with Gonzalo Navarro

1 Introduction

Chapter ?? describes the query operations that can be performed on text databases. In this chapter we cover the main techniques to implement those query operations.

We first concentrate on searching queries composed of words and on reporting the documents where they are found. The number of occurrences of a query in each document and even its exact positions in the text may also be required. Following, we concentrate on algorithms dealing with boolean operations. We then consider sequential search algorithms and pattern matching. At the end, we consider structured text and compression techniques.

An obvious option to search for a basic query is to scan the text sequentially. Sequential or on-line text searching is the problem of finding the occurrences of a pattern in a text when the text is not preprocessed. On-line searching is appropriate when the text is small (i.e. a few megabytes), and it is the only choice if the text collection is very volatile (i.e. undergoes modifications very frequently) or the index space overhead cannot be afforded.

A second option is to build data structures over the text (called indices) to speed up the search. It is worthwhile to build and maintain an index when the text collection is large and semi-static. Semi-static collections can be updated at reasonable regular intervals (e.g. daily) but they are not deemed to support thousands of insertions of single words per second, say. This is the case of most real text databases, not only dictionaries or other slow growing literary work. For instance, it is the case for Web search engines or journal archives.

Nowadays, the most successful techniques for medium size databases (say up to 300Mb) combine on-line and indexed searching.

We cover three main indexing techniques: inverted files, suffix arrays and signature files. Keyword based search is discussed first. We emphasize inverted files, which are currently the best choice for most applications. Suffix trees and arrays are faster for phrase searches and other less common queries, but are harder to build and maintain. Finally, signature files were popular in the 80's, but nowadays inverted files outperform them. For all the structures we pay attention not only to their search cost, but also to the cost of building and updating them.

We assume that the reader is familiar with basic data structures, such as sorted arrays, binary search trees, B-trees, hash tables and tries. Since tries are heavily used we give a brief and simplified reminder here. Tries, or digital search trees, are multiway trees that store sets of strings and are able to retrieve anyone in time proportional to its length (independent of the size of the answer). A special character is added to the end of the strings to ensure that no string is a prefix of another. Every edge of the tree is labeled with a letter. To search a string in the trie, one starts at the root and scans the string character-wise, descending by the appropriate edge of the trie. This continues until a leaf is found (which represents the searched string) or the appropriate edge to follow does not exist at some point (i.e. the string is not in the set). See figure 3 for an example of a text and a trie built on its words.

Although an index must be built prior to searching it, we present both tasks in the reverse order. We think that understanding first how a data structure is used makes clear how it is organized, and therefore eases the understanding of the construction algorithm, which is usually more complex.
Throughout this chapter we make the following assumptions. We call \( n \) the size of the text database. Whenever a pattern is searched, we assume that it is of length \( m \), which is much smaller than \( n \). We call \( M \) the amount of main memory available. We assume that the modifications which a text database undergoes are additions, deletions and replacements (which are normally made by deletion plus addition) of pieces of text of size \( n' < n \).

We give experimental measures for many algorithms to give the reader a grasp of the real times involved. To do this we use a reference architecture throughout the chapter, which is representative of the power of today's computers. We use a Sun UltraSparc-1 of 167 MHz with 64 Mb of RAM, running Solaris. The code is written in C and compiled with all optimization options. For the text data, we use collections from TREC-2, specifically WSJ, DOE, FR, ZIFF and AF. These are described in more detail in chapter ??.

## 2 Inverted Files

An inverted file (or inverted index) is a word-oriented mechanism to index a text collection in order to speed up search. The inverted file structure is composed of two elements: the *vocabulary* and the *occurrences*. The vocabulary is the set of all different words in the text. For each such word a list of all the text positions where the word appears is stored. The set of all these lists is called the "occurrences" (Figure 1 shows an example). Those positions can refer to words or characters. Word positions (i.e. position \( i \) refers to the \( i \)-th word) simplify phrase and proximity queries, while character positions (i.e. the position \( i \) is the \( i \)-th character) facilitate direct access to the matching text positions.

Some authors make the distinction between inverted files and inverted lists. In an inverted file, each element of a list points to a document or file name, while inverted lists match our definition. We prefer not to make such distinction because, as we will see later, this is a matter of the *addressing granularity*, which can range from text positions to logical blocks.

