behaviors are more difficult to measure and characterize. The area is also undergoing rapid change, and so the discussion in this chapter will emphasize recent developments rather than established wisdom.

The chapter will first outline the human side of the information seeking process and then focus on the aspects of this process that can best be supported by the user interface. Discussion will encompass current practice and technology, recently proposed innovative ideas, and suggestions for future areas of development.

Section 10.2 outlines design principles for human-computer interaction and introduces notions related to information visualization. section 10.3 describes information seeking models, past and present. The next four sections describe user interface support for starting the search process, for query specification, for viewing retrieval results in context, and for interactive relevance feedback. The last major section, section 10.8, describes user interface techniques to support the information access process as a whole. Section 10.9 speculates on future developments and Section 10.10 provides suggestions for further reading. Figure 10.1 presents the flow of the chapter contents.

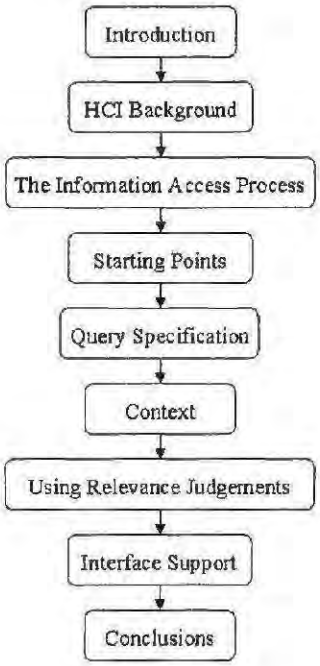


Figure 10.1 The flow of this chapter's contents.

10.2 Human-Computer Interaction

What makes an effective human-computer interface? Ben Shneiderman, an expert in the field, writes [725, p.10]:

> Well designed, effective computer systems generate positive feelings of success, competence, mastery, and clarity in the user community. When an interactive system is well-designed, the interface almost disappears, enabling users to concentrate on their work, exploration, or pleasure.

As steps towards achieving these goals, Shneiderman lists principles for design of user interfaces. Those which are particularly important for information access include (slightly restated): provide informative feedback, permit easy reversal of actions, support an internal locus of control, reduce working memory load, and provide alternative interfaces for novice and expert users. Each of these principles should be instantiated differently depending on the particular interface application. Below we discuss those principles that are of special interest to information access systems.

10.2.1 Design Principles

Offer informative feedback. This principle is especially important for information access interfaces. In this chapter we will see current ideas about how to provide

users with feedback about the relationship between their query specification and documents retrieved, about relationships among retrieved documents, and about relationships between retrieved documents and metadata describing collections. If the user has control of how and when feedback is provided, then the system provides an *internal locus of control.*

Reduce working memory load. Information access is an iterative process, the goals of which shift and change as information is encountered. One key way information access interfaces can help with memory load is to provide mechanisms for keeping track of choices made during the search process, allowing users to return to temporarily abandoned strategies, jump from one strategy to the next, and retain information and context across search sessions. Another memory-aiding device is to provide browsable information that is relevant to the current stage of the information access process. This includes suggestions of related terms or metadata, and search starting points including lists of sources and topic lists.

Provide alternative interfaces for novice and expert users. An important tradeoff in all user interface design is that of simplicity versus power. Simple interfaces are easier to learn, at the expense of less flexibility and sometimes less efficient use. Powerful interfaces allow a knowledgeable user to do more and have more control over the operation of the interface, but can be time-consuming to learn and impose a memory burden on people who use the system only intermittently. A common solution is to use a 'scaffolding' technique [684]. The novice user is presented with a simple interface that can be learned quickly and that provides the basic functionality of the application, but is restricted in power and flexibility. Alternative interfaces are offered for more experienced users, giving them more control, more options, and more features, or potentially even entirely different interaction models. Good user interface design provides intuitive bridges between the simple and the advanced interfaces.

Information access interfaces must contend with special kinds of simplicity/power tradeoffs. One such tradeoff is the amount of information shown about the workings of the search system itself. Users who are new to a system or to a particular collection may not know enough about the system or the domain associated with the collection to make choices among complex features. They may not know how best to weight terms, or in the case of relevance feedback, not know what the effects of reweighting terms would be. On the other hand, users that have worked with a system and gotten a feeling for a topic are likely to be able to choose among suggested terms to add to their query in an informed manner. Determining how much information to show the user of the system is a major design choice in information access interfaces.

10.2.2 The Role of Visualization

The tools of computer interface design are familiar to most computer users today: windows, menus, icons, dialog boxes, and so on. These make use of bit-mapped display and computer graphics to provide a more accessible interface

than command-line-based displays. A less familiar but growing area is that of *information visualization*, which attempts to provide visual depictions of very large information spaces.

Humans are highly attuned to images and visual information [769, 456, 483]. Pictures and graphics can be captivating and appealing, especially if well designed. A visual representation can communicate some kinds of information much more rapidly and effectively than any other method. Consider the difference between a written description of a person's face and a photograph of it, or the difference between a table of numbers containing a correlation and a scatter plot showing the same information.

The growing prevalence of fast graphics processors and high resolution color monitors is increasing interest in information visualization. Scientific visualization, a rapidly advancing branch of this field, maps physical phenomena onto two- or three-dimensional representations [433]. An example of scientific visualization is a colorful image of the pattern of peaks and valleys on the ocean floor; this provides a view of physical phenomena for which a photograph cannot (currently) be taken. Instead, the image is constructed from data that represent the underlying phenomena.

Visualization of inherently abstract information is more difficult, and visualization of textually represented information is especially challenging. Language is our main means of communicating abstract ideas for which there is no obvious physical manifestation. What does a picture look like that describes negotiations over a trade agreement in which one party demands concessions on environmental policies while the other requires help in strengthening its currency?

Despite the difficulties, researchers are attempting to represent aspects of the information access process using information visualization techniques. Some of these will be described later in this chapter. Aside from using *icons* and *color highlighting*, the main information visualization techniques include *brushing and linking* [233, 773], *panning and zooming* [71], *focus-plus-context* [502], *magic lenses* [95], and the use of *animation* to retain context and help make occluded information visible [676, 143]. These techniques support dynamic, interactive use. Interactivity seems to be an especially important property for visualizing abstract information, although it has not played as large a role within scientific visualization.

Brushing and linking refers to the connecting of two or more views of the same data, such that a change to the representation in one view affects the representation in the other views as well. For example, say a display consists of two parts: a histogram and a list of titles. The histogram shows, for a set of documents, how many documents were published each year. The title list shows the titles for the corresponding documents. Brushing and linking would allow the user to assign a color, say red, to one bar of the histogram, thus causing the titles in the list display that were published during the corresponding year to also be highlighted in red.

Panning and zooming refers to the actions of a movie camera that can scan sideways across a scene (panning) or move in for a closeup or back away to get a wider view (zooming). For example, text clustering can be used to show a

top-level view of the main themes in a document collection (see Figures 10.7 and 10.8). Zooming can be used to move 'closer,' showing individual documents as icons, and then zoom in closer still to see the text associated with an individual document.

When zooming is used, the more detail that is visible about a particular item, the less can be seen about the surrounding items. Focus-plus-context is used to partly alleviate this effect. The idea is to make one portion of the view — the focus of attention — larger, while simultaneously shrinking the surrounding objects. The farther an object is from the focus of attention, the smaller it is made to appear, like the effect seen in a fisheye camera lens (also in some door peepholes).

Magic lenses are directly manipulable transparent windows that, when overlapped on some other data type, cause a transformation to be applied to the underlying data, thus changing its appearance (see Figure 10.13). The most straightforward application of magic lenses is for drawing tasks, and it is especially useful if used as a two-handed interface. For example, the left hand can be used to position a color lens over a drawing of an object. The right hand is used to mouse-click on the lens, thus causing the appearance of the underlying object to be transformed to the color specified by the lens.

Additionally, there are a large number of graphical methods for depicting trees and hierarchies, some of which make use of animation to show nodes that would otherwise be occluded (hidden from view by other nodes) [286, 364, 407, 478, 676].

It is often useful to combine these techniques into an interface layout consisting of an *overview plus details* [321, 644]. An overview, such as a table-of-contents of a large manual, is shown in one window. A mouse-click on the title of the chapter causes the text of the chapter itself to appear in another window, in a linking action (see Figure 10.19). Panning and zooming or focus-plus-context can be used to change the view of the contents within the overview window.

10.2.3 Evaluating Interactive Systems

From the viewpoint of user interface design, people have widely differing abilities, preferences, and predilections. Important differences for information access interfaces include relative spatial ability and memory, reasoning abilities, verbal aptitude, and (potentially) personality differences [227, 725]. Age and cultural differences can contribute to acceptance or rejection of interface techniques [557]. An interface innovation can be useful and pleasing for some users, and foreign and cumbersome for others. Thus software design should allow for flexibility in interaction style, and new features should not be expected to be equally helpful for all users.

An important aspect of human-computer interaction is the methodology for evaluation of user interface techniques. Precision and recall measures have been widely used for comparing the ranking results of non-interactive systems, but are less appropriate for assessing interactive systems [470]. The standard evaluations

emphasize high recall levels; in the TREC tasks systems are compared to see how well they return the top 1000 documents (see chapter 3). However, in many interactive settings, users require only a few relevant documents and do not care about high recall to evaluate highly interactive information access systems, useful metrics beyond precision and recall include: time required to learn the system, time required to achieve goals on benchmark tasks, error rates, and retention of the use of the interface over time. Throughout this chapter, empirical results of user studies are presented whenever they are available.

Empirical data involving human users is time consuming to gather and difficult to draw conclusions from. This is due in part to variation in users' characteristics and motivations, and in part to the broad scope of information access activities. Formal psychological studies usually only uncover narrow conclusions within restricted contexts. For example, quantities such as the length of time it takes for a user to select an item from a fixed menu under various conditions have been characterized empirically [142], but variations in interaction behavior for complex tasks like information access are difficult to account for accurately. Nielsen [605] advocates a more informal evaluation approach (called heuristic evaluation) in which user interface affordances are assessed in terms of more general properties and without concern about statistically significant results.

10.3 The Information Access Process

A person engaged in an information seeking process has one or more *goals* in mind and uses a search system as a tool to help achieve those goals. Goals requiring information access can range quite widely, from finding a plumber to keeping informed about a business competitor, from writing a publishable scholarly article to investigating an allegation of fraud.

Information access *tasks* are used to achieve these goals. These tasks span the spectrum from asking specific questions to exhaustively researching a topic. Other tasks fall between these two extremes. A study of business analysts [614] found three main kinds of information seeking tasks: monitoring a well known topic over time (such as researching competitors' activities each quarter), following a plan or stereotyped series of searches to achieve a particular goal (such as keeping up to date on good business practices), and exploring a topic in an undirected fashion (as when getting to know an unfamiliar industry). Although the goals differ, there is a common core revolving around the information seeking component, which is our focus here.

10.3.1 Models of Interaction

Most accounts of the information access process assume an interaction cycle consisting of query specification, receipt and examination of retrieval results, and then either stopping or reformulating the query and repeating the process

until a perfect result set is found [700, 726]. In more detail, the standard process can be described according to the following sequence of steps (see Figure 10.2):

(1) Start with an information need.

(2) Select a system and collections to search on.

(3) Formulate a query.

(4) Send the query to the system.

(5) Receive the results in the form of information items.

(6) Scan, evaluate, and interpret the results.

(7) Either stop, or,

(8) Reformulate the query and go to step 4.

This simple interaction model (used by Web search engines) is the only model that most information seekers see today. This model does not take into account the fact that many users dislike being confronted with a long disorganized list of retrieval results that do not directly address their information needs. It also contains an underlying assumption that the user's information need is static and the information seeking process is one of successively refining a query until it retrieves all and only those documents relevant to the original information need.

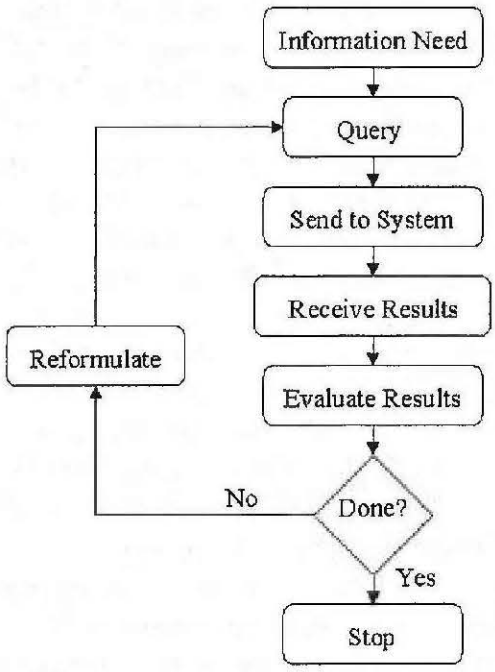


Figure 10.2 A simplified diagram of the standard model of the information access processes.

In actuality, users learn during the search process. They scan information, read the titles in result sets, read the retrieved documents themselves, viewing lists of topics related to their query terms, and navigating within hyperlinked Web sites. The recent advent of hyperlinks as a pivotal part of the information seeking process makes it no longer feasible to ignore the role of scanning and navigation within the search process itself. In particular, today a near-miss is much more acceptable than it was with bibliographic search, since an information seeker using the Web can navigate hyperlinks from a near-miss in the hopes that a useful page will be a few links away.

The standard model also downplays the interaction that takes place when the user scans terms suggested as a result of relevance feedback, scans thesaurus structures, or views thematic overviews of document collections. It deemphasizes the role of source selection, which is increasingly important now that, for the first time, tens of thousands of information collections are immediately reachable for millions of people.

