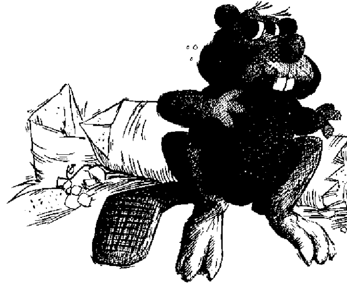
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



Unstructured Data Bases

Gaston H. Gonnet

CS-82-09

March, 1982

Unstructured Data Bases

by

Gaston H. Gonnet
Department of Computer Science
University of Waterloo

Abstract.

We present several algorithms to search data bases that consist of text. The algorithms apply mostly to very large data bases that are difficult to structure.

We describe algorithms which search the original database without transformation and algorithms requiring pre-processing.

The problem of misspellings, ambiguous spellings, simple errors, endings, etc. is nicely treated using signature functions.

Key words. Unstructured data bases; information retrieval; pattern matching; hashing; signatures; text searching; string searching; bibliographic search; membership testing; inverted files; full text search.

1. Introduction.

There are several examples of large collections of related data which cannot, under the current data base theory, be treated as a data base. This class is characterised by having little or no structure; units of data (records) consisting mostly of text; additions, but otherwise virtually no updates and unpredictable queries. A few examples of such collections of data clarify this immediately:

(a) Law,

(b) Journalism,

(c) Technical and scientific information,

(d) Consumers' information,

(e) Intelligence information.

In all these examples there is some structure that may provide a rough classification of the information. This classification usually is not enough to single out records and it is often of little value for the type of queries we want to perform.

Examples of queries on the above data bases are

(a) Recent cases of married women poisoned by their lovers

(b) All the articles, of any type, that mention "Mark Springlove" who will be named secretary of state

(c) Papers that mention "Unstructured data bases" more than twice

(d) Information about sand disposal in washing machines

(c) Country "X" claims to have an atomic bomb and we want to trace all radioactive material shipments that may have been diverted.

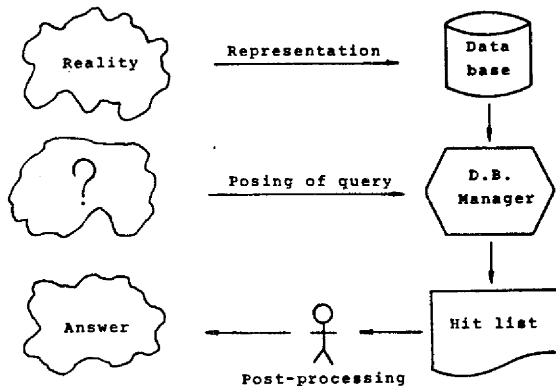
Although most of these queries could be properly answered with the right indexing it is important to note that we cannot hope for this ideal situation since

(1) the required indexing may be unpredictable at the time the record is stored (owing to lack of criteria or changing criteria). E.g. Shipments of radioactive material were a nuisance and did not have any strategic importance 50 years ago.

(2) The indexing may require professional manpower which may not be available or affordable given the rate of growth or starting size of the data base. E.g. Information institutes report that there are about 1.3 million scientific papers published in the world every year.

(3) Any error in the initial indexing may cause an important record to be ignored.

In principle the answer to this type of queries is a list of records, the *hit list*. Typically this list is not processed further inside the system, but it is post-processed by humans to determine its applicability, interpretation etc. Schematically:



The query Scheme

This process is certainly different from a transaction oriented process, which in some sense is much more "deterministic".

In all the examples presented we already have a method for representation of the information which is suitable for computing: the information is written. The weak link in this system is posing the query. This is a major problem in Information retrieval theory.

As in statistics, the list of hits is subject to two types of errors:

(I) *noise*; records that are listed but are not related to our query.

(II) *misses*; records that were relevant to our query but were not retrieved.

In information's retrieval terminology, the Cleverdon measures ***precision*** and *recall* are the complements of the type I and type II errors respectively. Invariably, trying to lower one type of error will increase the other. In real terms, however, we are much more interested in a low type II error since the human post-processing could easily get rid of type I errors. This is a crucial observation, *we are prepared to admit certain amount of noise in the answer*.

2. Definition of the Model and Notation.

Our data base consists of a collection of textual records. A ***record*** is the "atom" of information, i.e. it is indivisible (a paper in a journal, an article in a paper, etc. if split, the parts become meaningless). We will only use a sequential structure among the records, consequently we could possibly store the data base in tapes. Each record is uniquely identified by its *record number*. A *pre-processed* data base is one in which the contents of the records have been modified according to some function. Pre-processing does not require interpretation of the information.