![Figure 1: A sample text and an inverted index built on it. The words are converted to lower-case and some are not indexed. The occurrences point to character positions in the text.](image)

The space required for the vocabulary is rather small. According to Heaps' law (see chapter ??) the vocabulary grows as \( O(n^\beta) \), where \( \beta \) is a constant between 0 and 1 dependent on the text, being between 0.4 and 0.6 in practice. For instance, for 1 Gb of the TREC collection the vocabulary has a size of only 5 Mb. This may be further reduced by stemming and other normalization techniques as described in chapter ??.
The occurrences demand much more space. Since each word appearing in the text is referenced once in that structure, the extra space is \( O(n) \). Even omitting stop-words (which is the default practice when words are indexed), in practice the space overhead of the occurrences is between 30\% and 40\% of the text size.

To reduce space requirements, a technique called block addressing is used. The text is divided in blocks, and the occurrences point to the blocks where the word appears (instead of the exact positions). The classical indices which point to the exact occurrences are called "full inverted indices". By using block addressing not only the pointers can be smaller because there are less blocks than positions, but also all the occurrences of a word inside a single block are collapsed to one reference (see Figure 2). Indices of only 5\% overhead over the text size are obtained with this technique. The price to pay is that, if the exact occurrence positions are required (for instance for a proximity query) then an on-line search over the qualifying blocks has to be performed. For instance, block addressing indices with 256 blocks stops working well with texts of 200 Mb of size.

Table 1 presents the projected space taken by inverted indices for texts of different sizes. The full inversion stands for inverting all the non stop words and storing their exact positions, using four bytes per pointer. The document addressing index assumes that we point to documents which are of size 10 Kb (and the necessary number of bytes per pointer, i.e. one, two and three bytes). The block addressing index assumes that we use 256 or 64K blocks (one or two bytes per pointer) independently of the text size. The space taken by the pointers can be significantly reduced by using compression. We assume that 45\% of all the words are stop words, and that there is one non-stop word each 11.5 characters. Our estimation for the vocabulary is based on the Heaps' Law with parameters \( V = 30n^{0.8} \). All these decision were taking according to our experience and experimentally validated.

The blocks can be of fixed size (imposing a logical block structure over the text database) or they can be defined using the natural division of the text collection into files, documents, Web pages or others. The division into blocks of fixed size improves efficiency at retrieval time, i.e. the more variance in the block sizes, the more amount of text sequentially traversed on average. This is because larger blocks match queries more frequently and are more expensive to traverse.

On the other hand, the division using natural cuts may eliminate the need of the on-line traversal. For example, if one block per retrieval unit is used and the exact match positions...
Table 1: Percentages of the text taken by an inverted file with and without using stopwords for different file sizes and granularities.

are not required, there is no need to traverse the text for single-word queries, since it is enough to know which retrieval units to report. On the other hand, if many retrieval units are packed into a single block, the block has to be traversed to determine which units to retrieve.

It is important to notice that in order to use block addressing, the text must be readily available at search time. This is not the case of remote text (as in Web search engines), or if the text is in a CD-ROM that has to be mounted, for instance.

2.1 Searching

The search algorithm on an inverted index follows three general steps (some may be absent for specific queries):

**Vocabulary search:** the words and patterns present in the query are isolated and searched in the vocabulary. Notice that phrases and proximity queries are split in single words.

**Retrieval of occurrences:** the lists of the occurrences of all the words found are retrieved.

**Manipulation of occurrences:** the occurrences are processed to solve phrases, proximity or boolean operations. If block addressing is used it may be necessary to directly search the text to find the information missing from the occurrences (e.g. exact word positions to form phrases).

Hence, searching on an inverted index always starts in the vocabulary. Because of this it is a good idea to have it in a separate file. It is possible that this file fits in main memory even for large text collections.

Single-word queries can be searched using any suitable data structure to speed up the search, such as hashing, tries or B-trees. The first two give $O(m)$ search cost (independent of the text size). However, simply storing the words in lexicographical order is cheaper in space and very competitive in performance, since the word can be binary searched at $O(\log n)$ cost. Prefix and range queries
can also be solved with binary search, tries, or B-trees, but not with hashing. If the query is formed by single words, then the process ends by delivering the list of occurrences (we may need to make the union of many lists if the pattern matches many words).

Context queries are more difficult to solve with inverted indices. Each element must be searched separately and a list (in increasing positional order) generated for each one. Then, the lists of all elements are traversed in synchronization to find places where all the words appear in sequence (for a phrase) or they appear close enough (for proximity). If one list is much shorter than the others, it may be better to binary search its elements into the longer lists instead of performing a linear merge. It is possible to prove using the Zipf’s Law that this is normally the case. This is important because the most time-demanding operation on inverted indices is the merging or intersection of the lists of occurrences.

If the index stores character positions the phrase query cannot allow disregarding the separators, and the proximity has to be defined in terms of character distance.