Thus, while useful for describing the basics of information access systems, this simple interaction model is being challenged on many fronts [65, 614, 105, 365, 192]. Bates [65] proposes the 'berry-picking' model of information seeking, which has two main points. The first is that, as a result of reading and learning from the information encountered throughout the search process, the users' information needs, and consequently their queries, continually shift. Information encountered at one point in a search may lead in a new, unanticipated direction. The original goal may become partly fulfilled, thus lowering the priority of one goal in favor of another. This is posed in contrast to the assumption of 'standard' information retrieval that the user's information need remains the same throughout the search process. The second point is that users' information needs are not satisfied by a single, final retrieved set of documents, but rather by a series of selections and bits of information found along the way. This is in contrast to the assumption that the main goal of the search process is to hone down the set of retrieved documents into a perfect match of the original information need.

The berry-picking model is supported by a number of observational studies [236, 105], including that of O'Day and Jeffries [614]. They found that the information seeking process consisted of a series of interconnected but diverse searches on one problem-based theme. They also found that search results for a goal tended to trigger new goals, and hence search in new directions, but that the context of the problem and the previous searches was carried from one stage of search to the next. They also found that the main value of the search resided in the accumulated learning and acquisition of information that occurred during the search process, rather than in the final results set.

Thus, a user interface for information access should allow users to reassess their goals and adjust their search strategy accordingly. A related situation occurs when users encounter a 'trigger' that causes them to pursue a different strategy temporarily, perhaps to return to the current unfinished activity at a later time. An implication of these observations is that the user interface should support search strategies by making it easy to follow trails with unanticipated results. This can be accomplished in part by supplying ways to record the progress

of the current strategy and to store, find, and reload intermediate results, and by supporting pursuit of multiple strategies simultaneously.

The user interface should also support methods for monitoring the status of the current strategy in relation to the user's current task and high-level goals. One way to cast the activity of monitoring the progress of a search strategy relative to a goal or subgoal is in terms of a cost/benefit analysis, or an analysis of diminishing returns [690]. This kind of analysis assumes that at any point in the search process, the user is pursuing the strategy that has the highest expected utility. If, as a consequence of some local tactical choices, another strategy presents itself as being of higher utility than the current one, the current one is (temporarily or permanently) abandoned in favor of the new strategy.

There are a number of theories and frameworks that contrast *browsing*, *querying*, *navigating*, and *scanning* along several dimensions [75, 159, 542, 804]. Here we assume that users scan information structure, be it titles, thesaurus terms, hyperlinks, category labels, or the results of clustering, and then either select a displayed item for some purpose (to read in detail, to use as input to a query, to navigate to a new page of information) or formulate a query (either by recalling potential words or by selecting categories or suggested terms that have been scanned). In both cases, a new set of information is then made viewable for scanning. Queries tend to produce new, ad hoc collections of information that have not been gathered together before, whereas selection retrieves information that has already been composed or organized. Navigation refers to following a chain of links, switching from one view to another, toward some goal, in a sequence of scan and select operations. Browsing refers to the casual, mainly undirected exploration of information structures, and is usually done in tandem with selection, although queries can also be used to create subcollections to browse through. An important aspect of the interaction process is that the output of one action should be easily used as the input to the next.

10.3.2 Non-Search Parts of the Information Access Process

The O'Day and Jeffries study [614] found that information seeking is only one part of the full work process their subjects were engaged in. In between searching sessions many different kinds of work was done with the retrieved information, including reading and annotating [617] and analysis. O'Day and Jeffries examined the analysis steps in more detail, finding that 80% of this work fell into six main types: finding trends, making comparisons, aggregating information, identifying a critical subset, assessing, and interpreting. The remaining 20% consisted of cross-referencing, summarizing, finding evocative visualizations for reports, and miscellaneous activities. The Sensemaking work of Russell *et al.* [690] also discusses information work as a process in which information retrieval plays only a small part. They observe that most of the effort made in Sensemaking is in the synthesis of a good representation, or ways of thinking about, the problem at hand. They describe the process of formulating and crystallizing the important concepts for a given task.

From these observations it is convenient to divide the entire information access process into two main components: search/retrieval, and analysis/synthesis of results. User interfaces should allow both kinds of activity to be tightly interwoven. However, analysis/synthesis are activities that can be done independently of information seeking, and for our purposes it is useful to make a distinction between the two types of activities.

10.3.3 Earlier Interface Studies

The bulk of the literature on studies of human-computer information seeking behavior concerns information intermediaries using online systems consisting of bibliographic records (e.g., [546, 707, 104]), sometimes with costs assessed per time unit. Unfortunately, many of the assumptions behind those studies do not reflect the conditions of modern information access [335, 222]. The differences include the following:

- The text being searched now is often full text rather than bibliographic citations. Because users have access to full text, rather than document surrogates, it is more likely that simple queries will find relevant answers directly as part of the search process.
- Modern systems use statistical ranking (which is more effective when abstracts and full text are available than when only titles and citations are available) whereas most studies were performed on Boolean systems.
- Much of modern searching is done by end users, many new to online searching, rather than professional intermediaries, which were the focus of many of the earlier studies.
- Tens of thousands of sources are now available online on networked information systems, and many are tightly coupled via hyperlinks, as opposed to being stored in separate collections owned by separate services. Earlier studies generally used systems in which moving from one collection to another required prior knowledge of the collections and considerable time and effort to switch. A near miss is much more useful in this hyperlinked environment than in earlier systems, since hyperlinks allow users to navigate from the near miss directly to the source containing information of interest. In a card catalog environment, where documents are represented as isolated units, a near miss consists of finding a book in the general area of interest and then going to the bookshelf in the library to look for related books, or obtaining copies of many issues of a journal and scanning for related articles.
- Finally, most users have access to bit-mapped displays allowing for direct manipulation, or at least form fillin. Most earlier studies and bibliographic systems were implemented on TTY displays, which require command-line based syntax and do a poor job of retaining context.

Despite these significant differences, some general information seeking strategies have been identified that seem to transfer across systems. Additionally, although modern systems have remedied many of the problems of earlier online public access catalogs, they also introduce new problems of their own.

10.4 Starting Points

Search interfaces must provide users with good ways to get started. An empty screen or a blank entry form does not provide clues to help a user decide how to start the search process. Users usually do not begin by creating a long, detailed expression of their information need. Studies show that users tend to start out with very short queries, inspect the results, and then modify those queries in an incremental feedback cycle [22]. The initial query can be seen as a kind of 'testing the water' to see what kinds of results are returned and get an idea of how to reformulate the query [804, 65]. Thus, one task of an information access interface is to help users select the sources and collections to search on.

For example, there are many different information sources associated with cancer, and there are many different kinds of information a user might like to know about cancer. Guiding the user to the right set of starting points can help with the initial problem formulation. Traditional bibliographic search assumes that the user begins by looking through a list of names of sources and choosing which collections to search on, while Web search engines obliterate the distinctions between sources and plunge the user into the middle of a Web site with little information about the relationship of the search hit to the rest of the collection. In neither case is the interface to the available sources particularly helpful.

In this section we will discuss four main types of starting points: *lists*, *overviews*, *examples*, and *automated source selection*.

10.4.1 Lists of Collections

Typical online systems such as LEXIS-NEXIS require users to begin any inquiry with a scan through a long list of source names and guess which ones will be of interest. Usually little information beyond the name of the collection is provided online for these sources (see Figure 10.3). If the user is not satisfied with the results on one collection, they must reissue the query on another collection.

Frequent searchers eventually learn a set of sources that are useful for their domains of interest, either through experience, formal training, or recommendations from friends and colleagues. Often-used sources can be stored on a 'favorites' list, also known as a *bookmark* list or a *hotlist* on the Web. Recent research explores the maintenance of a personalized information profile for users or work groups, based on the kinds of information they've used in the past [277].

However, when users want to search outside their domains of expertise, a list of familiar sources is not sufficient. Professional searchers such as librarians

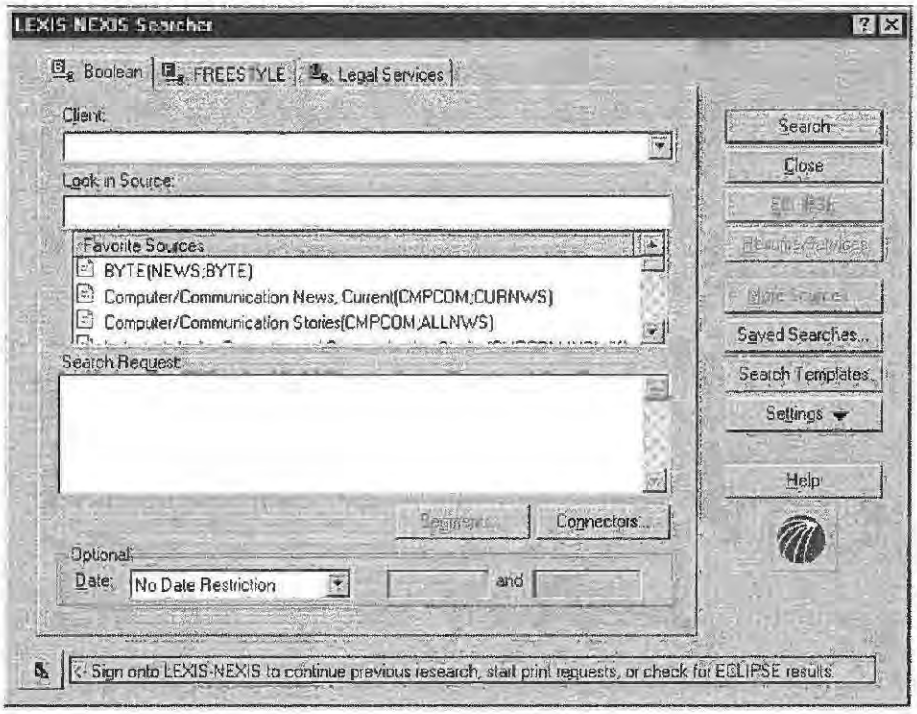


Figure 10.3 The LEXIS-NEXIS source selection screen.

learn through experience and years of training which sources are appropriate for various information needs. The restricted nature of traditional interfaces to information collections discourages exploration and discovery of new useful sources. However, recently researchers have devised a number of mechanisms to help users understand the contents of collections as a way of getting started in their search.

10.4.2 Overviews

Faced with a large set of text collections, how can a user choose which to begin with? One approach is to study an overview of the contents of the collections. An overview can show the topic domains represented within the collections, to help users select or eliminate sources from consideration. An overview can help users get started, directing them into general neighborhoods, after which they can navigate using more detailed descriptions. Shneiderman [724] advocates an interaction model in which the user begins with an overview of the information to be worked with, then pans and zooms to find areas of potential interest, and then view details. The process is repeated as often as necessary.

Three types of overviews are discussed in this subsection. The first is display and navigation of large topical *category hierarchies* associated with the documents of a collection. The second is automatically derived overviews, usually created by unsupervised *clustering techniques* on the text of documents, that attempt to extract overall characterizing themes from collections. The third type

of overview is that created by applying a variant of *co-citation analysis* on connections or links between different entities within a collection. Other kinds of overviews are possible, for example, showing graphical depictions of bookshelves or piles of books [681, 46].

Category or Directory Overviews

There exist today many large online text collections to which category labels have been assigned. Traditional online bibliographic systems have for decades assigned subject headings to books and other documents [752]. MEDLINE, a large collection of biomedical articles, has associated with it Medical Subject Headings (MeSH) consisting of approximately 18,000 categories [523]. The Association for Computing Machinery (ACM) has developed a hierarchy of approximately 1200 category (keyword) labels.† Yahoo![839], one of the most popular search sites on the World Wide Web, organizes Web pages into a hierarchy consisting of thousands of category labels.

The popularity of Yahoo! and other Web directories suggests that hierarchically structured categories are useful starting points for users seeking information on the Web. This popularity may reflect a preference to begin at a logical starting point, such as the home page for a set of information, or it may reflect a desire to avoid having to guess which words will retrieve the desired information. (It may also reflect the fact that directory services attempt to cull out low quality Web sites.)

The meanings of category labels differ somewhat among collections. Most are designed to help organize the documents and to aid in query specification. Unfortunately, users of online bibliographic catalogs rarely use the available subject headings [335, 222]. Hancock-Beaulieu and Drabenstott and Weller, among others, put much of the blame on poor (command line-based) user interfaces which provide little aid for selecting subject labels and require users to scroll through long alphabetic lists. Even with graphical Web interfaces, finding the appropriate place within a category hierarchy can be a time-consuming task, and once a collection has been found using such a representation, an alternative means is required for searching within the site itself.