In the algorithms we present in the next sections we will be looking for a common goal: speed of retrieval. Moreover we have to consider:

(a) efficient algorithms that will work on a typical main-frame computer

(b) simple and efficient algorithms that could be implemented in a microprocessor, and hence allow distributed processing of a query.

Efficiency in our context is the number of operations needed to answer a query. If we have to read all the data base to do this, efficiency is related to the internal speed of processing each character, since CPU time rather than I/O time will be the bottleneck in this case. Otherwise efficiency is proportional to the amount of information transferred and processed.

We assume that the updates to the file are mostly insertions and these can all be appended to the end of the file. In this model, unless we have a complicated pre-processing, updates will not cause trouble.

Our notation will be the same for all the algorithms, namely

n denotes the number of records in the data base.

α denotes the average number of characters per record. $n\alpha$ is the total number of characters in the data base.

m denotes the size of the internal tables used and is also the range of the hashing and signature functions.

k is the number of "patterns" or strings that we are simultaneously searching.

c is the average number of characters per word in our data base, α/c is the average number of words per record.

s is the shifted-preloading factor; (i.e. the number of times that we shift and reload the searched patterns).

r is the minimum number of words in any searched patterns or number of internal tables we use in the membership testing algorithms.

f_i('...') denotes a hashing or signature function that operates on text and returns an integer in the range $0..m-1$. Different subindices indicate different and independent hashing functions.

We may consider a *typical* data base as one with the following characteristics: $n=10^6$, $\alpha=1000$, $m=2^{16}$, $k=100$, $c=5.2$, $s=8$ and $r=3$. This means a data base with 10^9 characters which we search for 100 patterns simultaneously and with tables that will normally fit in main memory.

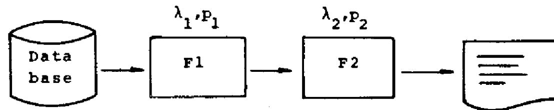
3. About the Queries.

As we mentioned earlier, the process of translating a question into computational terms is the most difficult one, and where we introduce the largest errors (both type I and type II). In this paper we will consider that our queries can be expressed as pattern matching problems.

E.g. identify those records where the string "data base{s} {without structure / unstructured}" appears.

The problem of misspellings, simple errors, case differences, punctuation, ambiguous spellings or ambiguous endings will be treated separately in section 6.

An algorithm that will give no error of type II (will retrieve at least all matching records) will be called a *filter*. Filters, even if they produce a large noise, are extremely useful since we can pipe the data base through several different ones. The next figure illustrates the use of two filters, F1 and F2 which have rates (time/token) λ_1 and λ_2 and noise error levels p_1 and p_2.



Two piped filters.

The resulting error level, if the filters are independent, is $p_1 p_2$, while the running time will be essentially determined by $n\alpha\lambda_1$. Unless we have huge noise levels we will pipe the data base through the fastest filter first. More precisely, if $\lambda_1+p_1\lambda_2 \leq \lambda_2+p_2\lambda_1$ the above filters are in the best order, otherwise they should be inverted. This concept could be extended to any number of filters and consequently reduce the noise error

arbitrarily at a modest search cost. If we use r filters all having the same rate and error level, the time will always be less than $n\alpha\lambda/(1-p)$ and the noise level p^r.

In the next two sections we will concentrate in exact matching of any of several strings in the data base. Counting for a given number of matches, as well as more complicated boolean operations can be implemented on top of this facility. In some sense, exact matching of one of several strings will be our primitive data base operation.

Finally we will try to find algorithms whose efficiency is insensitive to the number of strings to be searched.

4. Algorithms Which do not Require Pre-processing.

Without any structure and without pre-processing, we are bound to read all the data base to answer a query. Our goal, in these algorithms, is to achieve the highest processing speed per character.

4.1. Classical Algorithms.

In this group we have algorithms that use a classical dictionary structure (hashing, binary trees, B-trees, etc.) to store each of the strings to be matched. Most likely hashing would be preferred over the others, since its average search time can be made constant (independent of the number of strings to be searched) and rather small.

Direct application of these methods is too expensive, since they require one search per character read. (Note that at the level of efficiency we want to achieve, searching for the beginning of a word is an intolerable overhead.) This can be improved drastically with the following trick which we call *shifted preloading*.

> Search the data base only every s characters and load the dictionary with each string replicated and shifted s times.

E.g. If we want to search for "Unstructured data bases" and $s=3$, then we will store in the dictionary the following strings:

unstructured data bases
nstructured data bases
structured data bases

In this case we will search in the data base only every 3rd character, i.e. locations 1,4,7,... .