Finally, notice that if block addressing is used it is necessary to traverse the blocks for these queries, since the position information is needed. It is then better to intersect the lists to obtain the blocks which contain all the searched words and then sequentially search the context query in those blocks as explained in Section 5. Some care has to be exercised at block boundaries, since they can split a match. This part of the search, if present, is also quite time-consuming.

Using Heaps' and the generalized Zipf’s laws, it has been demonstrated that the cost to solve queries is sublinear in the text size, even for complex queries involving list merging. The time complexity is $O(n^\alpha)$, where $\alpha$ depends on the query and is close to 0.4..0.8 for queries with reasonable selectivity.

Even if block addressing is used and the blocks have to be traversed, it is possible to select the block size as an increasing function of $n$, so that not only the space requirement keeps sublinear but also the amount of text traversed in all useful queries is also sublinear.

Practical figures show, for instance, that both the space requirement and the amount of text traversed can be close to $O(n^{0.85})$. Hence, inverted indices allow having sublinear search time at sublinear space requirements. This is not possible on the other indices.

Search times on our reference machine for a full inverted index built on 250 Mb of text gives the following times: searching a simple word took 0.08 seconds, while searching a phrase took 0.25 to 0.35 seconds (from 2 to 5 words).

2.2 Construction

Building and maintaining an inverted index is a relatively low-cost task. In principle, an inverted index on a text of $n$ characters can be built in $O(n)$ time. All the vocabulary known up to now is kept in a trie data structure, storing for each word a list of its occurrences (text positions). Each word of the text is read and searched in the trie. If it is not found, it is added to the trie with an empty list of occurrences. Once it is in the trie, the new position is added to the end of its list of occurrences. Figure 3 illustrates this process.

Once the text is exhausted, the trie is written down to disk together with the list of occurrences. Since in the trie $O(1)$ operations are performed per text character, and the positions can be inserted at the end of the lists of occurrences in $O(1)$ time, the overall process is $O(n)$ worst-case time.
This is a text. A text has many words. Words are made from letters.

Vocabulary trie

Figure 3: Building an inverted index for the sample text.

However, the above algorithm is not practical for large texts where the index does not fit in main memory. A paging mechanism will severely degrade the performance of the algorithm. We describe an algorithm which is faster in practice.

The already described algorithm is used up to when the main memory is exhausted (if the trie takes up too much space it can be replaced by a hash table or another structure). When no more memory is available, the partial index $I_i$ obtained up to now is written to disk and erased from main memory before continuing with the rest of the text.

At the end, a number of partial indices $I_i$ exist on disk. Those indices are then merged in a hierarchical fashion. Indices $I_1$ and $I_2$ are merged to obtain the index $I_{1..2}$; $I_3$ and $I_4$ produce $I_{3..4}$; and so on. The resulting partial indices are approximately twice the size now. When all the indices of this level have been merged in that way, the merging proceeds at the next level, joining the index $I_{1..2}$ with the index $I_{3..4}$ to form $I_{1..4}$. This is continued until there is just one index comprising the whole text, as illustrated in Figure 4.

Figure 4: Merging the partial indices in a binary fashion. Rectangles represent partial indices, while rounded rectangles represent merging operations. The numbers inside the merging operations show a possible merging order.
It is good practice to split the index in two files. In a first one the lists of occurrences are stored contiguously. In this scheme, this file is typically called "posting file". In the other file, the vocabulary is sequentially stored in lexicographical order, storing for each word a pointer to its list in the occurrence file. This allows keeping the vocabulary in memory at search time in many cases. The number of occurrences of a word can be immediately known from the vocabulary with little or no space overhead.

Merging two indices consists of merging the sorted vocabularies, and whenever the same word appears in both indices, merging both lists of occurrences. By construction, the occurrences of the smaller-numbered index are before those of the larger-numbered index, and therefore the lists are just concatenated. This is a very fast process in practice, and its complexity is $O(n_1 + n_2)$, where $n_1$ and $n_2$ are the sizes of the indices.

The total time to generate the partial indices is $O(n)$ as before. The number of partial indices is $O(n/M)$. Each level of merging performs a linear process over the whole index (no matter how is it split into partial indices at this level) and thus its cost is $O(n)$. To merge the $O(n/M)$ partial indices, $\log_2(n/M)$ merging levels are necessary, and therefore the cost of this algorithm is $O(n \log(n/M))$.

More than two indices can be merged at once. Although this does not change the complexity, it improves efficiency since fewer merging levels exist. On the other hand the memory buffers for each partial index to merge will be smaller and hence more disk seeks will be performed. In practice it is a good idea to merge even 20 partial indices at once.