Most interfaces that depict category hierarchies graphically do so by associating a document directly with the node of the category hierarchy to which it has been assigned. For example, clicking on a category link in Yahoo! brings up a list of documents that have been assigned that category label. Conceptually, the document is stored within the category label. When navigating the results of a search in Yahoo!, the user must look through a list of category labels and guess which one is most likely to contain references to the topic of interest. A wrong path requires backing up and trying again, and remembering which pages contain which information. If the desired information is deep in the hierarchy, or

† `http://www.acm.org/class`

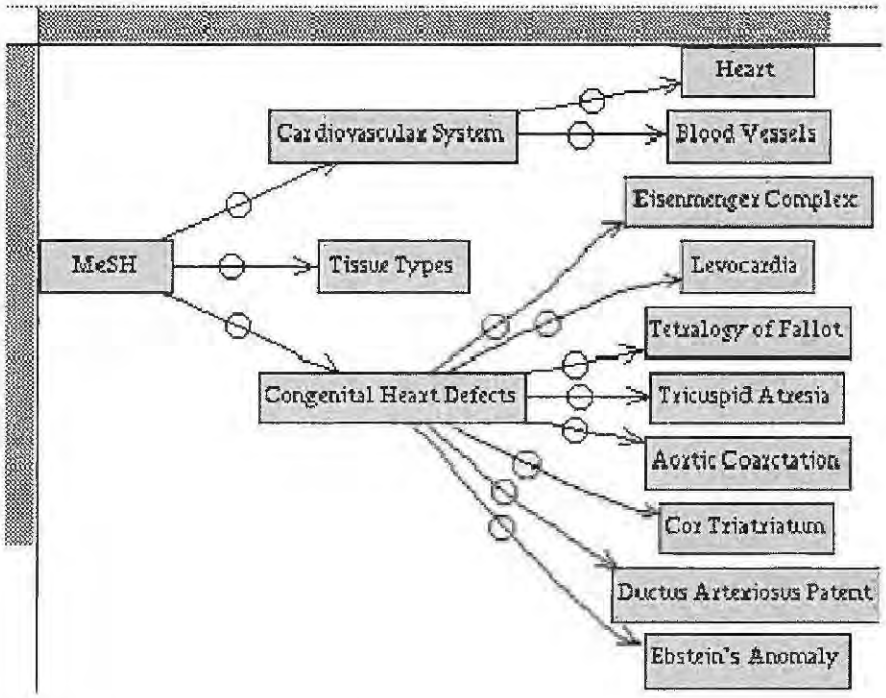


Figure 10.4 The MeSHBrowse interface for viewing category labels hierarchically [453].

not available at all, this can be a time-consuming and frustrating process. Because documents are conceptually stored 'inside' categories, users cannot create queries based on combinations of categories using this interface.

It is difficult to design a good interface to integrate category selection into query specification, in part because display of category hierarchies takes up large amounts of screen space. For example, Internet Grateful Med‡ is a Web-based service that allows an integration of search with display and selection of MeSH category labels. After the user types in the name of a potential category label, a long list of choices is shown in a page. To see more information about a given label, the user selects a link (e.g., Radiation Injuries). The causes the context of the query to disappear because a new Web page appears showing the ancestors of the term and its immediate descendants. If the user attempts to see the siblings of the parent term (Wounds and Injuries) then a new page appears that changes the context again. Radiation Injuries appears as one of many siblings and its children can no long be seen. To go back to the query, the illustration of the category hierarchy disappears.

The MeSHBrowse system [453] allows users to interactively browse a subset of semantically associated links in the MeSH hierarchy. From a given starting point, clicking on a category causes the associated categories to be displayed in a two-dimensional tree representation. Thus only the relevant subset of the

‡ `http://igm.nlm.nih.gov:80/`

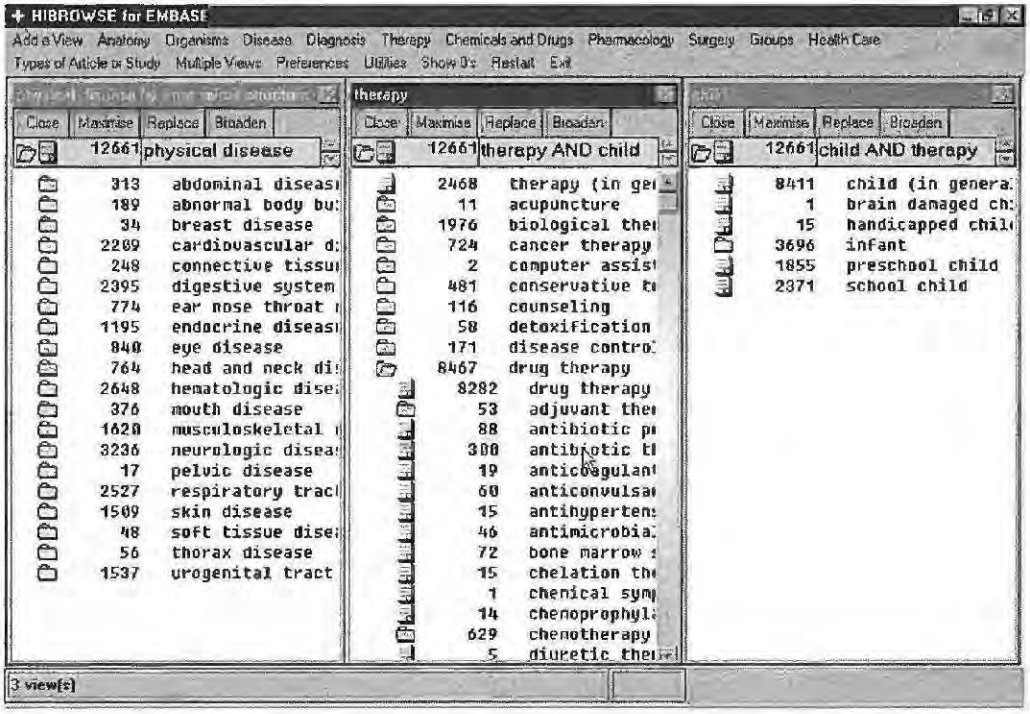


Figure 10.5 The HiBrowse interface for viewing category labels hierarchically and according to facets [646].

hierarchy is shown at one time, making browsing of this very large hierarchy a more tractable endeavor. The interface has the space limitations inherent in a two-dimensional hierarchy display and does not provide mechanisms for search over an underlying document collection. See Figure 10.4.

The HiBrowse system [646] represents category metadata more efficiently by allowing users to display several different subsets of category metadata simultaneously. The user first selects which attribute type (or facet, as attributes are called in this system) to display. For example, the user may first choose the 'physical disease' value for the Disease facet. The categories that appear one level below this are shown along with the number of documents that contain each category. The user can then select other attribute types, such as Therapy and Groups (by age). The number of documents that contain attributes from all three types are shown. If the user now selects a refinement of one of the categories, such as the 'child' value for the Groups attribute, then the number of documents that contain all three selected facet types are shown. At the same time, the number of documents containing the subcategories found below 'physical disease' and 'therapy (general)' are updated to reflect this more restricted specification. See Figure 10.5. A problem with the HiBrowse system is that it requires users to navigate through the category hierarchy, rather than specify queries directly. In other words, query specification is not tightly coupled with display of category metadata. As a solution to some of these problems, the Cat-a-Cone interface [358] will be described in section 10.8.

Automatically Derived Collection Overviews

Many attempts to display overview information have focused on automatically extracting the most common general themes that occur within the collection. These themes are derived via the use of unsupervised analysis methods, usually variants of document clustering. Clustering organizes documents into groups based on similarity to one another; the centroids of the clusters determine the themes in the collections.

The Scatter/Gather browsing paradigm [203, 202] clusters documents into topically-coherent groups, and presents descriptive textual summaries to the user. The summaries consist of topical terms that characterize each cluster generally, and a set of typical titles that hint at the contents of the cluster. Informed by the summaries, the user may select a subset of clusters that seem to be of most interest, and recluster their contents. Thus the user can examine the contents of each subcollection at progressively finer granularity of detail. The reclustering is computed on-the-fly; different themes are produced depending on the documents contained in the subcollection to which clustering is applied. The choice of clustering algorithm influences what clusters are produced, but no one algorithm has been shown to be particularly better than the rest when producing the same number of clusters [816].

A user study [640] showed that the use of Scatter/Gather on a large text collection successfully conveys some of the content and structure of the corpus. However, that study also showed that Scatter/Gather without a search facility was less effective than a standard similarity search for finding relevant documents for a query. That is, subjects allowed only to navigate, not to search over, a hierarchical structure of clusters covering the entire collection were less able to find documents relevant to the supplied query than subjects allowed to write queries and scan through retrieval results.

It is possible to integrate Scatter/Gather with conventional search technology by applying clustering on the results of a query to organize the retrieved documents (see Figure 10.6). An offline experiment [359] suggests that clustering may be more effective if used in this manner. The study found that documents relevant to the query tend to fall mainly into one or two out of five clusters, if the clusters are generated from the top-ranked documents retrieved in response to the query. The study also showed that precision and recall were higher within the best cluster than within the retrieval results as a whole. The implication is that a user might save time by looking at the contents of the cluster with the highest proportion of relevant documents and at the same time avoiding those clusters with mainly non-relevant documents. Thus, clustering of retrieval results may be useful for helping direct users to a subset of the retrieval results that contain a large proportion of the relevant documents.

General themes do seem to arise from document clustering, but the themes are highly dependent on the makeup of the documents within the clusters [359, 357]. The unsupervised nature of clustering can result in a display of topics at varying levels of description. For example, clustering a collection of documents about computer science might result in clusters containing documents about

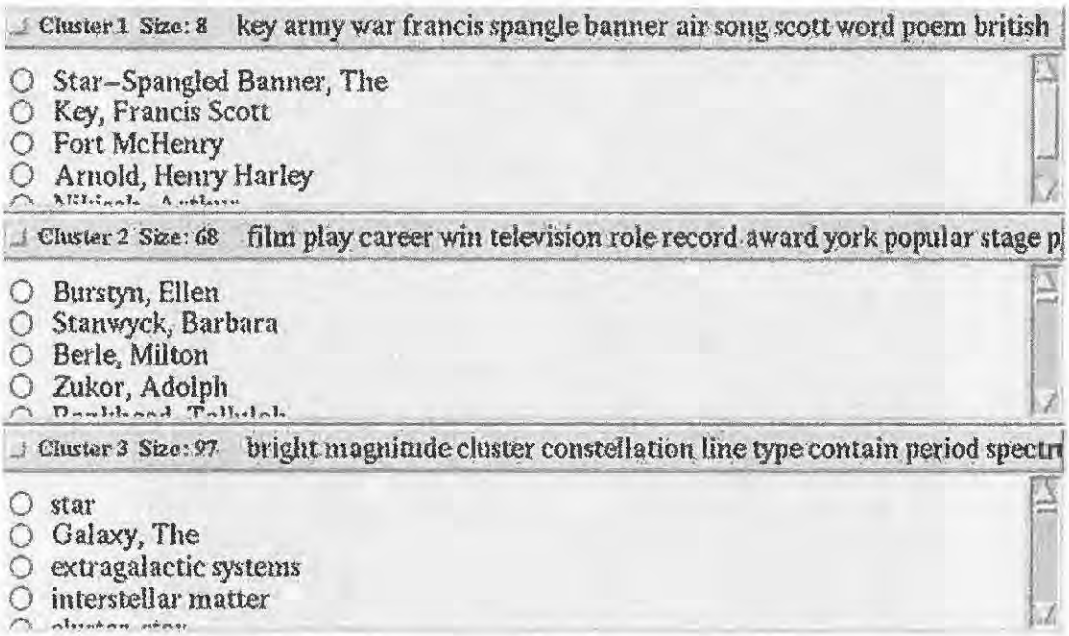


Figure 10.6 Display of Scatter/Gather clustering retrieval results [203].

artificial intelligence, computer theory, computer graphics, computer architecture, programming languages, government, and legal issues. The latter two themes are more general than the others, because they are about topics outside the general scope of computer science. Thus clustering can results in the juxtaposition of very different levels of description within a single display.

Scatter/Gather shows a textual representation of document clusters. Researchers have developed several approaches to map documents from their high dimensional representation in document space into a 2D representation in which each document is represented as a small glyph or icon on a map or within an abstract 2D space. The functions for transforming the data into the lower dimensional space differ, but the net effect is that each document is placed at one point in a scatter-plot-like representation of the space. Users are meant to detect themes or clusters in the arrangement of the glyphs. Systems employing such graphical displays include BEAD [156], the Galaxy of News [671], and ThemeScapes [821]. The ThemeScapes view imposes a three-dimensional representation on the results of clustering (see Figure 10.7). The layout makes use of 'negative space' to help emphasize the areas of concentration where the clusters occur. Other systems display inter-document similarity hierarchically [529, 14], while still others display retrieved documents in networks based on inter-document similarity [262, 761].

Kohonen's feature map algorithm has been used to create maps that graphically characterize the overall content of a document collection or subcollection [520, 163] (see Figure 10.8). The regions of the 2D map vary in size and shape corresponding to how frequently documents assigned to the corresponding themes occur within the collection. Regions are characterized by single words or phrases,

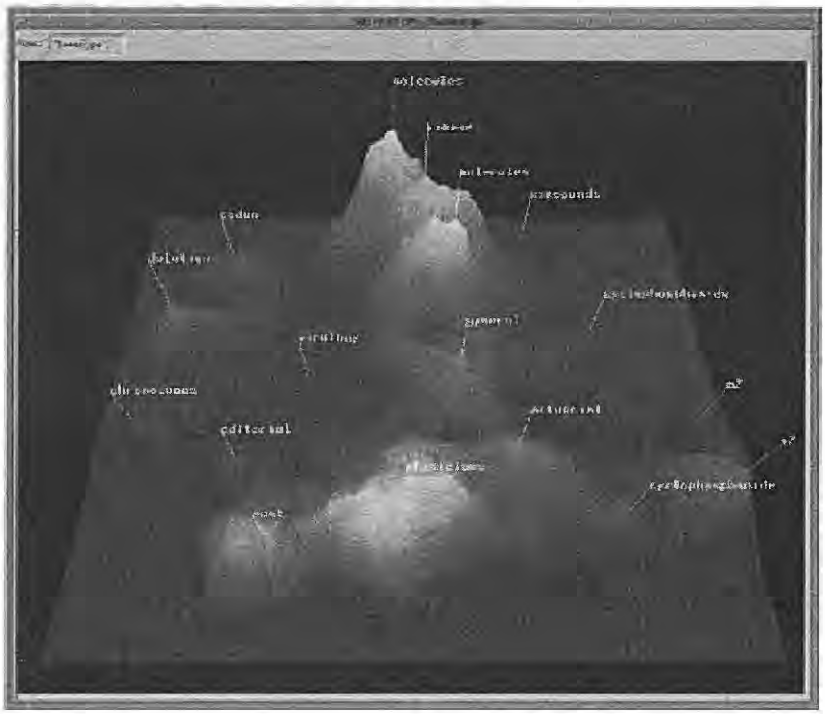


Figure 10.7 A three-dimensional overview based on document clustering [821].

and adjacency of regions is meant to reflect semantic relatedness of the themes within the collection. A cursor moved over a document region causes the titles of the documents most strongly associated with that region to be displayed in a pop-up window. Documents can be associated with more than one region.

