Information Technology: Research and Development (1982), *1* (131–156)
© 1982 Butterworths

IMPLEMENTING A PROBABILISTIC INFORMATION RETRIEVAL SYSTEM

M. F. PORTER

Department of Earth Sciences, University of Cambridge, Downing Street, Cambridge CB2 3EQ, England

(*Received 12 October 1981, revised 4 December 1981*)

ABSTRACT

CUPID is an information retrieval system which makes operational a simple probabilistic IR model, and differs markedly from traditional IR systems. This paper describes the basic structure of the system, the model upon which it is based, and the work that needs to be done to set such a system up. The size of CUPID can be kept very small by using the GOS program package to build up document collections and generate suitable index terms from them. An extended example is given of CUPID in use with a small document collection. CUPID involves many activities which are easily done in a relational DBMS, and the possibility of implementing CUPID within a DBMS is discussed.

1. INTRODUCTION

This paper should be of interest to two groups of people, firstly to those familiar with the GOS program package, who will discover ways in which data handled by GOS can be connected to an IR (information retrieval) system, and secondly to the much larger group of workers in IR, especially at the research end, who will discover how a fairly advanced probabilistic IR system can be made operational with surprisingly little effort. It is generally supposed that implementing a high quality IR system is a difficult and expensive operation. I wish to suggest that this is not the case, and that a practically useful IR system of the kind described here can, given the appropriate expertise, be set up for use in a few man months on almost any kind of present day computer. The probabilistic IR model which I have followed is essentially simple, but at the same time very effective. In fact one could go so far as to claim that we do not at the moment know of any way in which the performance level offered by this model can be significantly improved.

It is hoped that the account offered here will encourage other groups to set up similar systems. Such groups might include University Departments and Colleges, Research Organizations, and Businesses and Institutes which run their own information sections. CUPID has been used effectively with a collection of over 10000 documents. We do not at the moment know whether it would be equally effective with 100000, but have no reason to suppose that it would not. There is no provision in CUPID for dynamic updating of the document collection, and it would need to be considerably extended to provide for this.

The 'least effort' way of implementing the model at Cambridge has been to use GOS as a system for establishing and indexing a document collection, to use a sort-merge system to set up the files needed by the IR system, and then to have a small set of programs to operate the IR system. This latter set of programs is called CUPID (Cambridge University Probabilistic Independence Datamodel). These three stages, the use of GOS, establishing the files for CUPID, and CUPID itself, are described in the sequel. GOS itself can be bypassed, when the data is simple, or when some other system exists which will do the same work for the user, and this was the case in the first CUPID implementation at Cambridge, and in the implementation currently running at University College, Dublin, which has been implemented there by Alan Smeaton. Conceivably the whole of CUPID and the sort-merge activities could be embedded in a suitable relational database, as will again appear in the sequel.

2. AUTOMATIC INDEXING IN GOS

GOS (Porter, 1980a,b,c) is a general purpose system for constructing and manipulating catalogues by computer. A catalogue may be represented by one or more *files*, and each file will be broken up into *records*, the records themselves consisting of separate items of data in *fields* (or elements, as they are more properly called in GOS). The fields in a record are arranged in some hierarchical structure, the groupings within which may be used to reflect natural groupings in the data. In this way GOS may be used to represent bibliographic data along the lines of the MARC system, although the total range of possible applications for the package are really very wide. It was in fact originally written with museum data processing in mind.

In the sequel we will follow IR practice and use *document* to mean an item that we wish to establish for retrieval in an IR system, and *term* to mean an item of data that will be used to index one or more of the documents. It must not be supposed however that we are using these words in a narrow bibliographic sense. By a document we mean any record (and in this context a GOS record) which may be bibliographic but may equally contain information about a print or drawing, a social history object, a historical event, or an archaeological or geological find. By a term we mean a string of text characters which may be used to index a document. Thus a term could be a complete word of English, or a fragment of a word, or a group of words separated by spaces.

The process of taking a document and establishing a list of terms which will be used to index it is the processing of *indexing*, and it is assumed that indexing is done automatically by the computer, using the data in the GOS record which describes the document. Quite possibly the record will contain index terms supplied manually by the author of the document, or by someone who has indexed the collection. In this case this list of terms may, or may not, be made available to the automatic indexing process, which may, or may not, generate indexing terms the same as this supplied list of terms. In the example below the manually supplied term lists are used, but give rise to rather different terms in the automatic indexing.

We now give an example of the use of GOS in automatic indexing. Suppose that the documents (bibliographic this time) contain a field, denoted by <t>, which contains the document title. And suppose that terms are to be extracted from the <t> field by the following operations:

1. Divide the <t> fields into 'words', each word being text characters delimited by spaces.

Then for each word do the following:

2. Discard accents on letters and indications of italicization.
3. Discard all characters other than letters and digits.
4. Ignore the word if it now contains a single character, otherwise do operations (5) onwards.
5. Force all the letters to be lower case.
6. Ignore the word if it occurs in some appropriate list of stopwords, otherwise do operations (7) and (8).
7. Strip off the suffix of the word using some appropriate stemming algorithm.
8. Output the result as a term indexing this document.