Real times to build inverted indices on the reference machine are between 4-8 Mb/min for collections of up to 1 Gb (the slowdown factor as the text grows is barely noticeable). Of this time, 20-30% is spent on merging the partial indices.

To reduce build-time space requirements, it is possible to perform the merging in-place. That is, when two or more indices are merged, write the result in the same disk blocks of the original indices instead of on a new file. It is also a good idea to perform the hierarchical merging as soon as the files are generated (e.g. collapse $I_1$ and $I_2$ into $I_{1,2}$ as soon as $I_2$ is produced). This also reduces space requirements because the vocabularies are merged and redundant words are eliminated (there is no redundancy in the occurrences). The vocabulary can be a significative part of the smaller partial indices, since they represent a small text.

This algorithm changes very little if block addressing is used. Index maintenance is also cheap. Assume that a new text of size $n'$ is added to the database. The inverted index for the new text is built and then both indices are merged as done for partial indices. This takes $O(n + n' \log(n'/M))$. Deleting text can be done by an $O(n)$ pass over the index eliminating the occurrences that point inside eliminated text areas (and eliminating words if their lists of occurrences disappear in this process).

### 3 Other Indices for Text

#### 3.1 Suffix Trees and Suffix Arrays

Inverted indices assume that the text can be seen as a sequence of words. This restricts somewhat the kinds of queries that can be answered. Other queries such as phrases are expensive to solve. Moreover, the concept of word does not exist in some applications such as genetic databases.
In this section we present the suffix arrays. Suffix arrays are a space-efficient implementation of suffix trees. This type of index allows answering efficiently more complex queries. Its main drawbacks are its costly construction process, that the text must be readily available at query time, and that the results are not delivered in text position order. This structure can be used to index only words (without stop-words) as the inverted index as well as to index any text character. This makes it suitable to a wider spectrum of applications, such as genetic databases. However, for word based applications, inverted files perform better unless complex queries are an important issue.

This index sees the text as one long string. Each position in the text is considered as a text suffix (i.e. a string that goes from that text position to the end of the text). It is not difficult to see that two suffixes starting at different position are lexicographically different (assume that a character smaller than all the rest is placed at the end of the text). Each suffix is thus uniquely identified by its position.

Not all text positions need to be indexed. Index points are selected from the text, which point to the beginning of the text positions which will be retrievable. For instance, it is possible to index only word beginnings to have a functionality similar to inverted indices. Those elements which are not index points are not retrievable (as in an inverted index it is not possible to retrieve the middle of a word). Figure 5 exemplifies.

![Figure 5: The sample text with the index points of interest marked. Below, the suffixes corresponding to those index points.](image)

### 3.1.1 Structure

In essence, a suffix tree is a trie data structure built over all the suffixes of the text. At the leaf nodes the pointers to the suffixes are stored. To improve space utilization, this trie is compacted into a Patricia tree. This involves compressing unary paths, i.e. paths where each node has just one child. At the nodes which root a compressed path, an indication of how many characters to skip is stored. Once unary paths are not present the tree has $O(n)$ nodes instead of the worst-case $O(n^2)$ of the trie (see Figure 6).

The problem of this structure is its space. Depending on the implementation, each node of the trie takes 12 to 24 bytes, and therefore even if only word beginnings are indexed, an overhead of 120% to 240% over the text size is produced.
Suffix arrays provide essentially the same functionality as suffix trees at much less space requirements. If the leaves of the suffix tree are traversed in left-to-right order, all the suffixes of the text are retrieved in lexicographical order. A suffix array is simply an array containing all the pointers to the text suffixes listed in lexicographical order, as shown in Figure 7. Since they store one pointer per indexed suffix, the space requirements are almost the same as those for inverted indices (disregarding compression techniques), i.e. close to 40% overhead over the text size.

Suffix arrays are designed to be binary searched. If the suffix array is large (the usual case), this binary search can perform poorly because of the number of random disk accesses. To remedy this situation, the use of supra-indices over the suffix array has been proposed. The simplest supra-index is no more than a sampling of one out of $b$ suffix array entries, where for each sample the first suffix characters are stored in the supra-index. This supra-index is then used as a first step of the search to reduce external accesses. Figure 8 shows an example.

This supra-index does not need in fact to take samples at fixed intervals, nor to take samples of the same length. For word-indexing suffix arrays it has been suggested that a new sample could be taken each time the first word of the suffix changes, and to store the word instead of the characters. This is exactly the same as having a vocabulary of the text plus pointers to the array. In fact, the only important difference between this structure and an inverted index is that the occurrences of each word in an inverted index are sorted by text position, while in a suffix array are sorted lexicographically by the text following the word. Figure 9 illustrates this relationship.