Evaluations of Graphical Overviews

Although intuitively appealing, graphical overviews of large document spaces have yet to be shown to be useful and understandable for users. In fact, evaluations that have been conducted so far provide negative evidence as to their usefulness. One study found that for non-expert users the results of clustering were difficult to use, and that graphical depictions (for example, representing clusters with circles and lines connecting documents) were much harder to use than textual representations (for example, showing titles and topical words, as in Scatter/Gather), because documents' contents are difficult to discern without actually reading some text [443].

Another recent study compared the Kohonen feature map overview representation on a browsing task to that of Yahoo! [163]. For one of the tasks, subjects were asked to find an 'interesting' Web page within the entertainment category of Yahoo! and of an organization of the same Web pages into a Kohonen map layout. The experiment varied whether subjects started in Yahoo! or in the graphical map. After completion of the browing task, subjects were asked to attempt to repeat the browse using the other tool. For the subjects that

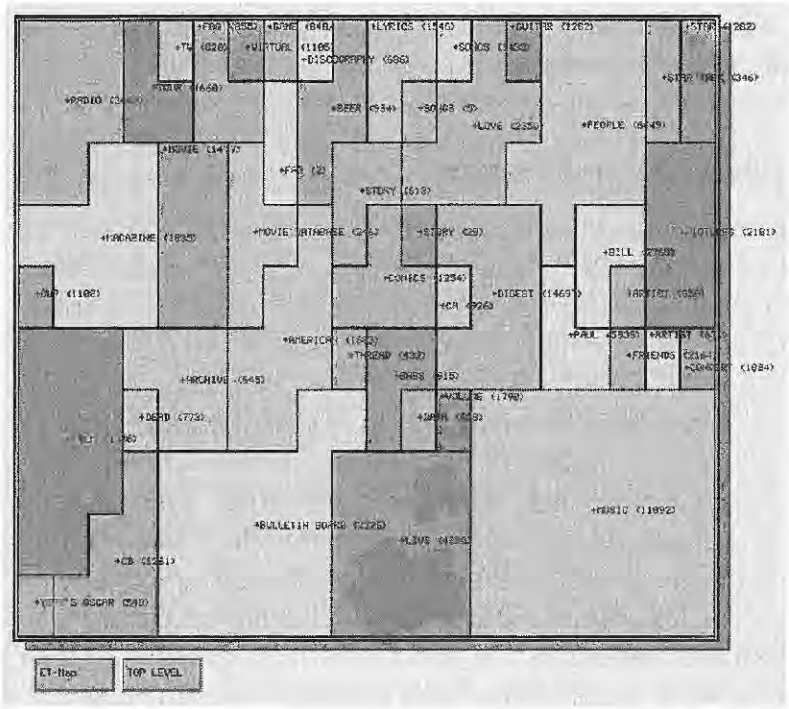


Figure 10.8 A two-dimensional overview created using a Kohonen feature map learning algorithm on Web pages having to do with the topic Entertainment [163].

began with the Kohonen map visualization, 11 out of 15 found an interesting page within ten minutes. Eight of these were able to find the same page using Yahoo!. Of the subjects who started with Yahoo!, 14 out of 16 were able to find interesting home pages. However, only two of the 14 were able to find the page in the graphical map display! This is strong evidence against the navigability of the display and certainly suggests that the simple label view provided by Yahoo! is more useful. However, the map display may be more useful if the system is modified to tightly integrate querying with browsing.

The subjects did prefer some aspects of the map representation. In particular, some liked the ease of being able to jump from one area to another without having to back up as is required in Yahoo!, and some liked the fact that the maps have varying levels of granularity. The subjects disliked several aspects of the display. The experimenters found that some subjects expressed a desire for a visible hierarchical organization, others wanted an ability to zoom in on a subarea to get more detail, and some users disliked having to look through the entire map to find a theme, desiring an alphabetical ordering instead. Many found the single-term labels to be misleading, in part because they were ambiguous (one region called 'BILL' was thought to correspond to a person's name rather than counting money).

The authors concluded that this interface is more appropriate for casual browsing than for search. In general, unsupervised thematic overviews are perhaps most useful for giving users a 'gist' of the kinds of information that can be

found within the document collection, but generally have not been shown to be helpful for use in the information access process.

Co-citation Clustering for Overviews

Citation analysis has long been recognized as a way to show an overview of the contents of a collection [812]. The main idea is to determine 'centrally-located' documents based on co-citation patterns. There are different ways to determine citation patterns: one method is to measure how often two articles are cited together by a third. Another alternative is to pair articles that cite the same third article. In both cases the assumption is that the paired articles share some commonalities. After a matrix of co-citations is built, documents are clustered based on the similarity of their co-citation patterns. The resulting clusters are interpreted to indicate dominant themes within the collection. Clustering can focus on the authors of the documents rather than the contents, to attempt to identify central authors within a field. This idea has recently been implemented using Web-based documents in the Referral Web project [432]. The idea has also been applied to Web pages, using Web link structure to identify major topical themes among Web pages [485, 639]. A similar idea, but computed a different way, is used to explicitly identify pages that act as good starting points for particular topics (called 'authority pages' by Kleinberg [444]).

10.4.3 Examples, Dialogs, and Wizards

Another way to help users get started is to start them off with an example of interaction with the system. This technique is also known as *retrieval by reformulation*. An early version of this idea is embodied in the Rabbit system [818] which provides graphical representations of example database queries. A general framework for a query is shown to the user who then modifies it to construct a partially complete description of what they want. The system then shows an example of the kind of information available that matches this partial description. For instance, if a user searching a computer products database indicates an interest in disks, an example item is retrieved with its disk descriptors filled in. The user can use or modify the displayed descriptors, and iterate the procedure.

The idea of retrieval by reformulation has been developed further and extended to the domains of user interface development [581] and software engineering [669]. The Helgon system [255] is a modern variant of this idea applied to bibliographic database information. In Helgon, users begin by navigating a hierarchy of topics from which they select structured examples, according to their interests. If a feature of an example is inappropriately set, the user can modify the feature to indicate how it would appear in the desired information. Unfortunately, in tests with users, the system was found to be problematic. Users had problems with the organization of the hierarchy, and found that searching for a useful example by critiquing an existing one to be tedious. This result

underscores an unfortunate difficulty with examples and dialogues: that of getting the user to the right starting dialogue or the right example strategy becomes a search problem in itself. (How to index prior examples is studied extensively in the case-based reasoning (CBR) literature [492, 449].)

A more dynamic variation on this theme is the interactive dialog. Dialog-based interfaces have been explored since the early days of information retrieval research, in an attempt to mimic the interaction provided by a human search intermediary (e.g., a reference librarian). Oddy did early work in the THOMAS system, which provided a question and answer session within a command-line-based interface [615]. More recently, Belkin *et al.* have defined quite elaborate dialog interaction models [75] although these have not been assessed empirically to date.

The DLITE system interface [192] uses an animated focus-plus-context dialog as a way to acquaint users with standard sequences of operations within the system. Initially an outline of all of the steps of the dialog is shown as a list. The user can expand the explanation of any individual step by clicking on its description. The user can expand out the entire dialog to see what questions are coming next, and then collapse it again in order to focus on the current tactic.

A more restricted form of dialog that has become widely used in commercial products is that of the *wizard*. This tool helps users in time-limited tasks, but does not attempt to overtly teach the processes required to complete the tasks. The wizard presents a step-by-step shortcut through the sequence of menu choices (or tactics) that a user would normally perform in order to get a job done, reducing user input to just a few choices with default settings [636]. A recent study [145] found wizards to be useful for goals that require many steps, for users who lack necessary domain knowledge (for example, a restaurant owner installing accounting software), and when steps must be completed in a fixed sequence (for example, a procedure for hiring personnel). Properties of successful wizards included allowing users to rerun a wizard and modify their previous work, showing an overview of the supported functions, and providing lucid descriptions and understandable outcomes for choices. Wizards were found not to be helpful when the interface did not solve a problem effectively (for example, a commercial wizard for setting up a desktop search index requests users to specify how large to make the index, but supplies no information about how to make this decision). Wizards were also found not to be helpful when the goal was to teach the user how to use the interface, and when the wizard was not user-tested. It maybe the case that information access is too variable a process for the use of wizards.

A *guided tour* leads a user through a sequence of navigational choices through hypertext links, presenting the nodes in a logical order for some goal. In a dynamic tour, only relevant nodes are displayed, as opposed to the static case where the author decides what is relevant before the users have even formulated their queries [329]. A recent application is the Walden Paths project which enables teachers to define instructionally useful paths through pages found on the Web [289]. This approach has not been commonly used to date for

information access but could be a promising direction for acquainting users with search strategies in large hyperlinked systems.

10.4.4 Automated Source Selection

Human-computer interfaces for helping guide users to appropriate sources is a wide open area for research. It requires both eliciting the information need from users and understanding which needs can be satisfied by which sources. An ambitious approach is to build a model of the source and of the information need of the user and try to determine which fit together best. User modeling systems and intelligent tutoring systems attempt to do this both for general domains [204, 814] and for online help systems [378].

A simpler alternative is to create a representation of the contents of information sources and match this representation against the query specification. This approach is taken by GlOSS, a system which tries to determine in advance the best bibliographic database to send a search request to, based on the terms in the query [765]. The system uses a simple analysis of the combined frequencies of the query words within the individual collections. The SavvySearch system [383] takes this idea a step further, using actions taken by users after a query to decide how to decrease or increase the ranking of a search engine for a particular query (see also Chapter 13).

The flip side to automatically selecting the best source for a query is to automatically send a query to multiple sources and then combine the results from the various systems in some way. Many metasearch engines exist on the Web. How to combine the results effectively is an active area of research, sometimes known as collection fusion [63, 767, 388].

10.5 Query Specification

To formulate a query, a user must select collections, metadata descriptions, or information sets against which the query is to be matched, and must specify words, phrases, descriptors, or other kinds of information that can be compared to or matched against the information in the collections. As a result, the system creates a set of documents, metadata, or other information type that match the query specification in some sense and displays the results to the user in some form.

Shneiderman [725] identifies five primary human-computer interaction styles. These are: *command language*, *form fillin*, *menu selection*, *direct manipulation*, and *natural language*.§ Each technique has been used in query specification interfaces and each has advantages and disadvantages. These are described below in the context of Boolean query specification.

§ This list omits non-visual modalities, such as audio.

10.5.1 Boolean Queries

In modern information access systems the matching process usually employs a statistical ranking algorithm. However, until recently most commercial full-text systems and most bibliographic systems supported only Boolean queries. Thus the focus of many information access studies has been on the problems users have in specifying Boolean queries. Unfortunately, studies have shown time and again that most users have great difficulty specifying queries in Boolean format and often misjudge what the results will be [111, 322, 558, 841].

Boolean queries are problematic for several reasons. Foremost among these is that most people find the basic syntax counter-intuitive. Many English-speaking users assume everyday semantics are associated with Boolean operators when expressed using the English words AND and OR, rather than their logical equivalents. To inexperienced users, using AND implies the widening of the scope of the query, because more kinds of information are being requested. For instance, 'dogs and cats' may imply a request for documents about dogs and documents about cats, rather than documents about both topics at once. 'Tea or coffee' can imply a mutually exclusive choice in everyday language. This kind of conceptual problem is well documented [111, 322, 558, 841]. In addition, most query languages that incorporate Boolean operators also require the user to specify complex syntax for other kinds of connectors and for descriptive metadata. Most users are not familiar with the use of parentheses for nested evaluation, nor with the notions associated with operator precedence.

By serving a massive audience possessing little query-specification experience, the designers of World Wide Web search engines have had to come up with more intuitive approaches to query specification. Rather than forcing users to specify complex combinations of ANDs and ORs, they allow users to choose from a selection of common simple ways of combining query terms, including 'all the words' (place all terms in a conjunction) and 'any of the words' (place all terms in a disjunction).

Another Web-based solution is to allow syntactically-based query specification, but to provide a simpler or more intuitive syntax. The '+' prefix operator gained widespread use with the advent of its use as a mandatory specifier in the Altavista Web search engine. Unfortunately, users can be misled to think it is an infix AND rather than a prefix mandatory operator, and thus assume that 'cat + dog' will only retrieve articles containing both terms (where in fact this query requires dog but allows cat to be optional).

Another problem with pure Boolean systems is they do not rank the retrieved documents according to their degree of match to the query. In the pure Boolean framework a document either satisfies the query or it does not. Commercial systems usually resort to ordering documents according to some kind of descriptive metadata, usually in reverse chronological order. (Since these systems usually index timely data corresponding to newspaper and news wires, date of publication is often one of the most salient features of the document.) Web-based systems usually rank order the results of Boolean queries using statistical algorithms and Web-specific heuristics.