Steps (1) to (8) are characteristic operations in automatic indexing. For example, if the words of the title are:

G. E. Moore's philosophy before 1903: the genesis of the *Principia Ethica*.

Then after operation (2) the list of words is as follows:

G.	the
E.	genesis
Moore's	of
philosophy	the
before	Principia
1903	Ethica

After operation (3) this becomes

G	the
E	genesis
Moores	of
philosophy	the
before	Principia
1903	Ethica

After operation (4)

Moores	genesis
philosophy	of
before	the
1903	Principia
the	Ethica

After operation (5)

```
moores        genesis
philosophy    of
before        the
1903          principia
the           ethica
```

After operation (6)

```
moores        genesis
philosophy    principia
1903          ethica
```

And after operation (7)

```
moor          genesi
philosophi    principia
1903          ethica
```

One can of course argue over details here. For example, it is not clear whether digit strings such as '1903' should be included as index terms, or should be simply discarded. In practice these operations might be tuned slightly depending on the subject area of the documents. The important point is the ease with which these operations, or variants of them, can be performed with the powerful editing capabilities supplied in GOS. An important facility in GOS is the so-called anel system, in which operations can be performed on text strings. It will be assumed that the text string contains a word. Then in the anel system the operations (2) to (8) can be performed as follows:

2. re(re(neq '← \') d d)

As in SNOBOL, a string is analysed using a cursor, which at any given time points to a particular character in the string. neq '← \' tests for the character at the cursor not being either '←' or '\'. The command re causes repetition, and re(neq '← \') causes the cursor to be moved right while the character at the cursor is not '←' or '\'. The cursor will therefore stop at the first '←' or '\' character, and the command d then deletes this character, while the second d deletes the following character. The re command surrounding the whole construction causes this total operation to be repeated all the way down the string, and the net effect is therefore to delete from the string all substrings of the form '←$' and '\ $', where $ is any character. Since accents are represented as '←a' for acute, '←g' for grave etc., and indications of shifts to italic and roman (or underlining and normal) by '\U' and '\N', this gives the desired effect.

3. b cls('ct0' re(re(cl 'LlD') d))

Command b takes the cursor back to the beginning of the string. The rest of this command causes characters other than those of class 'L', 'l' or 'D' (i.e., upper case letter, lower case letter or digit) to be deleted from the string. Characters can be put into any user defined classes in a so-called class table, and here the class table ct0 is being used, in which upper case letters are in class 'L' and so on.

```
4.  b tab 2
```

b takes the cursor back to the beginning of the string, and then tab 2 moves it right two places. If the string has length less than two characters, this operation will fail, and this failure effect can be used to cause the abandonment of operations (5) to (8).

```
5.  b re(s)
```

Command s causes the character at the cursor to be forced from upper to lower case, and re(s) causes the whole string to go into lower case. Characters other than upper case letters are not affected.

```
6.  not( + among 'a' 'about' 'above'
                    . . . .
                 'yours' 'yourself' 'yourselves')
```

We assume that we have a complete stopword list here. This command only succeeds if the word is not among the given list; in other words it fails if the word is in the given list, and then operations (7) and (8) are not performed. The list that we have used in practice is the one given on pp. 18–19 of van Rijsbergen (1980), but of course any list is possible. (The list is looked up by a binary chop technique in GOS, so that with the 250 of the actual list, only about eight comparisons are ever made in each call of this command.)

```
7.  + stem()
```

Here there is a switch out of the anel into a system called STEM. This is a specially written GOS module which applies the stemming algorithm fully described in Porter (1980d). The particular merits of the algorithm in the present context are that it is small enough to fit simply and naturally into GOS, and fast enough for its application to every document term (rather than a derived vocabulary list of such terms) to be practicable. Furthermore, the result of the stemming for a particular term is a function of that term alone, and not of the total vocabulary (as is the case with some stemming algorithms). This greatly simplifies its use. It is probably worth noting that despite its simplicity, the algorithm leads to retrieval performance as good as that achieved with more elaborate stemming systems (Lennon *et al.*, 1981). Of course use of the stemming algorithm will only be appropriate for titles in English.

```
8.  + ffterm()
```

Here a specially written GOS system, FFTERM, is called which outputs the text of the term in a fixed field format. This is for the benefit of the IBM sort-merge system.

A formal account of the semantics of expressions like the ones given above can be found in the GOS documentation, and here we have only given a rough sketch of how they work. The point to which we wish to draw attention, however, is their remarkable brevity. Operations such as (1) to (8) are an important part of practical IR work, and often cause programmers a lot of difficulty, particularly programmers who are not too experienced in non-numerical computing applications. But to a user versed in GOS they become quite simple.

Operations (2) to (8) form the inner part of a loop which also has the job of performing operation (1), to strip out successive words from the title field. The entire process may be defined in GOS as a construction called getterms0 as follows:

```
def('getterms0' + a
    re
        re(eq ' ') 1
        n re(neq ' ') r
        to(1)
        try
            v(1 re(re(neq '← \') d d)
                b cls('ct0' re(re(cl 'LID') d))                    (2.1)
                b tab 2
                b re(s)
                not( + among 'a' 'about' 'above'
                                    . . .
                             'yours' 'yourself' 'yourselves')
                + stem()
                + ffterm()
            )
)
```

and getterms0 can be applied to each <t> field in a record by using the construction:

```
+ e(re(nc(<t>) do 'getterms0'))
```

(Normally of course one would not expect more than one <t> field per record, but repetitions are possible, if for example the record describes one document which is held within another document.)

Variations immediately suggest themselves. For example the document records may contain lists of <k> fields, where each <k> field contains a keyword explicitly supplied for indexing purposes. The <k> fields will be used for index extraction in precisely the same way if we use the construction

```
+ e(re(nc(<t>) do 'getterms0')
    re(nc(<k>) do 'getterms0'))
```

On the other hand, we may prefer to use the construction

```
+ e(re(nc(<t>) do 'getterms0')
    re(nc(<k>) do 'getterms1'))
```

where getterms1 is a construction whose definition differs in certain significant respects from getterms0. For example, getterms1 might contain no stopword list, so that if 'and' was supplied as a keyword it would be indexed. One might wish to index under 'and' if for example a document was a paper in linguistics which dealt with the use of the connective 'and' in English.