The extra space requirements of supra-indices are modest. In particular, it is clear that the space requirements of the suffix array with a vocabulary supra-index are exactly the same as for...
This is a text. A text has many words. Words are made from letters.

Figure 8: A supra-index over our suffix array. One out of three entries are sampled, keeping their first four characters. The pointers (arrows) are in fact unnecessary.

Figure 9: Relationship between our inverted list and suffix array with vocabulary supra-index.

### 3.1.2 Searching

If a suffix tree on the text can be afforded, many basic patterns such as words, prefixes and phrases can be searched in $O(m)$ time by a simple trie search. However, suffix trees are not practical for large texts, as explained. Suffix arrays, on the other hand, can perform the same search operations in $O(\log n)$ time by doing a binary search instead of trie search.

This is achieved as follows: the search pattern originates two "limiting patterns" $P_1$ and $P_2$, so that we want any suffix $S$ such that $P_1 \leq S < P_2$. We binary search both limiting patterns in the suffix array. Then, all the elements lying between both positions point to exactly those suffixes that start like the original pattern (i.e. to the pattern positions in the text). For instance, in a purely alphabetical text, in order to find the word "hello" we search for "hello" and "hellp".

All those queries retrieve a subtree of the suffix tree or an interval of the suffix array. The results have to be later collected, which may imply to sort them in ascending text order. This is a complication of suffix trees or arrays with respect to inverted indices.

Simple phrase searching is a good case for these indices. A simple phrase of words can be searched as if it was a simple pattern. This is because the suffix tree/array sorts with respect to the complete suffixes and not only their first word. A proximity search, on the other hand, has to be solved element-wise. The matches for each element must be collected and sorted and then they have to be intersected as for inverted files.
The binary search performed on suffix arrays, unfortunately, is done on disk, where the accesses to (random) text positions orces a seek operation which spawns the disk tracks containing the text. Since a random seek is $O(n)$ in size, this makes the search cost $O(n \log n)$ time. Supra-indices are used as a first step in any binary search operation to alleviate this problem. To avoid performing $O(\log n)$ random access to the text on disk (and to the suffix array on disk), the search starts in the supra-index, which usually fits in main memory (text samples included). After that search is completed, the suffix array block which is between the two selected samples is brought into memory and the binary search is completed (performing random accesses to the text). This reduces disk search times to close to 25% of the original time. Modified binary search techniques that sacrifice the exact partition in the middle of the array taking into account the current disk head position allow a further reduction from 40% to 60%.

Search times in a 250 Mb text in our reference machine are close to 1 second for a simple word or phrase, while the part corresponding to the accesses to the text sums up 0.6 seconds. The use of supra-indices should put the total time close to 0.3 seconds. Notice that the times, although high for simple words, do not degrade for long phrases as in inverted indices.

### 3.1.3 Construction in Main Memory

A suffix tree for a text of $n$ characters can be built in $O(n)$ time. The algorithm, however, performs poorly if the suffix tree does not fit in main memory, which is especially stringent because of the large space requirements of the suffix trees. We do not cover the linear algorithm here because it is quite complex and only of theoretical interest.

We concentrate on direct suffix array construction. Since the suffix array is no more than the set of pointers lexicographically sorted, the pointers are collected in ascending text order and then just sorted by the text they point to. Notice that in order to compare two suffix array entries the corresponding text positions must be accessed. Those accesses are basically random. Hence, both the suffix array and the text must be in main memory. This algorithm costs $O(n \log n)$ string comparisons.

An algorithm to build the suffix array in $O(n \log n)$ character comparisons follows. All the suffixes are bucket-sorted in $O(n)$ time according to the first letter only. Then, each bucket is bucket-sorted again, now according to their first two letters. At iteration $i$, the suffixes begin already sorted by their $2^{i-1}$ first letters and end up sorted by their first $2^i$ letters. As at each iteration the total cost of all the bucket sorts is $O(n)$, the total time is $O(n \log n)$, and the average is $O(n \log \log n)$ (since $O(\log n)$ comparisons are necessary on average to distinguish two suffixes of a text). This algorithm accesses the text only in the first stage (bucket sort for the first letter).