10.5.2 From Command Lines to Forms and Menus

Aside from conceptual misunderstandings of the logical meaning of AND and OR, another part of the problem with pure Boolean query specification in online bibliographic systems is the arbitrariness of the syntax and the contextlessness nature of the TTY-based interface in which they are predominantly available. Typically input is typed at a prompt and is of a form something like the following:

> COMMAND ATTRIBUTE value {BOOLEAN-OPERATOR ATTRIBUTE value}*
> e.g.,
> FIND PA darwin AND TW species OR TW descent
> or
> FIND TW Mt St. Helens AND DATE 1981

(These examples are derived from the syntax of the telnet interface to the University of California Melvyl system [526].) The user must remember the commands and attribute names, which are easily forgotten between usages of the system [553]. Compounding this problem, despite the fact that the command languages for the two main online bibliographic systems at UC Berkeley have different but very similar syntaxes, after more than ten years one of the systems still reports an error if the author field is specified as PA instead of PN, as is done in the other system. This lack of flexibility in the syntax is characteristic of interfaces designed to suit the system rather than its users.

The new Web-based version of Melvyl‖ provides form fillin and menu selection so the user no longer has to remember the names and types of attributes available. Users select metadata types from listboxes and attributes are shown explicitly, allowing selection as an alternative to specification. For example, the 'search type' field is adjacent to an entry form in which users can enter keywords, and a choice between AND and NOT is provided adjacent to a list of the available document types (editorial, feature, etc.). Only the metadata associated with a given collection is shown in the context of search over that collection. (Unfortunately the system is restricted to searching over only one database at a time. It does however provide a mechanism for applying a previously executed search to a new database.) See Figure 10.9.

The Web-based version of Melvyl also allows retention of context between searches, storing prior results in tables and hyperlinking these results to lists containing the retrieved bibliographic information. Users can also modify any of the previously submitted queries by selecting a checkbox beside the record of the query. The graphical display makes explicit and immediate many of the powerful options of the system that most users would not learn using the command-line version of the interface.

Bit-mapped displays are an improvement over command-line interface, but do not solve all the problems. For example, a blank entry form is in some ways

‖ http://www.melvyl.ucop.edu/

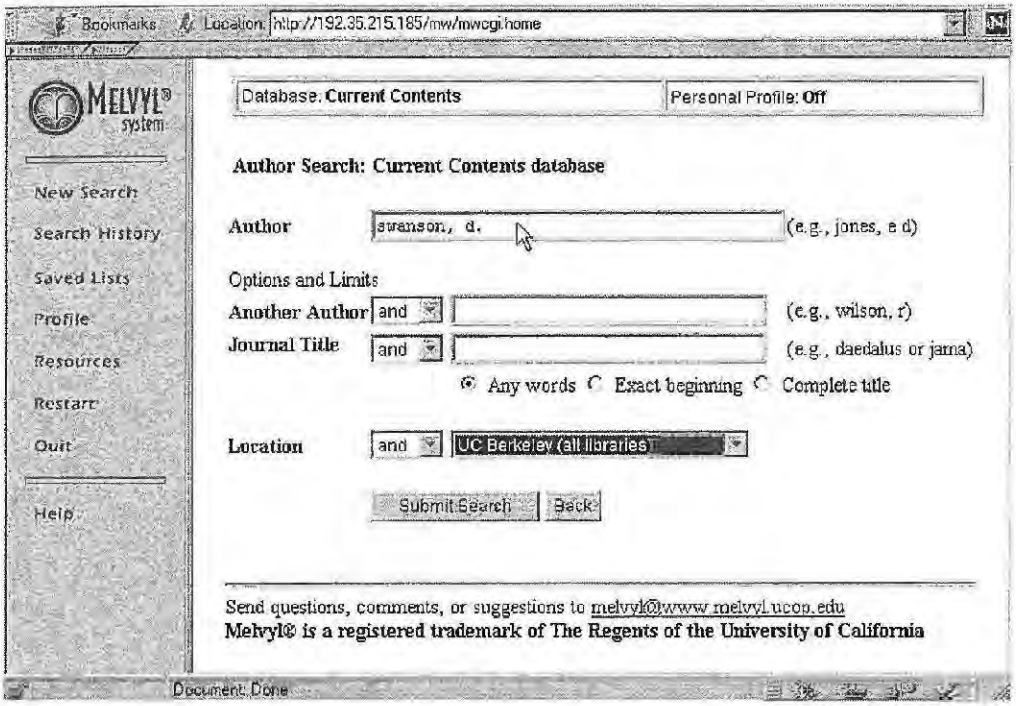


Figure 10.9 A view of query specification in the Web-based version of the Melvyl bibliographic catalog. Copyright © 1998, The Regents of the University of California.

not much better than a TTY prompt, because it does not provide the user with clues about what kinds of terms should be entered.

10.5.3 Faceted Queries

Yet another problem with Boolean queries is that their strict interpretation tends to yield result sets that are either too large, because the user includes many terms in a disjunct, or are empty, because the user conjoins terms in an effort to reduce the result set. This problem occurs in large part because the user does not know the contents of the collection or the role of terms within the collection.

A common strategy for dealing with this problem, employed in systems with command-line-based interfaces like DIALOG's, is to create a series of short queries, view the number of documents returned for each, and combine those queries that produce a reasonable number of results. For example, in DIALOG, each query produces a resulting set of documents that is assigned an identifying name. Rather than returning a list of titles themselves, DIALOG shows the set number with a listing of the number of matched documents. Titles can be shown by specifying the set number and issuing a command to show the titles. Document sets that are not empty can be referred to by a set name and combined with AND operations to produce new sets. If this set in turn is too small, the user can back up and try a different combination of sets, and this process is repeated in pursuit of producing a reasonably sized document set.

This kind of query formulation is often called a *faceted* query, to indicate that the user's query is divided into topics or facets, each of which should be

present in the retrieved documents [553, 348]. For example, a query on drugs for the prevention of osteoporosis might consist of three facets, indicated by the disjuncts

(osteoporosis OR 'bone loss')
(drugs OR pharmaceuticals)
(prevention OR cure)

This query implies that the user would like to view documents that contain all three topics.

A technique to impose an ordering on the results of Boolean queries is what is known as *post-coordinate* or *quorum-level* ranking [700, Ch. 8]. In this approach, documents are ranked according to the size of the subset of the query terms they contain. So given a query consisting of 'cats,' 'dogs,' 'fish,' and 'mice,' the system would rank a document with at least one instance of 'cats,' 'dogs,' and 'fish' higher than a document containing 30 occurrences of 'cats' but no occurrences of the other terms.

Combining faceted queries with quorum ranking yields a situation intermediate between full Boolean syntax and free-form natural language queries. An interface for specifying this kind of interaction can consist of a list of entry lines. The user enters one topic per entry line, where each topic consists of a list of semantically related terms that are combined in a disjunct. Documents that contain at least one term from each facet are ranked higher than documents containing terms only from one or a few facets. This helps ensure that documents which contain discussions of several of the user's topics are ranked higher than those that contain only one topic. By only requiring that one term from each facet be matched, the user can specify the same concept in several different ways in the hopes of increasing the likelihood of a match. If combined with graphical feedback about which subsets of terms matched the document, the user can see the results of a quorum ranking by topic rather than by word. Section 10.6 describes the TileBars interface which provides this type of feedback.

This idea can be extended yet another step by allowing users to weight each facet. More likely to be readily usable, however, is a default weighting in which the facet listed highest is assigned the most weight, the second facet is assigned less weight, and so on, according to some distribution over weights.

10.5.4 Graphical Approaches to Query Specification

Direct manipulation interfaces provide an alternative to command-line syntax. The properties of direct manipulation are [725, p.205]: (1) continuous representation of the object of interest, (2) physical actions or button presses instead of complex syntax, and (3) rapid incremental reversible operations whose impact on the object of interest is immediately visible. Direct manipulation interfaces often evoke enthusiasm from users, and for this reason alone it is worth exploring their use. Although they are not without drawbacks, they are easier to use than other methods for many users in many contexts.

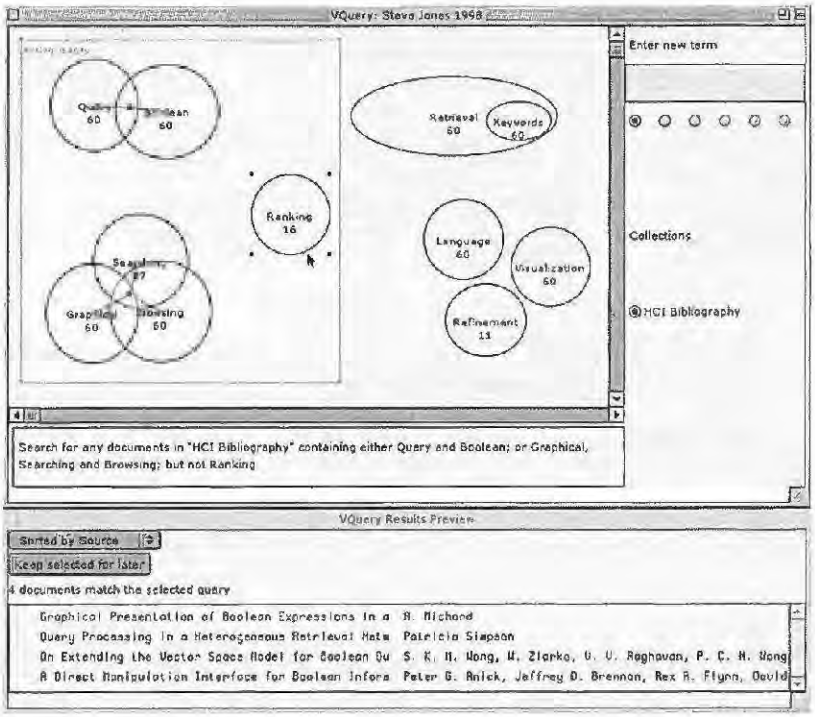


Figure 10.10 The VQuery Venn diagram visualization for Boolean query specification [417].

Several variations of graphical interfaces, both directly manipulable and static, have been developed for simplifying the specification of Boolean syntax. User studies tend to reveal that these graphical interfaces are more effective in terms of accuracy and speed than command-language counterparts. Three such approaches are described below.

Graphical depictions of *Venn diagrams* have been proposed several times as a way to improve Boolean query specification. A query term is associated with a ring or circle and intersection of rings indicates conjunction of terms. Typically the number of documents that satisfy the various conjuncts are displayed within the appropriate segments of the diagram. Several studies have found such interfaces more effective than their command-language-based syntax [417, 368, 558]. Hertzum and Frokjaer found that a simple Venn diagram representation produced faster and more accurate results than a Boolean query syntax. However, a problem with this format is the limitations on the complexity of the expression. For example, a maximum of three query terms can be ANDed together in a standard Venn diagram. Innovations have been designed to get around this problem, as seen in the VQuery system [417] (see Figure 10.10). In VQuery, a direct manipulation interface allows users to assign any number of query terms to ovals. If two or more ovals are placed such that they overlap with one another, and if the user selects the area of their intersection, an AND is implied among those terms. (In Figure 10.10, the term 'Query' is conjoined with 'Boolean'.) If the user selects outside the area of intersection but within the ovals, an OR is implied among the corresponding terms. A NOT operation

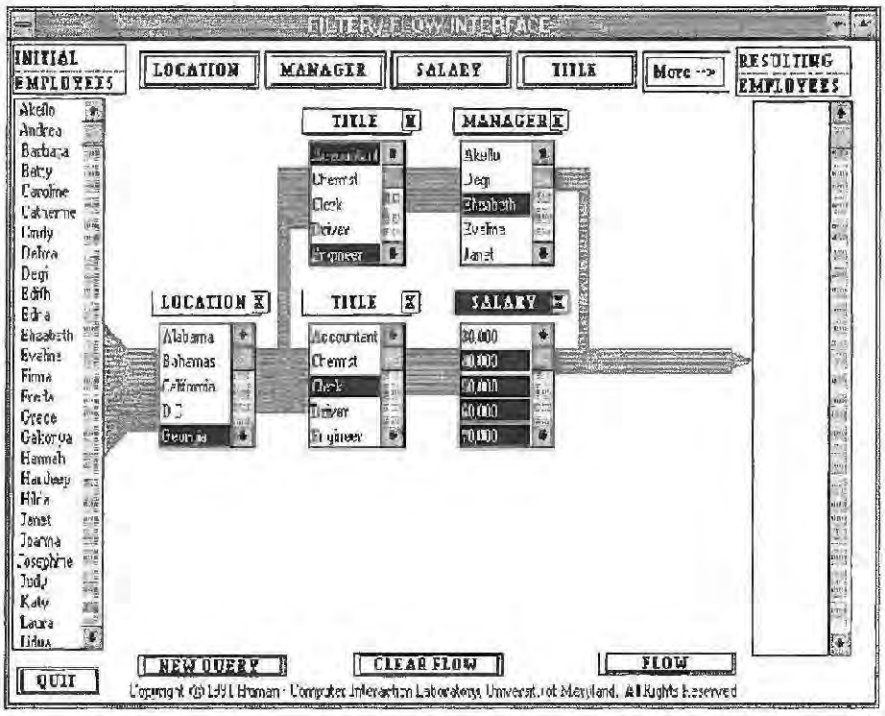


Figure 10.11 The filter-flow visualization for Boolean query specification [841].

is associated with any term whose oval appears in the active area of the display but which remains unselected (in the figure, NOT 'Ranking' has been specified). An active area indicates the current query; all groups of ovals within the active area are considered part of a conjunction. Ovals containing query terms can be moved out of the active area for later use.