More generally one could have an entire library of constructions along the lines of getterms0 and getterms1, which in GOS is held in a so-called GOS element library, and members of this library could be called up on the various fields in a document record, to give a very wide range of possible indexing effects. In this way great versatility can be achieved. These ideas will all be very familiar to users of GOS.

3. SETTING UP THE FILES FOR CUPID

The indexing process described above produces a file consisting of a simple list of document numbers and terms:

```
1 moor
1 philosophi
1 1903
1 genesi
. . .
```

In the language of relational databases (Date, 1981, supplies a standard text), the result of the indexing can be thought of as a two-column relation called I which has the form $[D,t]$. Column D contains the number of the document, and column t contains the text form of a term which indexes the corresponding document in the D column. It will be useful to introduce some relational database nomenclature at this point for two reasons: because it gives an easy way of describing the operations which have to be done to set a document collection up for retrieval by CUPID, and because the whole of CUPID might very well be implemented within a relational database, in which case it may be possible to perform these operations very much as they stand.

In the discussion of relations which follows I have been influenced by the relational database management system CODD (King, 1979), and its algebraic query language CHIPS, designed by Tim King and Charles Jardine in the Computer Laboratory, Cambridge. To fix the ideas here, we say that a relational database operates on *relations*, and a relation may be thought of as a matrix with a small number of named columns and a large number of rows. Each row contains a *tuple*, and if there are n columns in the relation the rows may be called *n-tuples.* It is assumed that the tuples are all different and ordered. That is, if two adjacent tuples are

$$(a_1, a_2, \ldots a_n)$$
$$(b_1, b_2, \ldots b_n)$$

it will be the case that

$$a_i = b_i \quad \text{for } i = 1, 2, \ldots j-1$$

and

$$a_j < b_j$$

for some j in the range $1 \leq j \leq n$. Each column may contain strings or numbers, but may not mix both. The definition of ordering between strings is left open. In practice an ordering is chosen which can be tested rapidly, but one can keep in mind alphabetical ordering in the discussion that follows.

Of the typical operations performed on relations, select, project, join, union, intersection and difference, a project is liable to interchange columns and therefore liable to cause a sort of the tuples. We can expect these sorts to be slow compared with other operations, and so in the sequel, projects which give rise to sorts will be scrutinized rather carefully. (We will employ only a simple version of join that does not involve any sorting.)

The relation I is unsorted. More exactly, it is sorted on the D-column, but not on the t-column (assuming that the indexing process numbers the documents 1, 2, 3 . . . in a natural way). Furthermore the relation could contain duplicates, since the same index term can be picked up out of one document from a number of different places. It is assumed here that in the relations subsequently formed from this one, a precise ordering is achieved and duplicates are removed.

The first operation is a projection:

$$J[t,D] := I[D,t] \rightarrow 2,1 \tag{3.1}$$

This means that relation J is derived from I by reordering columns 1 and 2 as 2,1 (i.e., swapping them round), and then resorting. J has the form $[t,D]$, and the presence of the expressions $[D,t]$ and $[t,D]$ in (3.1) purely acts as a program comment. The forward arrow symbol '$\rightarrow$', used to denote projection, corresponds to the CHIPS symbol '%'.

The next operation is a projection, which gives a simple list of terms:

$$K[t] := J[t,D] \rightarrow 1$$

In other words K is derived from J by retaining only column 1 of J. Duplicates in J are removed.

We assume the existence of a function $NUM\,(R)$, where R is a relation. The value of this function is a new relation with an extra column at the end consisting of the numbers 1, 2, 3 . . . in turn. $NUM\,(R)$ therefore numbers the rows in relation R. A unique integer is therefore given to each term of K in the relation L, formed by:

$$L\,[t,T] := NUM(K\,[t])$$

We can suppose that L is formed at once from J by the command

$$L\,[t,T] := NUM(J\,[t,D] \rightarrow 1) \tag{3.2}$$

The relation TD is formed from J by replacing each t in J by the corresponding number T supplied in the L relation. This can be done by joining L with J over one column (the first), and it is assumed that the diadic operator *(1) will do this, and then projecting the result:

$$M\,[t,T,D] := L\,[t,T] \,{*}(1)\, J\,[t,D]$$
$$TD\,[T,D] := M\,[t,T,D] \rightarrow 2,3$$

or as a single command:

$$TD\,[T,D] := (L\,[t,T]\ *(1)\ J\,[t,D]) \rightarrow 2,3 \tag{3.3}$$

The relation *DT* is now formed by a simple inversion of *TD*,

$$DT\,[D,T] := TD\,[T,D] \rightarrow 2,1 \tag{3.4}$$

We next assume the existence of a function *HASH* (*R*), the value of which is relation *R* with each string in the last column of *R* replaced by an integer in the range 1 to *h* derived from a hash function applied to the string. Then *L* is inverted to make a relation *Tt*,

$$Tt\,[T,t] := L\,[t,T] \rightarrow 2,1 \tag{3.5}$$

and a relation *HT* is formed as follows:

$$TH\,[T,H] := HASH\,(Tt\,[T,t]) \tag{3.6a}$$
$$HT\,[H,T] := TH\,[T,H] \rightarrow 2,1 \tag{3.6b}$$