This trick essentially produces an s times speedup at the cost of increased noise in the answer.

4.2. Membership Testing Algorithms.

Any algorithm for membership testing should be applicable, in principle, to our problem. Moreover since we are prepared to accept some level of error we can use approximate membership testers Bloom [1] and Carter et al. [3]. The membership tester will be unsuccessful in most of its trials since only a fraction of the database is expected to be retrieved. Consequently we will select a tester that has a good unsuccessful case. The best suited is the first one described in both [1,3]:

> Let $f('...') \to 0..m-1$ be a hashing function over the set of strings (of fixed length) giving values from 0 to $m-1$. A table of m bits will be set up so that the hashed value of each searched string is set to 1. We will use the shifted preloading technique. The input data base is scanned every s characters. Each time we compute the hashing function and if the corresponding bit in the table is 0 we discard the input, else this record may be a hit.

Obviously, with a single table, unless m is very large, the noise will be too high. This method makes an excellent filter if we use independent and different hashing functions. Here we can do all the filtering steps at once if we can keep all tables in main memory. For example if the probability of a random hit is $p=1/10$, by having r tables we reduce the probability to 10^{-r}. Note that the processing time remains bounded by $1/(1-p) \approx 1.1$ searches here, since we only search in the next table once we succeed in the previous. The expected value of the noise is

$$E[noise] = \frac{n\alpha}{s}\left(1-(1-1/m)^{sk}\right)^r$$

If we have a total of M bits and we want to optimise r so that we have minimum noise, then

$$r_{opt} = \frac{M\ln 2}{sk}\left(1 + \frac{(3-\ln 4)\ln 2}{4(\ln(2)-1)sk} + O(k^{-2})\right)$$

and

$$E[noise] \approx \frac{n\alpha}{s}e^{-\frac{\ln^2(2)M}{sk}}.$$

For this optimal r the tables are about 50% filled, so on the average we do 2 searches for every group of s input characters.

Note that a noise level of 1 per billion, which should be considered excellent, requires less than 45 bits per string to be searched; this is very economical.

The second method described in both [1,3], although more economical in storage, requires two comparisons per search and the computation of two hashing functions. The methods proposed in [1,3] work with a single table, i.e. all the filters work superimposed over the

same bit table. The noise remains almost unaffected under our variant.

In principle we assume that all the functions have equal cost. This need not be so, and since the first few hashing functions will have a dominant effect in the total running time, we may want to select particularly fast functions to be first. An example of such economy may be to insist that the first function uses only the first four characters of the strings.

4.3. Pattern Matching Algorithms.

These algorithms are motivated from the fast pattern matching algorithm by Knuth, Morris and Pratt [8]. Briefly, for each string we build a programme that, in linear time (one action per character), determines whether there is a match or not. For example if we want to match the string "abad" the programme will look like:

```
Start: Read; if not "a" goto Start;
1:     Read; if "a" goto 1;
             if not "b" goto Start;
2:     Read; if not "a" goto Start;
3:     Read; if "b" goto 2;
             if not "d" goto Start;
       Succeed;     ("abad" was matched)
```

These programmes are extremely simple and could be implemented in hardware with a small memory and a clock. We can envision then, several "pattern matching micros" working in parallel over a common input stream. Each micro is responsible for one string, and each input character is made available to all micros in parallel. Here the processing speed will be limited to one character per memory cycle. The number of strings we can search simultaneously is limited by the number of microprocessors. The latter suggests that this type of method is not likely to be very successful.

The improvements by Boyer and Moore and later Galil [2,4,6] may drastically speed up the searching. Their main idea is to skip as many characters as possible once the partial matching failed. These algorithms would require more sophisticated micros, and if the input is shared among several micros it is unlikely that all the micros would agree on skipping several characters. The advantages over KMP [8] algorithm would disappear.

4.4. Other applications.

The algorithms described in sections 4.1 and 4.2 are suitable for any type of string searching. They could be used, for example, to implement string search in a text editor.

Harrison [5] was probably the first to propose the use of hashing functions for fast string searching. Tharp and Tai [12] use signatures of entire lines to quickly search for occurrences. Both methods assume that we compute the signatures of all lines and keep them to later search with the signature of a given string. We would do the converse: set up a table with the string to be searched and compute the hashing function on the strings as we search.

5. Methods Using Pre-processing.

The algorithms presented in this section will use a copy of the data base which has been mechanically (without interpretation) changed. Very frequently we will use a signature function. A *signature* function is a function which maps words into integers in the range $0..m-1$. This function is clearly many-to-one (not invertible) and in many aspects is a hashing function. Here the complexity of computing the signature is not too relevant and consequently we could use sophisticated functions.