Young and Shneiderman [841] found improvements over standard Boolean syntax by providing users with a direct manipulation *filter-flow* model. The user is shown a scrollable list of attribute types on the left-hand side and selects attributes from another list of attribute types shown across the top of the screen. Clicking on an attribute name causes a listbox containing values for those attributes to be displayed in the main portion of the screen. The user then selects which values of the attributes to let the flow go through. Placing two or more of these attributes in sequence creates the semantics of a conjunct over the selected values. Placing two or more of these in parallel creates the semantics of a disjunct. The number of documents that match the query at each point is indicated by the width of the 'water' flowing from one attribute to the next. (See Figure 10.11.) A conjunct can reduce the amount of flow. The items that match the full query are shown on the far right-hand side. A user study found that fewer errors were made using the filter flow model than a standard SQL database query. However, the examples and study pertain only to database querying rather than information access, since the possible query terms for information access cannot be represented realistically in a scrollable list. This interface could perhaps be modified to better suit information access applications by having the user supply initial query terms, and using the attribute selection facility to show those terms

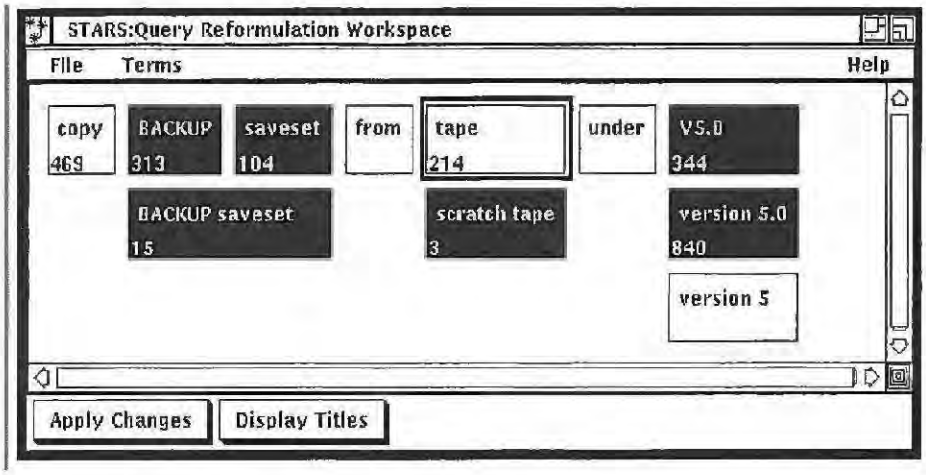


Figure 10.12 A block-oriented diagram visualization for Boolean query specification [21].

that are conceptually related to the query terms. Another alternative is to use this display as a category metadata selection interface (see Section 10.4).

Anick *et al.* [21] describe another innovative direct manipulation interface for Boolean queries. Initially the user types a natural language query which is automatically converted to a representation in which each query term is represented within a block. The blocks are arranged into rows and columns (See Figure 10.12). If two or more blocks appear along the same row they are considered to be ANDed together. Two or more blocks within the same column are ORed. Thus the user can represent a technical term in multiple ways within the same query, providing a kind of faceted query interface. For example, the terms 'version 5', 'version 5.0', and 'v5' might be shown in the same column. Users can quickly experiment with different combinations of terms within Boolean queries simply by activating and deactivating blocks. This facility also allows users to have multiple representations of the same term in different places throughout the display, thus allowing rapid feedback on the consequences of specifying various combinations of query terms. Informal evaluation of the system found that users were able to learn to manipulate the interface quickly and enjoyed using it. It was not formally compared to other interaction techniques [21].

This interface provides a kind of *query preview*: a low cost, rapid turnaround visualization of the results of many variations on a query [643]. Another example of query previewing can be found in some help systems, which show all the words in the index whose first letters match the characters that the user has typed so far. The more characters typed, the fewer possible matches become available. The HiBrowse system described above [646] also provides a kind of preview for viewing category hierarchies and facets, showing how many documents would be matched if a category one level below the current one were selected. It perhaps could be improved by showing the consequences of more combinations of categories in an animated manner. If based on prior action and interests of the user, query previewing may become more generally applicable for information access interfaces.

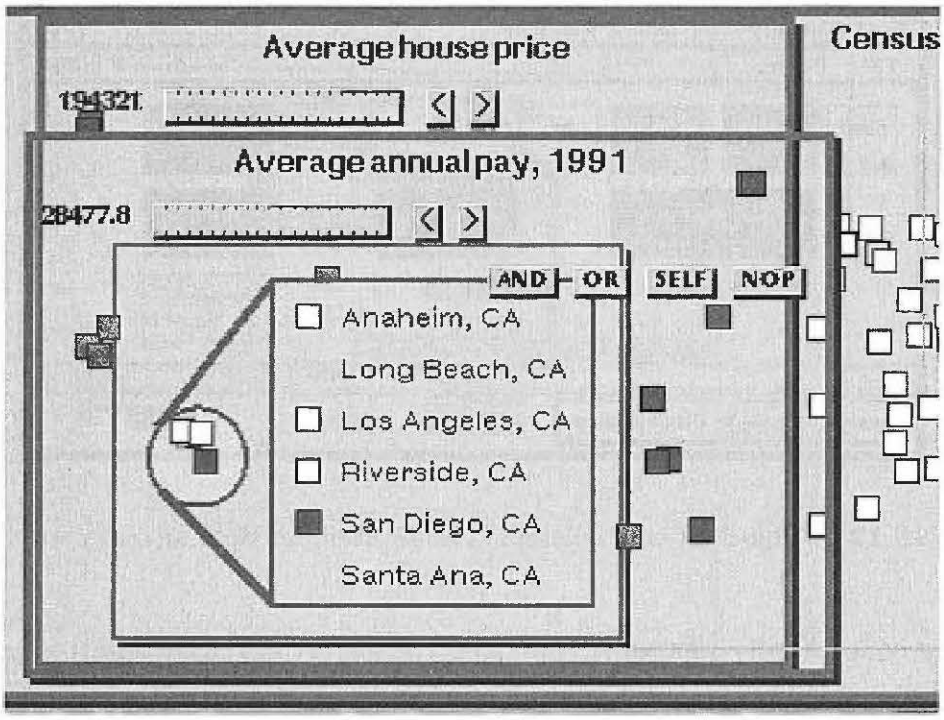


Figure 10.13 A magic lens interface for query specification (courtesy of Ken Fishkin).

A final example of a graphical approach to query specification is the use of magic lenses. Fishkin and Stone have suggested an extension to the usage of this visualization tool for the specification of Boolean queries [256]. Information is represented as lists or icons within a 2D space. Lenses act as filters on the document set. (See Figure 10.13.) For example, a word can be associated with a transparent lens. When this lens is placed over an iconic representation of a set of documents, it can cause all documents that do not contain a given word to disappear. If a second lens representing another word is then laid over the first, the lenses combine to act as a conjunction of the two words with the document set, hiding any documents that do not contain both words. Additional information can be adjusted dynamically, such as a minimum threshold for how often the term occurs in the documents, or an on-off switch for word stemming. For example, Figure 10.13 shows a disjunctive query that finds cities with relatively low housing prices or high annual salaries. One lens 'calls out' a clump of southern California cities, labeling each. Above that is a lens screening for cities with average house price below $194,321 (the data is from 1990), and above this one is a lens screening for cities with average annual pay above $28,477. This approach, while promising, has not been evaluated in an information access setting.

10.5.5 Phrases and Proximity

In general, proximity information can be quite effective at improving precision of searches. On the Web, the difference between a single-word query and a two-word

exact phrase match can mean the difference between an unmanageable mess of retrieved documents and a short list with mainly relevant documents.

A large number of methods for specifying phrases have been developed. The syntax in LEXIS-NEXIS requires the proximity range to be specified with an infix operator. For example, 'white w/3 house' means 'white within 3 words of house, independent of order.' Exact proximity of phrases is specified by simply listing one word beside the other, separated by a space. A popular method used by Web search engines is the enclosure of the terms between quotation marks. Shneiderman *et al.* [726] suggest providing a list of entry labels, as suggested above for specifying facets. The difference is, instead of a disjunction, the terms on each line are treated as a phrase. This is suggested as a way to guide users to more precise query specification.

The disadvantage of these methods is that they require exact match of phrases, when it is often the case (in English) that one or a few words comes between the terms of interest. For example, in most cases the user probably wants 'president' and 'lincoln' to be adjacent, but still wants to catch cases of the sort 'President Abraham Lincoln.' Another consideration is whether or not stemming is performed on the terms included in the phrase. The best solution may be to allow users to specify exact phrases but treat them as small proximity ranges, with perhaps an exponential fall-off in weight in terms of distance of the terms. This has been shown to be a successful strategy in non-interactive ranking algorithms [174]. It has also been shown that a combination of quorum ranking of faceted queries with the restriction that the facets occur within a small proximity range can dramatically improve precision of results [356, 566].

10.5.6 Natural Language and Free Text Queries

Statistical ranking algorithms have the advantage of allowing users to specify queries naturally, without having to think about Boolean or other operators. But they have the drawback of giving the user less feedback about and control over the results. Usually the result of a statistical ranking is the listing of documents and the association of a score, probability, or percentage beside the title. Users are given little feedback about why the document received the ranking it did and what the roles of the query terms are. This can be especially problematic if the user is particularly interested in one of the query terms being present.

One search strategy that can help with this particular problem with statistical ranking algorithms is the specification of 'mandatory' terms within the natural language query. This in effect helps the user control which terms are considered important, rather than relying on the ranking algorithm to correctly weight the query terms. But knowing to include a mandatory specification requires the user to know about a particular command and how it works.

The preceding discussion assumes that a natural language query entered by the user is treated as a bag of words, with stopwords removed, for the purposes of document match. However, some systems attempt to parse natural language queries in order to extract concepts to match against concepts in the

text collection [399, 552, 748].

Alternatively, the natural language syntax of a question can be used to attempt to answer the question. (Question answering in information access is different than that of database management systems, since the information desired is encoded within the text of documents rather than specified by the database schema.) The Murax system [463] determines from the syntax of a question if the user is asking for a person, place, or date. It then attempts to find sentences within encyclopedia articles that contain noun phrases that appear in the question, since these sentences are likely to contain the answer to the question. For example, given the question 'Who was the Pulitzer Prize-winning novelist that ran for mayor of New York City?,' the system extracts the noun phrases 'Pulitzer Prize,' 'winning novelist,' 'mayor,' and 'New York City.' It then looks for proper nouns representing people's names (since this is a 'who' question) and finds, among others, the following sentences:

> The Armies of the Night (1968), a personal narrative of the 1967 peace march on the Pentagon, won **Mailer** the **Pulitzer Prize** and the National Book Award.
>
> In 1969 **Mailer** ran unsuccessfully as an independent candidate for **mayor** of **New York City**.

Thus the two sentences link together the relevant noun phrases and the system hypothesizes (correctly) from the title of the article in which the sentences appear that Norman Mailer is the answer.

Another approach to automated question answering is the FAQ finder system which matches question-style queries against question-answer pairs on various topics [130]. The system uses a standard IR search to find the most likely FAQ (frequently asked questions) files for the question and then matches the terms in the question against the question portion of the question-answer pairs.

A less automated approach to question answering can be found in the Ask Jeeves system [34]. This system makes use of hand-picked Web sites and matches these to a predefined set of question types. A user's query is first matched against the question types. The user selects the most accurate rephrase of their question and this in turn is linked to suggested Web sites. For example, the question 'Who is the leader of Sudan?' is mapped into the question type 'Who is the head of state of X (Sudan)?' where the variable is replaced by a listbox of choices, with Sudan the selected choice in this case. This is linked to a Web page that lists current heads of state. The system also automatically substitutes in the name 'Sudan' in a query against that Web page, thus bringing the answer directly to the user's attention. The question is also sent to standard Web search engines. However, a system is only as good as its question templates. For example a question 'Where can I find reviews of spas in Calistoga?' matches the question 'Where can I find X (reviews) of activities for children aged Y (1)?' and 'Where can I find a concise encyclopedia article on X (hot springs)?'

10.6 Context

This section discusses interface techniques for placing the current document set in the *context* of other information types, in order to make the document set more understandable. This includes showing the relationship of the document set to query terms, collection overviews, descriptive metadata, hyperlink structure, document structure, and to other documents within the set.

10.6.1 Document Surrogates

The most common way to show results for a query is to list information about documents in order of their computed relevance to the query. Alternatively, for pure Boolean ranking, documents are listed according to a metadata attribute, such as date. Typically the document list consists of the document's title and a subset of important metadata, such as date, source, and length of the article. In systems with statistical ranking, a numerical score or percentage is also often shown alongside the title, where the score indicates a computed degree of match or probability of relevance. This kind of information is sometimes referred to as a *document surrogate*. See Figure 10.14 from [824].

Some systems provide users with a choice between a short and a detailed view. The detailed view typically contains a summary or abstract. In bibliographic systems, the author-written or service-written abstract is shown. Web search engines automatically generate excerpts, usually extracting the first few lines of non-markup text in the Web page.

In most interfaces, clicking on the document's title or an iconic representation of the document shown beside the title will bring up a view of the document itself, either in another window on the screen, or replacing the listing of search results. (In traditional bibliographic systems, the full text was unavailable online, and only bibliographic records could be readily viewed.)

10.6.2 Query Term Hits Within Document Content

In systems in which the user can view the full text of a retrieved document, it is often useful to *highlight* the occurrences of the terms or descriptors that match those of the user's query. It can also be useful for the system to scroll the view of the document to the first passage that contains one or more of the query terms, and highlight the matched terms in a contrasting color or reverse video. This display is thought to help draw the user's attention to the parts of the document most likely to be relevant to the query. Highlighting of query terms has been found time and again to be a useful feature for information access interfaces [481],[542, p.31]. Color highlighting has also recently been found to be useful for scanning lists of bibliographic records [52].