As well as producing the original relation *I*, GOS can be used to produce a print of each document in a form suitable for viewing during retrieval. This can be thought of as a one-column relation *d*, where the *n*th row consists of the text form of the *n*th document. This is numbered to form a relation *Dd*

$$Dd\,[D,d] := NUM\,(d\,[d]) \rightarrow 2,1 \tag{3.7}$$

CUPID uses five of these generated relations, namely:

TD	the term–document index
Tt	the term list
DT	the document–term list
Dd	the document list
HT	the term list by hashed term value

On the IBM 370/165 at Cambridge, steps (3.1) to (3.7) to set up these five relations can be done quite easily by the IBM sort-merge and a specially written program of about 70 lines. Step (3.1) is a single operation by the sort-merge system. Steps (3.2), (3.3), (3.5) and (3.6a) involve no sorting, and derive the relations *TD*, *Tt* and *TH* from *J*. This can be done in a single pass over the file representing *J* by the specially written program. Steps (3.4) and (3.6b) to derive *DT* from *TD* and *HT* from *TH* can be done by the sort-merge package. It is also possible at Cambridge to do these operations in CODD, which is very suitable for handling large volumes of relatively static data, or within GOS, although the sorting operations would then be very much slower.

The purpose of using term numbers rather than the text of the terms in the relations *TD* and *DT* (more especially in *DT*) is to save space in the direct access files which CUPID sets up and uses. To find the text *t* of term number *T* is just a matter of looking *T* up in the direct access file corresponding to the relation *Tt*. To find the number *T* of the term with text *t* is a little more tricky: first *t* is hashed to give a number H, H is looked up in HT, and the term numbers there found are looked up in Tt to give a list of textual terms which may be compared with t. So long as t is in Tt a match will be found. This is admittedly a far from optimal approach as far as access to secondary storage is concerned, but compared with the total access to secondary storage which will take place in a retrieval run, the inefficiency here is not too great. As it stands, it has the advantage that all direct access files required by CUPID can be indexed simply by the integers 1, 2, 3 . . . which helps keep its design simple.

Of the five relations given above, the relation Dd is usually the largest, and will frequently be many times larger than the other four put together. If disk space is tight, an economic way of running CUPID would be to have the text versions of the documents (i.e., a suitable print of the relation Dd) to hand on line-printer paper or computer generated microfiche, and to dispense with the relation Dd as a direct access file. CUPID only uses this file for printing documents out for the user, and could just as easily direct him to a hard-copy file containing the documents. One would of course expect this approach to be much more laborious for the poor human, but it is worthwhile remembering that an effective IR system for a large document collection can be established without there needing to be provision for all the documents residing in text form on disk.

The form of the five relations TD, Tt, etc. is machine dependent, and will be a function of whatever system has been chosen to generate them out of the relation I derived from the GOS file of the documents. Consequently the part of CUPID which sets the relations up as a direct access file will be machine dependent, at least on the input side. Without fear of ambiguity, the direct access files set up by CUPID from the relations can be given the same names as the relations, namely TD, Tt, etc. These direct access files have one of two forms, either they consist of vectors of numbers indexed by the integers 1, 2, 3 . . . (in the case of TD, DT and HT) or of strings of text indexed by the integers 1, 2, 3 . . . (as in the case of Tt and Dd). Thus in the case of TD, we have that a particular integer T indexes a vector

$$D(T) = (D_{T1}, D_{T2}, \ldots D_{Tk(T)})$$

in other words term number T indexes documents $D_{T1}, \ldots D_{Tk}$ where k is a function of T. It is assumed that $D_{T1} < D_{T2} < \ldots$ Similarly in the case of DT, we have that a particular document D indexes a vector

$$T(D) = (T_{D1}, T_{D2}, \ldots T_{Dl(D)})$$

in other words document number D is indexed by the terms $T_{D1}, \ldots T_{Dl}$ where l is a function of D. It is assumed that $T_{D1} < T_{D2} < \ldots$ In the case of Tt, a particular integer T indexes a piece of text which is the textual representation of term T.

4. THE RETRIEVAL MODEL USED BY CUPID

CUPID implements a simple probabilistic model for document retrieval. The theoretic development is given in Robertson and Sparck Jones (1976), and is further developed in Chapter 6 of van Rijsbergen (1980). Here we merely summarize the results. In CUPID a query is represented by a vector of terms:

$$Q = (T_1, T_2, \ldots T_{k(Q)}) \tag{4.1}$$

This is of course a very simple representation of a query compared with a Boolean expression of terms, which is the traditional way of representing queries in practical IR systems. In the course of a retrieval run with the query Q, CUPID builds up two sets: a set S, which contains the numbers of those documents which have been seen by the user, and a set R, which contains the numbers of those documents which the user has judged to be relevant to his request. R is a subset of S. As the retrieval run continues, the terms in Q may change.

With T_i we associate the following values:

n_i is the number of documents indexed by T_i, that is

$$n_i = k(T_i)$$

$k(T_i)$ being the length of the vector $D(T_i)$.

r_i is the number of relevant documents indexed by T_i, and is the size of the intersection of R with the elements of $D(T_i)$.

N will indicate the total number of documents in the collection, and the number of items in R will again be denoted by R.