In order to sort the strings in the $i$-th iteration, notice that since all suffixes are sorted by their first $2^{i-1}$ letters, to sort the text positions $T_{a...}$ and $T_{b...}$ in the suffix array (assuming that they are in the same bucket, i.e. they share their first $2^{i-1}$ letters), it is enough to determine the relative order between text positions $T_{a+2^{i-1}}$ and $T_{b+2^{i-1}}$ in the current stage of the search. This can be done in constant time by storing the reverse permutation. We do not enter here into further detail.
3.1.4 Construction of Suffix Arrays for Large Texts

There is still the problem that large text databases will not fit in main memory. It could be possible to apply an external memory sorting algorithm. However, each comparison involves accessing the text at random positions in disk. This will severely degrade the performance of the sorting process.

We explain an algorithm especially designed for large texts. Split the text in 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 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 in 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 into the deleted areas.

As it 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 5 to 10 times slower than the construction of inverted indices.
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 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.

3.2.1 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-or’ing 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 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 most $\ell$ of bits set. A good model assumes that $\ell$ bits are randomly set in the mask (with possible repetition). Let $\alpha = \ell/B$. Since each of the $b$ words sets $\ell$ bits at random, the probability that a given bit of the mask is set in a word signature is $1 - (1 - 1/B)^{\ell b} \approx 1 - e^{-\alpha b}$. Hence, the probability that the $\ell$ random bits set in the query are also set in the mask of the text block is

$$ (1 - e^{-\alpha b})^\alpha B $$
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. On the other hand, 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.

### 3.2.2 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 $(W \& B_i = W)$, then 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 on-line 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 only found 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.
3.2.3 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-\texttt{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 \(\ell\) bits which are set in the query have to be traversed.

4 Boolean Queries

We cover now 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 ??, where the concept of \textit{query syntax tree} is defined.

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

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

As all operations need to pair the same document in their both operands, it is good practice to keep the sets sorted, so that operations like intersection, union, etc. can proceed sequentially on both lists and generate also 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 \textit{full or lazy} form. In the full evaluation, 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 for a long time. Hybrid schemes are possible, for instance 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 12 illustrates.

Figure 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.

The complexity to solve these types of queries, apart from the cost to obtain 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.

5 Sequential Searching

We cover now 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).

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 exist. We sketch the main algorithms, and leave aside a lot of 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 it is not practical. We briefly cover also multipattern algorithms (that search many patterns at once), since a query may have many patterns and it may be more efficient to retrieve all them at once. At the end 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 \( \sigma \), whose first character is at position 1. The average-case analysis assumes random text and patterns.

5.1 Brute Force

The brute-force (BF) algorithm is the simplest possible one. It consists in merely trying all possible pattern positions in the text. For each such position, it verifies whether the pattern matches at
that position. See Figure 13.

```
\texttt{abracadabra}
```

Figure 13: Brute-force search algorithm for the pattern "abracadabra". Squared areas show the comparisons performed.

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 \textit{window} of length $m$ which is slid over the text. It is \textit{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 \textit{shifted} forward. The algorithms mainly differ in the way they check and shift the window.

\section{5.2 \textit{Knuth-Morris-Pratt}}

The \textit{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 on 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 \textit{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 \textit{next}. The \textit{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 $j - \text{next}[j] + 1$ 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", 5 positions can be safely skipped since $\text{next}[7] = 1$.

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 \textit{next}, but the pointer position in the text does not change. Since at each text
comparison the window or the pointer advance in at least one position, the algorithm performs at most $2n$ comparisons (and at least $n$). Figure 14 shows an example.

$\text{next} = 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 4$

Figure 14: KMP algorithm searching "abracadabra". On top, an illustration of the $\text{next}$ 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". Below, a search example. Grayed areas show the prefix information reused.

The Aho-Corasick algorithm can be regarded as an extension of KMP to match a set of patterns. The patterns are arranged into a trie-like data structure. Each trie node represents having matched a prefix of some pattern(s). The $\text{next}$ function is replaced by a more general set of $\text{failure}$ 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 15 illustrates.

Figure 15: Aho-Corasick trie example for the set "hello", "elbow" and "eleven" showing one failure transition.

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 $O(n)$. Much as KMP, it makes at most $2n$ inspections.

### 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 $\text{suffix}$ 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 $\text{next}$ 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 "abracadabra". 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 shifted by 7. See Figure 16.

Figure 16: BM algorithm searching "abracadabra". Squared areas show the comparisons performed. Grayed areas have been already compared (but the algorithm compares them again). The dashed box shows the match heuristic, which was not chosen.

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 to 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 anymore important 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.

### 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 using cleverly this fact, the number of operations that an algorithm performs can be cut down by a factor of at most $w$. Since in current architectures $w$ is 32 or 64, the speedup is very significative in practice.