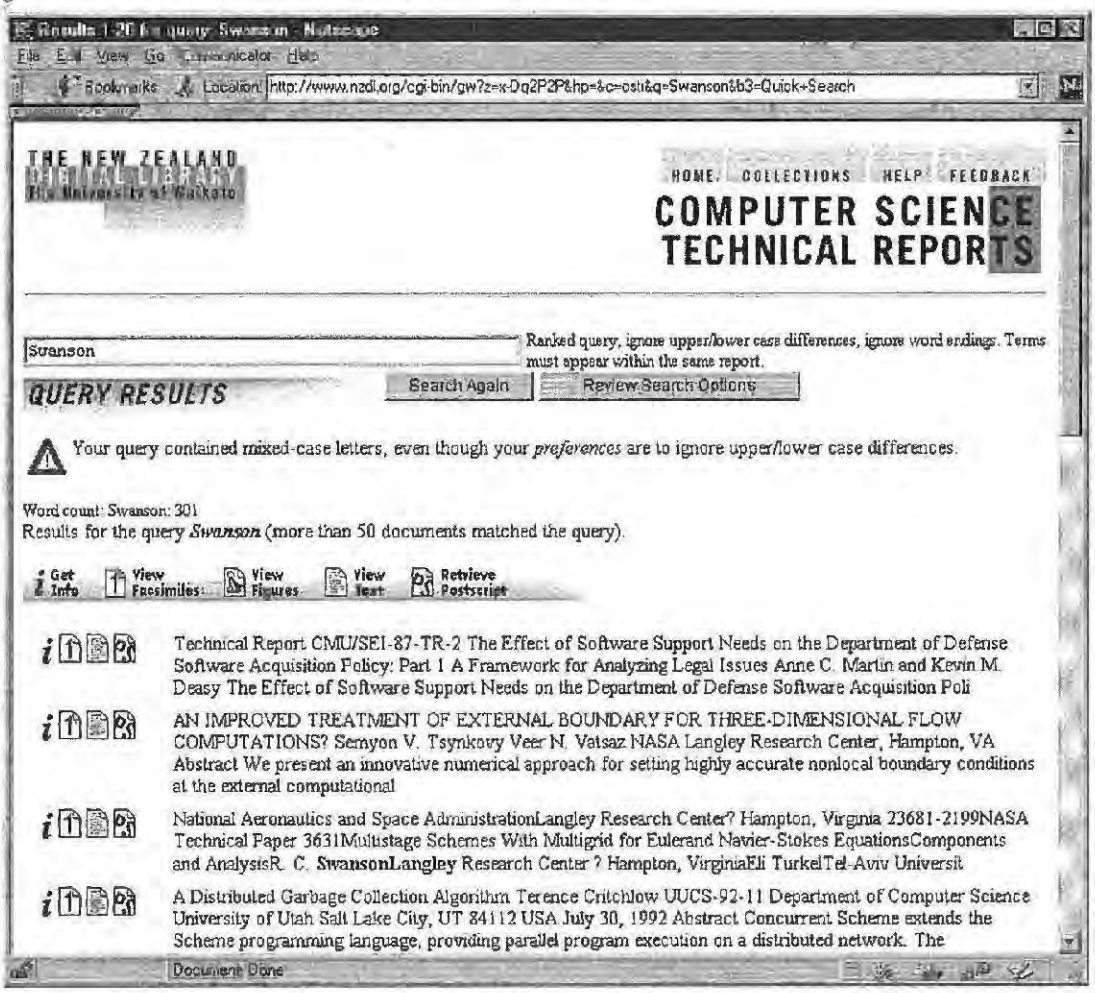


Figure 10.14 An example of a ranked list of titles and other document surrogate information [824].

KWIC

A facility related to highlighting is the *keyword-in-context* (KWIC) document surrogate. Sentence fragments, full sentences, or groups of sentences that contain query terms are extracted from the full text and presented for viewing along with other kinds of surrogate information (such as document title and abstract). Note that a KWIC listing is different than an abstract. An abstract summarizes the main topics of the document but might not contain references to the terms within the query. A KWIC extract shows sentences that summarize the ways the query terms are used within the document. This display can show not only which subsets of query terms occur in the retrieved documents, but also the context they appear in with respect to one another.

Tradeoff decisions must be made between how many lines of text to show and which lines to display. It is not known which contexts are best selected for viewing but results from text summarization research suggest that the best fragments to show are those that appear near the beginning of the document and that contain the largest subset of query terms [464]. If users have specified which

terms are more important than others, then those fragments containing important terms should be shown before those that contain only less important terms. However, to help retain coherence of the excerpts, selected sentences should be shown in order of their occurrence in the original document, independent of how many search terms they contain.

The KWIC facility is usually not shown in Web search result display, most likely because the system must have a copy of the original document available from which to extract the sentences containing the search terms. Web search engines typically only retain the index without term position information. Systems that index individual Web sites can show KWIC information in the document list display.

TileBars

A more compact form of query term hit display is made available through the TileBars interface. The user enters a query in a faceted format, with one topic per line. After the system retrieves documents (using a quorum or statistical ranking algorithm), a graphical bar is displayed next to the title of each document showing the degree of match for each facet. TileBars thus illustrate at a glance which passages in each article contain which topics – and moreover, how frequently each topic is mentioned (darker squares represent more frequent matches).

Each document is represented by a rectangular bar. Figure 10.15 shows an example. The bar is subdivided into rows that correspond to the query facets. The top row of each TileBar corresponds to 'osteoporosis,' the second row to 'prevention,' and the third row to 'research.' The bar is also subdivided into columns, where each column refers to a passage within the document. Hits that overlap within the same passage are more likely to indicate a relevant document than hits that are widely dispersed throughout the document [356]. The patterns are meant to indicate whether terms from a facet occur as a main topic throughout the document, as a subtopic, or are just mentioned in passing.

The darkness of each square corresponds to the number of times the query occurs in that segment of text; the darker the square the greater the number of hits. White indicates no hits on the query term. Thus, the user can quickly see if some subset of the terms overlap in the same segment of the document. (The segments for this version of the interface are fixed blocks of 100 tokens each.)

The first document can be seen to have considerable overlap among the topics of interest towards the middle, but not at the beginning or the end (the actual end is cut off). Thus it most likely discusses topics in addition to research into osteoporosis. The second through fourth documents, which are considerably shorter, also have overlap among all terms of interest, and so are also probably of interest to the user. (The titles help to verify this.) The next three documents are all long, and from the TileBars we can tell they discuss research and prevention, but do not even touch on osteoporosis, and so probably are not of interest.

Because the TileBars interface allows the user to specify the query in terms

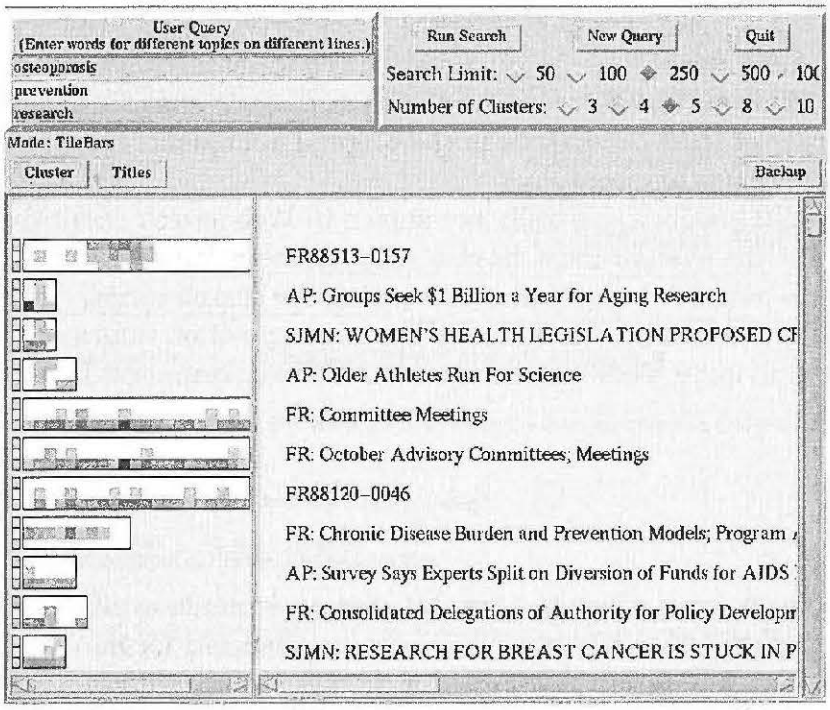


Figure 10.15 An example of the TileBars retrieval results visualization [355].

of facets, where the terms for each facet are listed on an entry line, a color can be assigned to each facet. When the user displays a document with query term hits, the user can quickly ascertain what proportion of search topics appear in a passage based only on how many different highlight colors are visible. Most systems that use highlighting use only a single color to bring attention to all of the search terms.

It would be difficult for users to specify in advance which patterns of term hits they are interested in. Instead, TileBars allows users to scan graphic representations and recognize which documents are and are not of interest. It may be the case that TileBars may be most useful for helping users discard misleadingly interesting documents, but only preliminary studies have been conducted to date. Passages can correspond to paragraphs or sections, fixed sized units of arbitrary length, or to automatically determined multiparagraph segments [355].

SeeSoft

The SeeSoft visualization [232] represents text in a manner resembling columns of newspaper text, with one 'line' of text on each horizontal line of the strip. (See Figure 10.16.) The representation is compact and aesthetically pleasing. Graphics are used to abstract away the details, providing an overview showing the amount and shape of the text. Color highlighting is used to pick out various attributes, such as where a particular word appears in the text. Details of a smaller portion of the display can be viewed via a pop-up window; the overview

shows more of the text but in less detail.

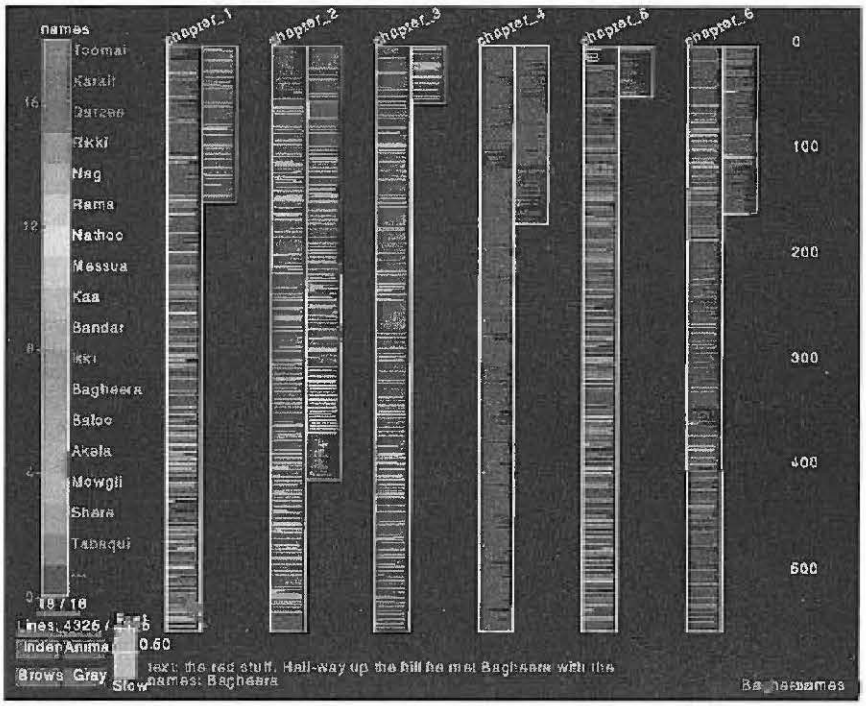


Figure 10.16 An example of the SeeSoft visualization for showing locations of characters within a text [232].

SeeSoft was originally designed for software development, in which a line of text is a meaningful unit of information. (Programmers tend to place each individual programming statement on one line of text.) Thus SeeSoft shows attributes relevant to the programming domain, such as which lines of code were modified by which programmer, and how often particular lines have been modified, and how many days have elapsed since the lines were last modified. The SeeSoft developers then experimented with applying this idea to the display of text, although this has not been integrated into an information access system. Color highlighting is used to show which characters appear where in a book of fiction, and which passages of the Bible contain references to particular people and items. Note the use of the abstraction of an entire line to stand for a single word such as a character's name (even though though this might obscure a tightly interwoven conversation between two characters).

10.6.3 Query Term Hits Between Documents

Other visualization ideas have been developed to show a different kind of information about the relationship between query terms and retrieved documents. Rather than showing how query terms appear within individual documents, as is done in KWIC interfaces and TileBars, these systems display an overview or summary of the retrieved documents according to which subset of query terms they contain. The following subsections describe variations on this idea.

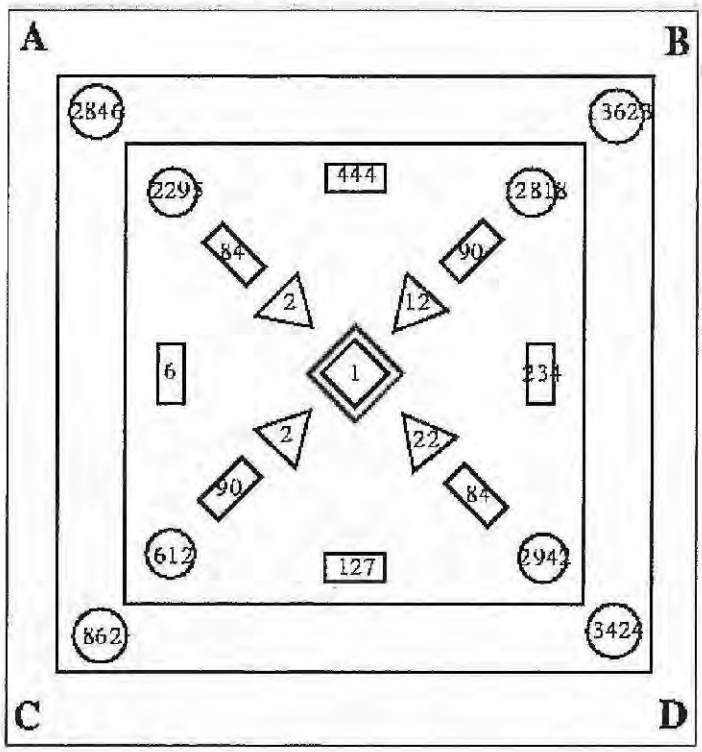


Figure 10.17 A sketch of the InfoCrystal retrieval results display [738].