When we use a signature function, the preprocessing consists of converting all the words of the data base into their signatures. This conversion by itself provides three advantages:

(1) there is a significant reduction in the volume of the data base (if an integer smaller than m can be represented in two bytes, for example, then we have a compression factor of $c/2$), and consequently in the processing time.

(2) the problems originated by separators (multiple blanks, tab characters, new-line characters, etc.) disappear.

(3) In most computers and some programming languages it is easier and more efficient to work with integers than it is to work with strings.

5.1. Shift-And Method.

The Shift-And method searches in a pre-processed data base in the following way:

> Let r be the minimum number of words in any given string to be matched. We will use an internal table T with m locations, each location with capacity at least r bits. The loading of the table is done in the usual way, except that for each string the first word will set the first bit of the corresponding location, the second word will set the second bit, and so on.

E.g. If the strings to be searched are:

data bases without structure
unstructured data bases

$r=3$

f('data') = 10
f('bases') = 101
f('without') = 587
f('structure') = 707
f('unstructured') = 200

...	0	...	0	...	0	...	0	...	1	...
...	0	...	1	...	0	...	1	...	0	...
...	1	...	1	...	0	...	0	...	0	...
...	1	...	0	...	1	...	0	...	0	...
	10		101		200		587		707	

Table T for the above example.

For each signature S of the data base we perform the operation

```
A = ( shiftleft(A,1) or 1 ) and T(S);
if ( A and 2^(r-1) ) ≠ 0 then 'succeed';
```

The counter A keeps track of partial matches, e.g. its second bit on means that the before last signature matched the signature of a first word and the last signature matched a second word.

The noise produced by this method has a formula similar to the previous case

$$E[noise] = \frac{n\alpha}{c}\left(1-(1-1/m)^k\right)^r .$$

Here we cannot alter r so our only choice to lower the noise is to increase the value of m. If the value of r is too small ($r=1$ or $r=2$), in order to obtain reasonably small noise we can add information to each entry on whether one of its bits correspond to a last word of a string. In terms of the previous example, an associated table T* should contain T*(707) = 01000 (a fourth word being last) and T*(101) = 0100 (a third word being last).

For the latter case, if there are k_i strings with i words, ($k=k_1+k_2+\ldots$) then the expected noise is less than

$$E[noise] \le \frac{n\alpha}{c}\sum_{i\ge 1}\left(1-(1-1/m)^k\right)^{i-1}\left(1-(1-1/m)^{k_i}\right).$$

5.2. Inversion.

Although inversion [7] is typically a successful technique for this kind of problem, it is easy to see that in this case it may fail badly. For example, for our string "data bases without structure" we may find that

roughly all papers contain the word "without", about 50% contain the word "structure" or the word "data" and probably 20% contain the word "base". Consequently our string searched in an inverted file will retrieve 5% of the data base. This will normally be unacceptable even as a filter. These may not be true across all applications, some technical terms may be rather unique, but clearly this method is too sensitive to common words to be of real interest.

Several commercially available systems allow text searching facilities using inverted files. These systems use data bases which are far more restricted than the ones we consider.

(a) Searching is done in the title, abstract and keywords but the text itself is not searched. This restriction is crucial to the success of inverted files.

(b) The inversion is done under a controlled vocabulary and common words ("stop words") are not used.

(c) The inverted file may contain information about the position where the word appears, so that phrase searching can be done.

5.3. Inversion on Signatures.

An inverted file on the signatures of the data base is likely to be worse that direct inversion. We will now introduce a new signature function that works on pairs of words rather than on single words. For example

f('unstructured','data') = 107
f('data','bases') = 532

The pre-processing will consist of a file with the signatures of the first and second word, second and third, third and fourth and so on. Clearly a data base pre-processed in this way will be of the same size as the previously described pre-processing (one signature less per record).

We will now invert the data base on these signatures and search it as such. For example, suppose that we want to search for the string "unstructured data bases". Using the above definitions of the signature function, we will need to search the entries 107 and 532 in the inverted file and find all the records that are referenced in both lists. On the average, if we have a string with r words, we will have to retrieve a fraction $(r-1)/m$ of the total data base. The noise is

$$E[noise] = nm\left(\frac{\alpha}{cm}\right)^r$$

Note that we do not have the restriction on m that we had before, namely that we had a table in main memory of size m. The value of m will only

affect how atomised is the inverted file and consequently affect linearly the total size of the data base. This method searches only one string at a time.