The Shift-Or algorithm uses bit-parallelism to simulate the operation of a nondeterministic automaton that searches the pattern in the text (see Figure 17). As this automaton is simulated in time $O(mn)$, Shift-Or algorithm achieves $O(mn/w)$ worst-case time (optimal speedup).

The algorithm builds first 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 $p_i = c$ (see Figure 17). The state of the
search is kept in a machine word $D = d_m \ldots d_1$, where $d_i$ is zero whenever the state numbered $i$ in Figure 17 is active. Therefore, a match is reported whenever $d_m$ is zero. In the following, we use "\texttt{\&}" to denote the bitwise-or and "\texttt{|}" 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 "\texttt{\textless \textless}" means shifting all the bits in $D$ 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 nondeterministic 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 was later enhanced to support 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).

### 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 nondeterministic version of this automaton has a very regular structure and is shown in Figure 18.

![Figure 18: A nondeterministic suffix automaton. Dashed lines represent \(\varepsilon\)-transitions (i.e. they occur without consuming any input). I is the initial state of the automaton.](image)

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 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 reaching 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 19.

\[
\begin{array}{cccccc}
a_1 & b_1 & r_1 & a_1 & c_1 & a_1 b_1 r_1 a_1 c_1 a_1 d_1 a_1 b_1 r_1 a_1 \\
X & X & & & & \\
X & X & X & X
\end{array}
\]

Figure 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.

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 BM in a wide range of cases. This algorithm simulates the nondeterministic 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.

5.6 Practical Comparison

Figure 20 shows a practical comparison among string matching algorithms run on our reference machine. The values are correct within 5% of accuracy with 95% confidence. 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 times. 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 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 shows 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 apart from 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 uses better the information on the pattern.

![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.](image)

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.

### 5.7 Phrases and Proximity

If a phrase 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.

6 Pattern Matching

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

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 number of errors allowed $k$, 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.

6.1.1 Dynamic Programming

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

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

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 is $O(m)$ time. Figure 21 exemplifies this algorithm applied to compute edit distance.

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).
Figure 21: The dynamic programming algorithm search "survey" in the text "surgery" with two errors. Bold entries indicate matching positions.

6.1.2 Automaton

It is interesting to notice that the problem can be reduced to a nondeterministic finite automaton (NFA). Consider the NFA for $k = 2$ errors shown in Figure 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 $\varepsilon$-transitions). The automaton accepts a text position as the end of a match 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 vertical vector of the dynamic programming algorithm. Figure 22 illustrates (compare to Figure 21).

Figure 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".

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.

### 6.1.3 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.

### 6.1.4 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, $O(kn \log (m)/m)$ 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 in $k+1$ pieces, any approximate occurrence must contain at least one of the pieces with no errors, since $k$ errors cannot alter 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).

### 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 nondeterministic 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. $O(m^{2m})$. See Figure 23.

![Figure 23: The nondeterministic (a) and deterministic (b) automata for the regular expression $b^* (b \mid b^*a)$.](image)

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 nondeterministic automaton is simulated instead. One bit per automaton
state is used to represent whether the state is active or not. All the transitions move forward except for \( \varepsilon \)-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 \( \varepsilon \)-transitions. A function \( E \) from bit masks to bit masks is precomputed so that all the corresponding bits are moved according to the \( \varepsilon \)-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 \( E(B_1\ldots B_j) = E(B_1)\ldots E(B_j) \), where \( B_i \) are the submasks. Hence, the scheme performs \( |m/8| \) or \( |m/16| \) operations per text character and needs \( |m/8|2^8|m/w| \) or \( |m/16|2^{16}|m/w| \) 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.

### 6.3 Pattern Matching using Indices

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

#### 6.3.1 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. on-line) 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.

Search times for simple words allowing errors on 250 Mb of text took on 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.
6.3.2 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 collection and sorting of 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 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 on-line 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 this 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 on-line algorithm.

Suffix trees are able to perform other complex searches that we have not considered in our query language (see chapter ??). 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 tries at $O(n)$ 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 as 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.

7 Structural Queries

The algorithms to search on structured text (see chapter ??) 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 of HTML text but not the case of C code where the marks are implicit and are inherent to the language. They then rely 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 it easier to obtain the structure information. On the other hand, some queries such as direct ancestorship 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 4. However, the operations tend to be more complex, and it is not always possible to find an evaluation algorithm which is 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 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 the last 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 list of numbers can be compressed has been covered on chapter ??.

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 code for 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 the 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 ??.

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 the 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 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, the Shift-Or algorithm considers that a new text symbol has been read, whose bit mask is that of the leaf (see Figure 24). This algorithm is surprisingly simple and efficient.