If p_i is the probability that a relevant document is indexed by T_i, then it is possible to estimate p_i by r_i/R. If q_i is the probability that a non-relevant document is indexed by T_i, then a possible estimate for q_i is $(n_i - r_i)/(N - R)$, although it is conceded that there may be relevant documents outside the set R. In the probabilistic model followed by CUPID the correct term weight for T_i is given by

$$\log \frac{p_i(1-q_i)}{q_i(1-p_i)} \tag{4.2}$$

which may be estimated by

$$\log \frac{r_i(N-n_i-R+r_i)}{(R-r_i)(n_i-r_i)} \tag{4.3}$$

In fact for small samples this is not a particularly realistic estimate, and we prefer to use

$$w_i = \log \frac{(r_i+\frac{1}{2})(N-n_i-R+r_i+\frac{1}{2})}{(R-r_i+\frac{1}{2})(n_i-r_i+\frac{1}{2})} \tag{4.4}$$

Using (4.4), we can therefore estimate

$$\sum_{i=1}^{k(Q)} x_i \log \frac{p_i(1-q_i)}{q_i(1-p_i)}$$

as

$$f(D) = \sum_{i=1}^{k(Q)} x_i w_i \tag{4.5}$$

where $x_i = 1$ or 0 according as T_i does or does not index D, i.e., according as T_i does or does not occur in $T(D)$, and according to the model of Robertson and Sparck Jones, $f(D)$ can be used as a measure of how far document D is likely to be relevant to query Q. $f(D)$ itself is not a probability, but $f(D_1) > f(D_2)$ means that the probability of relevance of D_1 exceeds the probability of relevance of D_2. In the model the range of terms included in the summation of (4.5) is left open, but here we suppose that it is given simply by the terms of the query.

If therefore the documents in the collection are ordered by decreasing f-value, and presented to the user in that order, the user should observe a high proportion of relevant documents at the beginning of the list with a general tailing off of relevance thereafter.

Another aspect of query modification (besides computing new weights) consists of adding new terms into Q. Such terms can be presented to the user as likely candidates for inclusion in an expanded version of Q. The approach adopted here, which has been found to be quite effective experimentally, is to take the terms which index the various documents in R, and rank them by decreasing order of association with R. It is not at all clear what constitutes a suitable measure of association, but it was found empirically while developing CUPID that the following measure is attractive:

$$g(T) = \frac{r}{R} - \frac{n}{N} \tag{4.6}$$

where r is the number of documents in R indexed by T, and n is the number of documents indexed by T in the whole collection. g avoids giving a high association measure to low frequency terms which happen to occur in R with high concentration. Thus if $R = 10$ and $N = 1000$, then $r = 1$, $n = 1$ will produce a lower g value than $r = 3$, $n = 50$. Very low frequency terms are not normally useful in retrieval, and the user will not want to see them heading a list of additional useful terms. Equally g avoids giving a high association measure to terms which attain a high r value because they are common. So $g(T) = 0$ if $r = 1$, $R = 10$ and $n = 100$, $N = 1000$.

Spotting the usefulness of formula (4.6) is a good example of the way in which a system like CUPID can be used to complement IR research. Formula (4.6) can be expressed as

$$g(T) = P(T \mid \text{rel}) - P(T)$$

i.e., the probability that T indexes a document given that the document is relevant, minus the probability that T indexes a document. In other words it is the extent to which the probability of T indexing a relevant document exceeds the probability of T indexing any document, and expressed in this way it becomes a very plausible measure of the usefulness of T in indexing relevant documents. It was however first found to be useful while experimenting in CUPID with different association measures.

5. SETTING UP THE QUERY

We have seen that the query, Q, is represented by a vector of terms. It is not of course expected that the user should explicitly present this vector to the system, but rather that it should be formed from some other data which he presents to the system. An approach which has a long pedigree in IR research is to have the user input a piece of text which represents his query, for example:

Guides to zoological and biological nomenclature (5.1)

And then establishing a query vector from such a piece of text is exactly analogous to the process of indexing a document, the only difference being that instead of having a collection of separate fields from which the index terms may be extracted, we have effectively a single field. A query such as (5.1) is very similar in form to a document title. This suggests that the indexing process appropriate to a query could be like the one adopted for a <t> field in the example indexing process in section 2 above. This further suggests that GOS can be used to set up the vector Q from the user's query, and this has been the approach adopted in practice. There are actually two possibilities. Either the user supplies a simple piece of text to represent his query, as in the case of (5.1), or he supplies a complete GOS record, which can then be subjected to index term extraction exactly as was done with the original set of GOS records used to represent the document collection. In the examples given later we have used the former approach. The latter approach is more elaborate but has the advantage that the index terms may be extracted from different parts of the record in different ways, and it may be possible for the user to take advantage of this. So to continue the earlier example he might force 'and' to be an index term by including it in a field where it is not treated as a stopword.

6. THE PRACTICAL REALIZATION OF THE MODEL

In the implementation of CUPID on the IBM 370/165 at Cambridge, the commands issued by the user are in fact *phoenix* commands, phoenix being the multi-access system plus command language for this machine. Each phoenix command then calls up CUPID with a suitable set of CUPID commands. This approach is clumsy, but gets over the problems of complete interactive working not being available to users of the 370 computer. (The pool, mentioned in section 8, is written out to disk between commands, thereby preserving CUPID's internal memory.) Nevertheless it is instructive to note that CUPID can be run non-interactively, even if one is relying on some level of pseudo-interaction here. In fact the system could be run with no interaction at all. We now describe some of these commands.

QUERY FROM *f*

This takes a user query (such as (5.1)) from the file *f*, and establishes from it a query vector of terms, Q. As we have seen, this command will make use of GOS.

DQ *n*

where n is some integer. This forms a set called M from the first n documents in the ordering of documents not in S, the set of documents which have been seen by the user, by decreasing f-value, f being given by (4.5). The summation extends over the terms of Q. Using (4.1) as the representation of Q, this involves doing a merge on k (Q) document lists D (T_1) . . . D ($T_{k(Q)}$). The algorithm is given in section 8 below.