This concept may be further extended to use functions that take into account 3 or more words at a time. By doing this we will increase the amount of information on the ordering of the words, and hopefully reduce the noise in our queries. The signatures will become much more "randomised", since there are very few groups of 3 words or longer that occur with high probability. Both advantages are eclipsed because for any given query we will now reduce the number of list to "and". For example, using tri-words, the string "unstructured data bases" will retrieve an entire list instead (of a likely smaller set) the intersection of two list as proposed.

The idea of hashing sequences of tokens to include information on order was present in the method suggested by Harrison [5].

5.4. Related algorithms.

Signatures and superimposed codes have been used extensively in searching: Gustafson as reported in [7], Knuth [7], Rivest [10] and more recently Roberts [11] and Pfaltz et al. [9].

The main differences between our work and the above referenced are:

(1) The records have either a fixed format of a well defined structure.

(2) The queries are well defined in advance and of most concern are partial match queries.

(3) A signature replaces (or describes) the entire record.

(4) We keep the values of the hashing function as opposed to keeping a bit map representation of it.

6. Signature Functions.

In Section 5 we use various signature functions which were defined to be essentially hashing functions. Since the signatures have to be computed once over the whole data base (pre-processing) and then only on the words of each query, we can in principle afford a computationally expensive signature function.

We will use this extra power to try to correct misspellings, single errors, endings etc. There are two basic approaches to build such a signature function.

(a) The Soundex method (as reported in [7]), or a modified version of it to allow for the proper range. Such a method will group letters according to their sound, eliminate duplicates and endings etc. in such a way that similar sounding words are likely to end with the

same soundex code.

(b) A thesaurus method for which each group of words is numbered sequentially. Each word is looked up in the thesaurus and its group number becomes its signature. The thesaurus should be expanded with plurals, multiple spellings, verbs in all their tenses and complemented with some strategy for words that still cannot be found in it. For the latter we could use a closest match to an existing word or a soundex method.

Of course all signature functions should be case independent.

When posing a query, for each string we form (maybe automatically) all the variations we think are appropriate. This will generate a rather large set of strings which, when converted to signatures, will surely contract due to the properties of the functions.

The thesaurus method seems vastly superior to others since it may catch synonyms overlooked by the initial posing of the query.

7. Conclusions.

In this paper we present several new algorithms to quickly search a textual data base. The emphasis is in speed of internal processing. The inclusion of our version of signature functions gives several advantages: reduced volume, speed of processing, handling of ambiguities, misspellings, simple errors, etc.

8. Acknowledgements.

The author wishes to acknowledge F. Burkowski who sparked our interest in the subject and the useful discussions and collaboration by P.A. Larson, F.W. Tompa and N. Ziviani.

9. References.

[1] Bloom, B.H.: Space/Time Trade-Offs in Hash Coding with Allowable Errors; Comm. ACM, 13(7):422-426 (July 1970).

[2] Boyer, R.S. and Moore, J.S.: A Fast String Searching Algorithm; Comm. ACM, 20(10):762-772, (Oct 1977).

[3] Carter, L., Floyd, R., Gill, J., Markowski, G. and Wegman, M.: Exact and Approximate Membership Testers; Proceedings 10th SIGACT-STOC Conference, San Diego, CA. :59-65, (May 1978).

[4] Galil, Z.: On Improving the Worst-Case Running Time of the Boyer-Moore String Matching Algorithm; Comm. ACM, 22(9):505-508, (Sep 1979).

[5] Harrison, M.C.: Implementation of the Substrings Test by Hashing; Comm. ACM, 14(12):777-779, (Dec 1971).

[6] Horspool, R.N.: Practical Fast Searching in Strings; Software: Practice and Experience, 10(6):501-506, (June 1980).

[7] Knuth, D.E.: The Art of Computer Programming, Sorting and Searching; Addison-Wesley, Don Mills, Ont. (1973).

[8] Knuth, D.E., Morris, J.H. and Pratt, V.B.: Fast Pattern Matching in Strings; SIAM J on Computing, 6(2):323-350, (1977).

[9] Pfaltz, J.L., Berman, W.J. and Cagley, E.M.: Partial-Match Retrieval Using Indexed Descriptor Files; Comm. ACM, 23(9):522-528, (Sep 1980).

[10] Rivest, R.L.: Partial-Match Retrieval Algorithms; SIAM J on Computing, 5(1):19-50, (Mar 1976).

[11] Roberts, C.S.: Partial-Match Retrieval Via the Method of Superimposed Codes; Proceedings IEEE, 67(12):1624-1642, (Dec 1979).

[12] Tharp, A.L. and Tai, K-C.: The Practicality of Text Signatures for Accelerating String Searching; Software: Practice and Experience, 12(1):35-44, (Jan 1982).