Figure 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 solve 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 5.6). However, if the I/O times are considered, compressed searching is faster than all the on-line algorithms. For complex queries,
this scheme is by far unbeaten.

On the reference machine, the CPU times are 14 Mb/sec without the BM speed-up and 18 Mb/sec with the simple search. Agrep, on the other hand, runs at 15 Mb/sec on simple searches and at 1-4 Mb/sec for complex ones if the I/O times are considered.

8.2 Compressed Indexing

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. Those differences can be represented using less space by using techniques that favor small numbers (see Chapter ??). 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 transferring 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.

Independently of the index, the text can also be compressed. The text will be decompressed only to display it, or to traverse it in case of block addressing. Notice in particular that the on-line search technique described for compressed text in Section 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 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 suffix arrays, which impose much smaller performance penalties.

Suffix Arrays, on the other hand, are very hard to compress further. This is because they represent an almost perfectly random permutation of the pointers to the text.

On the other hand, 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 less text blocks are needed. Searching is faster because a large part of the search time is spent in disk seeks 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 8.1 is used. However, the Huffman code on words is replaced by a Hu-Tucker coding. The Hu-Tucker code respects the lexicographical relationships among 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 alternatives to compress signature files. All of them are based in the fact that only a few bits are set in all the 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 reducing 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.

9 Trends and Research Issues

In this chapter we cover extensively the current techniques to deal with text retrieval. We first covered indices and then on-line searching. We then reviewed set manipulation, complex pattern matching and finally considered compression techniques. Figure 25 summarizes the trade-off of the space needed for the index versus the time to search one single word.

Probably the most adequate indexing technique in practice is the inverted file. As we carefully showed across the chapter, many hidden details in other structures make them harder to use and less efficient in practice, as well as less flexible to deal with new types of queries. Those 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 on 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 anymore. On the other hand, the speed of the processors and the relative slowness of external devices change 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 mentioned changes 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 also trade space for processor time.
10 Bibliographic Discussion

A detailed explanation of a full inverted index and its construction and querying process can be found in [ANZ97]. That work includes also an analysis of the algorithms on inverted lists using the distribution of natural language. The in-place construction is described in [MB95].

The idea of block addressing inverted indices was first presented in a public system called Glimpse [?], which also firstly exposed the idea of performing complex pattern matching using the vocabulary of the inverted index. Block addressing indices are analyzed in [BYN97], 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 [LST96].

Suffix arrays were presented in [MM90] together with the algorithm to build them in $O(n \log n)$ character comparisons. They were independently discovered by [Gon87] under the name of “PAT arrays”. The algorithm to build large suffix arrays is presented in [GBYS92]. The use of supra-indices over suffix array is proposed in [BYBZ94], while the modified binary search techniques to reduce seek time are presented in [BNBY+95]. The linear-time construction of suffix trees is described in [Ukk95].

The material on signature files is based on [FC87]. The different alternatives to store the signature file are exposed in [FC88].

The original references for the sequential search algorithms are: KMP [KMP77], BM [BM77], BMH [Hor80], BMS [Sun90], Shift-Or [BYG92], BDM [CCG+94] and BNDM [NR98]. The multi-pattern versions are found in [AC75, CW79], and MultiBDM in [CR94]. Many enhancements of bit-parallelism to support extended patterns and allow errors are presented in [WM92b]. Many ideas of that paper were implemented in a widely distributed software for on-line searching called Agrep [WM92a].

The reader interested in more details about sequential searching algorithms may look for the original references or in good books on algorithms such as [GBY91, CR94].

One source for the classical solution to approximate string matching is [Sel80]. An $O(kn)$ worst-case algorithm is described in [LV88]. The use of a DFA is proposed in [Ukk85]. The bit-parallel approach to this problem started in [WM92b], although currently the fastest bit-parallel algorithms are [Mye98] and [BYN98]. Among all the filtering algorithms, the fastest one in practice is based on an idea presented in [WM92b], later enhanced in [BYP96] and finally implemented in [BYN98].

A good source to learn about regular expressions and building a DFA is [HU79]. The bit-parallel implementation of the NFA is explained in [WM92b]. Regular expression searching on suffix trees is described in [BYG96], while searching allowing errors is presented in [Ukk93].

The Huffman coding was first presented in [Huf52], while the word-oriented alternative is proposed in [Mof89]. Sequential searching on text compressed using that technique is described in [?]. Compression used in combination with inverted files is described in [ZM95], with suffix trees in [KU94], with suffix arrays in [MNZ97], and with signature files in [FC87, FC88]. A good general reference on compression is [TBW90].

References


34


Figure 25: Trade-off of index space versus word searching time.