If n is omitted some suitable default will be used, e.g., $n = 60$. It is not supposed that having formed set M, the user will necessarily inspect all its members. (DQ stands for *D*ocuments from *Q*uery.)

PDOCS n

where n is an integer. This prints out the first n documents in the list of set M which are not in S, adding them into S. So if S is null to begin with, *PDOCS* 2 prints out the first two documents from M, and *PDOCS* 3 then prints out the next three, leaving the first five documents of M in S. If n is omitted $n = 1$ is assumed.

TORELS *list*

This adds the given list of documents to R, the set of relevant documents.

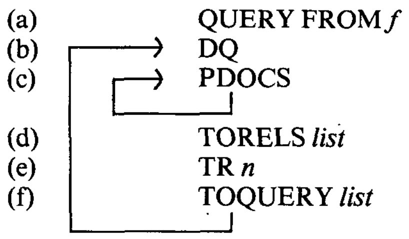
TR n

This finds and prints out the n terms with highest g-value from the set which index the various documents in R but which are not in Q. The terms are printed out by decreasing g-value. Again if n is omitted some default is used, e.g., $n = 10$.

TOQUERY *list*

This adds the given list of terms to the query.

These commands can be used to establish a simple relevance feedback cycle as follows:



(6.1)

This represents a flow diagram for a sequence of actions by the user. Initially sets R and S are both empty.

Step (a) establishes the query, and step (b) finds, for the query, the set M of best matching relevant documents. Step (c) repeats, and causes the documents to be inspected one at a time until the tail off in precision causes the user to stop and move on to step (d). Here he indicates which of the documents he has inspected are relevant (they are added to set R) and then in step (e) looks at closely associated terms. In step (f) he can add into the query terms which he believes will be useful in finding more relevant documents. (Steps (e) and (f) may be omitted.) The process may now continue from (b). A further DQ will establish a new M which will not include the documents he has already inspected (the ones in set S), and which should be improved by the extra query terms, if any, and the use of the set R in calculating the new w_i, and hence the new f-values. Some further commands are:

RELS *list*

This establishes a new environment in which R and S contain the documents in the given list.

DR *n*

This extracts from the terms which index the documents in R the n with the highest g-value, and sets Q, the query, equal to this set of terms. The command DQ m is then performed, to put in M the top ranking m documents for the query. m will be some suitably chosen constant, e.g., $m = 15$.

With these commands searching can be done without the user needing to specify a query. For example:

$$\begin{tikzcd}[row sep=small]
\text{(a)} & \text{RELS } \textit{list} \\
\text{(b)} & \text{DR} \\
\text{(c)} & \text{PDOCS} \arrow[loop left] \\
\text{(d)} & \text{TORELS } \textit{list} \arrow[uu, bend left=60]
\end{tikzcd} \qquad (6.2)$$

Step (a) makes the given list of documents relevant. The user, we may suppose, is interested in other documents which deal with similar subjects. Step (b) sets up M, and step (c) prints out successive documents from M for as long as it is repeated. In step (d) the user declares which of these documents is relevant, and then the process repeats from (b) with an improved Q and w_i values.

Finally there are the commands:

TM *n*

This is the same as TR n, but the top m documents of set M are used instead of the set R. This is handy when no, or very few, relevant documents have been found. Again m will be some suitably chosen number.

TQ *n*

This finds the n terms most closely associated with the terms in Q. It effectively works by performing the DQ command and then the TM command, and is useful as a first stage if the user feels some dissatisfaction with his initial query formulation.

7. EXAMPLES OF THE USE OF CUPID

These various procedures will now be illustrated by a couple of examples using a document collection which describes part of the library of the Museum Documentation Association. Essentially the library is concerned with all aspects of museum documentation, and especially documentation techniques which involve the use of computers. The collection was originally put into machine readable form for processing by GOS, and keywords were incorporated for the generation of keyword indexes for manual use. It must be emphasized that it was not originally supposed that the data would be accessed through any interactive IR system, although in fact it was only half a day's work to set it up for use by CUPID.

The basic statistics of this collection are:

number of documents	1758
number of terms	3122
average indexing	8 terms per document

This is of course rather a small sample, but adequate for purposes of illustration. The terms were derived from title and keyword lists. When the data were set up for use by CUPID they were not fully edited or complete, which explains the occasional spelling slips and absence of keyword lists in some of the example records below.

The first example illustrates the use of strategy (5.1). The user is interested in finding guides to the use of zoological and botanical nomenclature in identifying and naming biological specimens. He therefore inputs the query

Guides to zoological and botanical nomenclature

The command QUERY sets up a query, *Q*, with four stemmed terms as follows:

333:	botan
1161:	guid
1861:	nomenclatur
2906:	zoolog

Command DQ is now applied, and then PDOCS 2, which retrieves as the two top ranking documents:

761 Doc MDA 510.
Hancock, E. G. and Morgan, P. J. (eds.). 1980. A survay of zoological and botanical material in museums and other institutions in Great Britain. Biology Curators Group Report No. 1.: Biology Curators Group.

876 Doc MDA 2311.
International Commission on Zoological Nomenclature. 1964. International Code of Zoological Nomenclature adopted by the XV International Congress of Zoology. London: International Trust for Zoological Nomenclature.

keywords: ICZN, nomenclature, taxonomy, zoology, classified identification.

Of these, document 876 is judged relevant, and 761 non-relevant. PDOCS 2 again gives the next two documents:

804 Doc MDA 2298.
Heywood, V. H. (ed.). 1968. Modern method in plant taxonomy. Botanical Society of the British Isles. Conference Report No. 10. London: Academic Press. (see Brennan, McNeill and Stearn).

keywords: biology, taxonomy, nomenclature.