InfoCrystal

The InfoCrystal shows how many documents contain each subset of query terms [738]. This relieves the user from the need to specify Boolean ANDs and ORs in their query, while still showing which combinations of terms actually appear in documents that were ordered by a statistical ranking (although beyond four terms the interface becomes difficult to understand). The InfoCrystal allows visualization of all possible relations among N user-specified 'concepts' (or Boolean keywords). The InfoCrystal displays, in a clever extension of the Venn diagram paradigm, the number of documents retrieved that have each possible subset of the N concepts. Figure 10.17 shows a sketch of what the InfoCrystal might display as the result of a query against four keywords or Boolean phrases, labeled A, B, C, and D. The diamond in the center indicates that one document was discovered that contains all four keywords. The triangle marked with '12' indicates that 12 documents were found containing attributes A, B, and D, and so on.

The InfoCrystal does not show proximity among the terms within the documents, nor their relative frequency. So a document that contains dozens of hits on 'volcano' and 'lava' and one hit on 'Mars' will be grouped with documents that contain mainly hits on 'Mars' but just one mention each of 'volcano' and 'lava.'

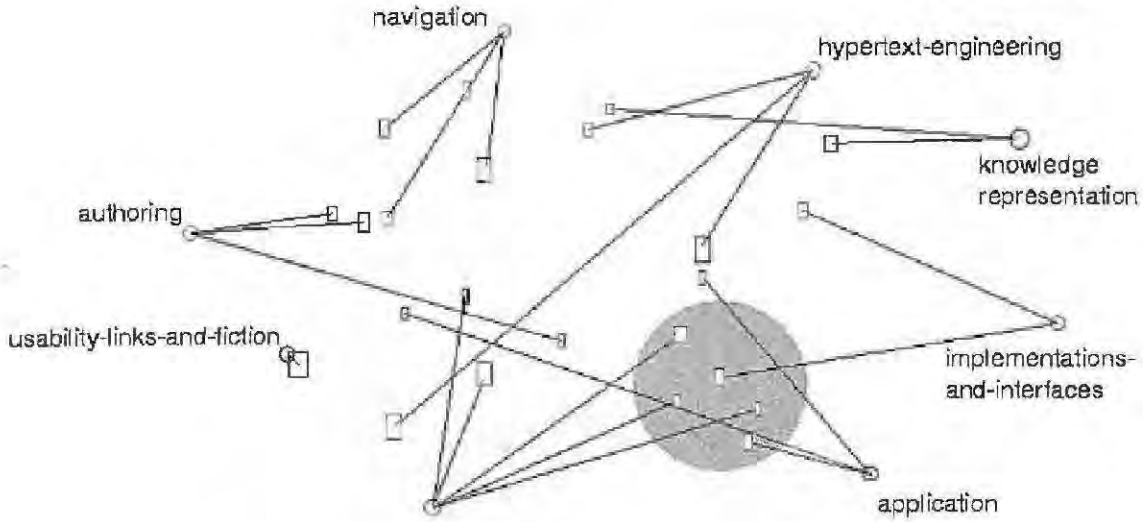


Figure 10.18 An example of the VIBE retrieval results display [452].

VIBE and Lyberworld

Graphical presentations that operate on similar principles are VIBE [452] and Lyberworld [363]. In these displays, query terms are placed in an abstract graphical space. After the search, icons are created that indicate how many documents contain each subset of query terms. The subset status of each group of documents is indicated by the placement of the icon. For example, in VIBE a set of documents that contain three out of five query terms are shown on an axis connecting these three terms, at a point midway between the representations of the three query terms in question. (See Figure 10.18.) Lyberworld presents a 3D version of this idea.

Lattices

Several researchers have employed a graphical depiction of a mathematical lattice for the purposes of query formulation, where the query consists of a set of constraints on a hierarchy of categories (actually, semantic attributes in these systems) [631, 147]. This is one solution to the problem of displaying documents in terms of multiple attributes; a document containing terms A, B, C, and D could be placed at a point in the lattice with these four categories as parents. However, if such a representation were to be applied to retrieval results instead of query formulation, the lattice layout would in most cases be too complex to allow for readability.

None of the displays discussed in this subsection have been evaluated for effectiveness at improving query specification or understanding of retrieval results, but they are intriguing ideas and perhaps are useful in conjunction with other displays.

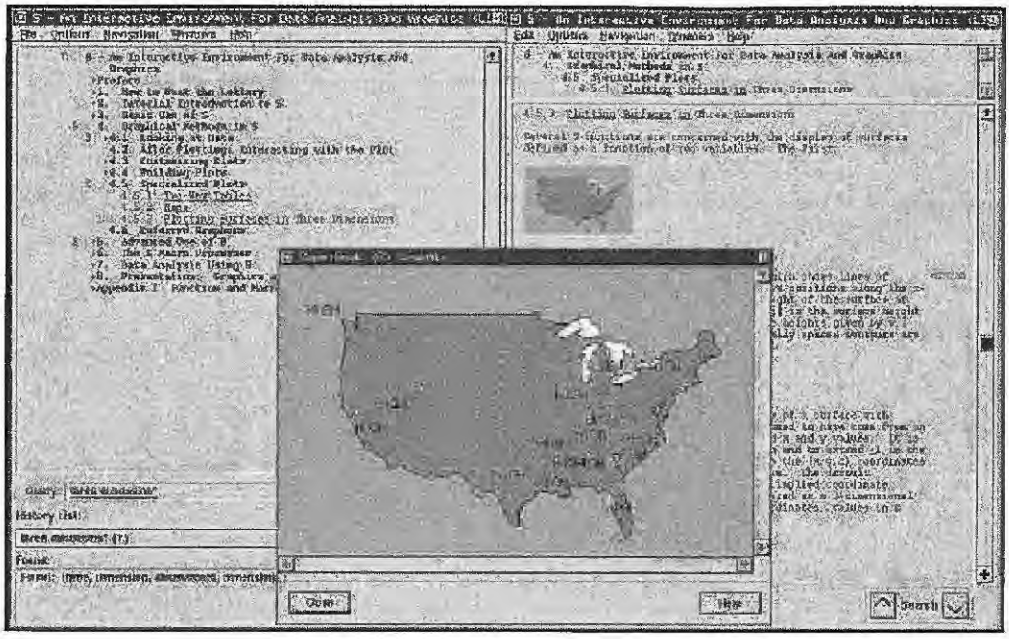


Figure 10.19 The SuperBook interface for showing retrieval results on a large manual in context [481].

10.6.4 SuperBook: Context via Table of Contents

The SuperBook system [481, 229, 230] makes use of the structure of a large document to display query term hits in context. The table of contents (TOC) for a book or manual are shown in a hierarchy on the left-hand side of the display, and full text of a page or section is shown on the right-hand side. The user can manipulate the table of contents to expand or contract the view of sections and subsections. A focus-plus-context mechanism is used to expand the viewing area of the sections currently being looked at and compress the remaining sections. When the user moves the cursor to another part of the TOC, the display changes dynamically, making the new focus larger and shrinking down the previously observed sections.

After the user specifies a query on the book, the search results are shown in the context of the table of contents hierarchy. (See Figure 10.19.) Those sections that contain search hits are made larger and the others are compressed. The query terms that appear in chapter or section names are highlighted in reverse video. When the user selects a page from the table of contents view, the page itself is displayed on the right-hand side and the query terms within the page are highlighted in reverse video.

The SuperBook designers created innovative techniques for evaluating its special features. Subjects were compared using this system against using paper documentation and against a more standard online information access system. Subjects were also compared on different kinds of carefully selected tasks: browsing topics of interest, citation searching, searching to answer questions, and searching and browsing to write summary essays. For most of the tasks

SuperBook subjects were faster and more accurate or equivalent in speed and accuracy to a standard system. When differences arose between SuperBook and the standard system, the investigators examined the logs carefully and hypothesized plausible explanations. After the initial studies, they modified SuperBook according to these hypotheses and usually saw improvements as a result [481].

The user studies on the improved system showed that users were faster and more accurate at answering questions in which some of the relevant terms were within the section titles themselves, but they were also faster and more accurate at answering questions in which the query terms fell within the full text of the document only, as compared both to a paper manual and to an interface that did not provide such contextualizing information. SuperBook was not faster than paper when the query terms did not appear in the document text or the table of contents. This and other evidence from the SuperBook studies suggests that query term highlighting is at least partially responsible for improvements seen in the system.

10.6.5 Categories for Results Set Context

In section 10.4 we saw the use of category or directory information for providing overviews of text collection content. Category metadata can also be used to place the results of a query in context.

For example, the original formulation of SuperBook allowed navigation within a highly structured document, a computer manual. The CORE project extended the main idea to a collection of over 1000 full-text chemistry articles. A study of this representation demonstrated its superiority to a standard search system on a variety of task types [228]. Since a table of contents is not available for this collection, context is provided by placing documents within a category hierarchy containing terms relevant to chemistry. Documents assigned a category are listed when that category is selected for more detailed viewing, and the categories themselves are organized into a hierarchy, thus providing a hierarchical view on the collection.

Another approach to using predefined categories to provide context for retrieval results is demonstrated by the DynaCat system [650]. The DynaCat system organizes retrieved documents according to which types of categories, selected from the large MeSH taxonomy, are known in advance to be important for a given query type. DynaCat begins with a set of query types known to be useful for a given user population and collection. One query type can encompass many different queries. For example, the query type 'Treatment-Adverse Effects' covers queries such as 'What are the complications of a mastectomy?' as well as 'What are the side-effects of aspirin?' Documents are organized according to a set of criteria associated with each query type. These criteria specify which types of categories that are acceptable to use for organizing the documents and consequently, which categories should be omitted from the display. Once categories have been assigned to the retrieved documents, a hierarchy is formed based on where the categories exist within MeSH. The algorithm selects only a subset of the category

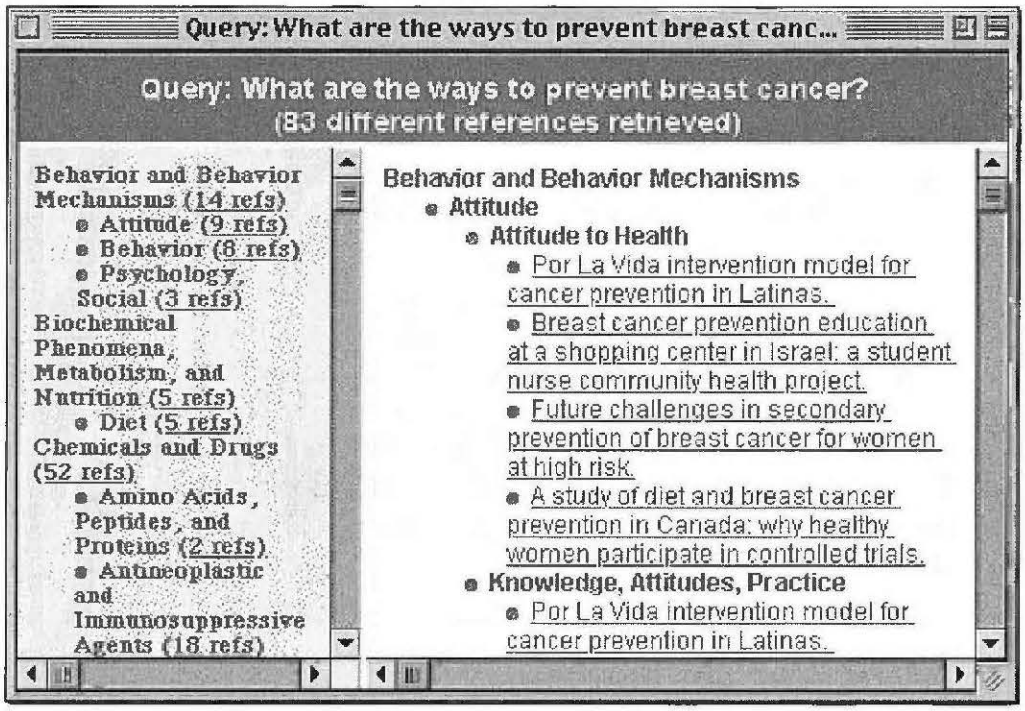


Figure 10.20 The DynaCat interface for viewing category labels that correspond to query types [650].

labels that might be assigned to the document to be used in the organization.

Figure 10.20 shows the results for a query on breast cancer prevention. The interface is tiled into three windows. The top window displays the user's query and the number of documents found. The left window shows the categories in the first two levels of the hierarchy, providing a table of contents view of the organization of search results. The right pane displays all the categories in the hierarchy and the titles of the documents that belong in those categories.

An obstacle to using category labels to organize retrieval results is the requirement of precompiled knowledge about which categories are of interest for a particular user or a particular query type. The SONIA system [692] circumvents this problem by using a combination of unsupervised and supervised methods to organize a set of documents. The unsupervised method (document clustering similar to Scatter/Gather) imposes an initial organization on a user's personal information collection or on a set of documents retrieved as the result of a query. The user can then invoke a direct manipulation interface to make adjustments to this initial clustering, causing it to align more closely with their preferences (because unsupervised methods do not usually produce an organization that corresponds to a human-derived category structure [357]). The resulting organization is then used to train a supervised text categorization algorithm which automatically classifies any new documents that are added to the collection. As the collection grows it can be periodically reorganized by rerunning the clustering algorithm and redoing the manual adjustments.