667 Doc MDA 2269.

Gotch, A. F. 1979. Mammals — their Latin names explained. A guide to animal classification.: Blandford Press. ISBN 0-7137-0939-1.

price £5.95.

keywords: nomenclature, taxonomy, zoology, classified identification.

Both these documents are judged to be relevant. A further PDOCS 2 gives

1793 Doc MDA 1296.
Williams, D. B. 1975. The use of E.D.P. in zoological collections.

in Brennan, Ross and Williams. 177.

keywords: zoology, computing system.

1091 Doc MDA 772.
Mello, J. F. The use of the SELGEM system in support of systematics.

in Brennan, J. P. M., Ross, R. and Williams, J. T. (eds.). 1975. Computers in botanical collections. London: Plenum Press. 125–138.

This time both documents are judged to be non-relevant. Further use of PDOCS is now suspended, and the three relevant documents are marked as relevant by the command

```
TORELS 876 804 667
```

and the command TR is now applied to get hold of terms which relate closely to the query terms. TR gives

```
2637:  taxonomi
2907:  zoologi
 476:  classifi
1261:  identif
2884:  xv
 961:  explain
1503:  latin
1257:  iczn
```

These terms are supposedly in decreasing order of closeness of association with the terms of the query. Term 2907, 'zoologi', turns up here because of a weakness in the stemming algorithm, which stems 'zoology' and 'zoological' to 'zoologi' and 'zoolog' respectively. It might be thought that there is a good case here for adjusting the stemming algorithm, but it will always be the case that the stemming can never be perfect, and so query expansion can be used as means of putting into the query certain variant forms of a word. The user decides to add the first three terms from this list into the query, which he does by the command

```
TOQUERY 2637 2907 476
```

and then he repeats DQ. PDOCS 2 then gives

918 Doc MDA 2339.
Jeffrey, C. 1977. Biological nomenclature. 2nd edition. London: Edward Arnold.

keywords: botany, zoology, systematics, nomenclature, taxonomy, biology, terminology control, classification system, classification, natural science, glossary, bibliography, classified identification, ICZN, ICBN, ICNB, IAPT, type, data categories, conventions, classified identification authority, Systematics Association.

1025 Doc MDA 2350.
Lincoln, R. G. and Sheals, J. G. 1979. Invertebrate animals. Collection and preservation. London: British Museum Natural History.
ISBN 0-521-29677-3 (paper).

price £3.50 (paper).

keywords: zoology, invertebrate, collection method, preservation method, classified identification, labelling, nomenclature, BMNH.

The first document, 918, is at once judged to be relevant. It will be seen that this document is indexed by all three of the terms added into the query in the previous stage. Obviously the relevance feedback cycle with possible query expansion could be repeated a second time, but our suspicion is that for a document collection as small as this it is probably not useful. As it is, the further command PDOCS 10 produces a list of ten related, but not especially relevant, further documents.

The weights, w_i, attached to the terms of the query frequently change as the query is reused in a manner which reflects one's intuitive feeling about their relative importance. Thus the weights for the two applications of DQ are as follows:

term	1st	2nd
botan	5.1	4.7
guid	3.7	3.2
nomenclatur	4.9	7.1
zoolog	6.2	6.0
classifi	—	5.0
taxonomi	—	6.4
zoologi	—	5.1

Thus 'nomenclatur' increases from 4.9 to 7.1, becoming the highest weighted term, while 'guid' decreases from 3.7 to 3.2, and is the lowest weighted term.

The second example illustrates strategy (5.2). The user is interested in finding documents about the use of computers, especially micros and minis, in handling archaeological data. His starting point is document 675, which is:

675 Doc MDA 2273.
Graham, I. 1981. Microcomputers for archaeological excavation recording. Intelligent computer terminals for archaeological site

recording. Report to the British Library . . . on project number SI/G/216. Final report . . . BLRDR 5600. London: Institute of Archaeology.

keywords: archaeology, stratigraphy, archaeological, BL, microcomputer, archaeological archive, data storage, excavation report, site material, recording medium.

RELS 675 adds this document to the set *R* of relevant documents. *DR* followed by PDOCS 2 retrieves as the two top ranking documents the following:

674 Doc MDA 456.
Graham, I. 1976. Intelligent terminals for excavation recording. Computer Applications in Archaeology 48-52.

1169 Doc MDA 2357.
Museum of London. Department of Urban Archaeology. 1980. Site manual. Part I: the written record. London: Museum of London.

keywords: archaeology, LDMoL. DUA, archaeological archive, excavation, stratigraphy, archaeological, documentation system.

The user judges 674 only to be relevant. PDOCS 2 then gives:

1243 Doc MDA 2361.
Nyukska, J. P. 1980. Biodeterioration and biostability of library materials. Restaurator 4 (1) 71-77. Aslib Inf. 1754/81.

keywords: conservation, books, libraries, museums, recording medium, storage, conservation, biodeterioration.

1399 Doc MDA 1023.
Roper, M. 1978. PROSPEC-SA: Pilot project. The development of PROSPEC for wider use in providing guides to record offices. Final Report, covering the period March 1977 to September 1978, to the British Library Research and Development on Project Number S1/G/217.: (?).

which are clearly off the track. TORELS 674 now makes 674 a relevant document, and DR is repeated, followed by PDOCS 7. The following documents are now retrieved:

426 Doc MDA 2213.
Council for British Archaeology. 1978. Computer retrieval of archaeological information. CBA Day School — 28th September 1978.: unpublished typescript.

keywords: CBA, MDA, archaeology, computing system, information retrieval, SMR, excavation recording, microcomputer, bibliographic system, catalogue, architecture, building.

215 Doc MDA 133.
Boddington, A. 1978. The excavation record: Part 1. Stratification Northamptonshire County Council. Archaeological Occasional Paper No. 1. Northampton: Northamptonshire County Council.

keywords: archaeology, excavation, Northamptonshire, stratigraphy, archaeological, documentation system.

924 Doc MDA 637.
Johnson, I. 1980. Computer recording of excavation data from Hunter-gatherer sites. Computer applications in archaeology 8–16.

742 Doc MDA 2292.
Hackmann, W. D. 1973. The evaluation of a museum communication format. Part I. Collection of input data. Final report for the period 1970–September 1972 and extension November 1972–December 1972. Report to OSTI on Project SI/56/07. OSTI Report No. 5154. Oxford: University of Oxford (for the Museum of the History of Science).

keywords: OSTI, OXFHS, IRGMA, science museum, technology museum, scientific instrument card, recording medium, documentation system, CGDS, MCF, MDS, scientific instrument.

782 Doc MDA 526.
Hector, E. J. 1977. An examination of cataloguing and indexing procedures in a number of fine art collections in British museums, and an evaluation of the usefulness of the A5 IRGMA fine art card. [MA in Librarianship thesis]. Sheffield: Sheffield University.

keywords: fine art, catalogue, documentation system, IRGMA, fine art card, progress, art gallery, pictorial representation, visual arts, prints, drawings, museums, information retrieval, enquiries, staff, museum documentation, recording medium, indexing system, GLAHM, ScM, BLKMG, BOLMG, KIRMG, KIMMG.

296 Doc MDA 184.
Buckland, P. and Wilcock, J. D. 1973. Remote terminals for on-site recording. Computer Applications in Archaeology 1973.

252 Doc MDA 161.
British Columbia. Provincial Museum and Heritage Conservation Branch. n.d. Guide to the B.C. Archaeological Site Inventory Form. British Columbia: Provincial Museum.

keywords: archaeology, Canada, British Columbia, archaeological archive, archaeology record centre, recording medium.

426, 924 and 296 are judged relevant, and are added to the relevance set using *TORELS. DR* is applied for the third time, and *PDOCS* 10 gives a list of ten further documents, out of which three are most probably relevant.

It is very instructive to look at the improvement in the query during this retrieval run. The query, Q, is constructed from terms which index documents in R each time DR is applied. Since there were three applications, there were three successive versions of Q, as follows:

1st	2nd	3rd
5600	termin	record
sig216	intellig	excav
blrdr	excav	termin
termin	record	comput
intellig	5600	intellig
microcomput	sig216	microcomput
stratigraphi	blrdr	site
final	microcomput	archaeolog
number	stratigraphi	archaeologi
medium	final	data
materi	number	smr
excav	medium	sig216
bl	materi	remot
site	bl	onsit
storag	site	huntergather

The terms are arranged in this list in 'best' to 'worst' order. In the third version of the query, the last five terms, namely 'smr', 'sig216', 'remot', 'onsit' and 'huntergather', are all single occurrence terms and index documents already in R. This means that they have no effect whatever on the retrieval process, so that the list of terms in the last column effectively stops at 'data'.

8. STRUCTURE OF CUPID

CUPID contains a resident library of utilities: a freespace management system, a primitive command language decoder, a number of miscellaneous procedures for constructing and manipulating in-core data structures (basically structures consisting of sets of short vectors or tuples), and various straightforward procedures for reading the direct access files. It also contains a collection of modules which are loaded and run in turn as a sequence of CUPID commands gets obeyed. The total amount of code involved is so small however (just over 2000 lines of source code at present) that it should be possible to have the entire system core resident, even on a 64K byte micro.

As CUPID performs its task of finding relevant documents, it will require information in the direct access files. Thus to print documents it will require strings of text from *Dd*; to compute the r_i, and hence the w_i, in (4.4) it will require the vectors from *DT* corresponding to the documents in set R; to print out the text of a term given its term number it will require a text string from *Tt*, and so on. Since CUPID is essentially iterative, repeating its basic cycle with improved w_i and query terms, it seems very likely that it will require the same entries from the direct access files more than once in the course of a run. For this reason many of these entries, once they have been read, are held in-core in a so-called *pool*. Prior to reading an entry from secondary storage, the system first checks to see whether it is in the pool, in which case the access to secondary storage is unnecessary. The pool also contains the sets R, S and M of relevant, seen and best matching documents. As the retrieval run continues the pool grows in size, and may need to be cut down if it overfills the primary store. (In practice this has never happened, and the code to deal with this case remains to be written!)

Given this basic structure, the various CUPID operations become quite simple. We will consider only the operation DQ. If the query is

$$Q = (T_1, T_2, \ldots T_k)$$

the various $D(T_i)$, which can be represented as

$$\begin{aligned} D(T_1) &= (D_{11}, D_{12}, \ldots) \\ D(T_2) &= (D_{21}, D_{22}, \ldots) \\ &\ldots \\ D(T_k) &= (D_{k1}, D_{k2}, \ldots) \end{aligned}$$

are opened as k streams for reading. A set Z is formed consisting of k 2-tuples. Each 2-tuple has the form (D_{ij},i), i.e., it consists of a document number D_{ij} and the number i of the stream from which it was read, and gives the last document to have been read from the ith stream. With the usual notion of tuple ordering, we can talk of the smallest 2-tuple of set Z.

To find the value $f(D_{ij})$ for each distinct D_{ij} we do the following:

(a) Remove (D_{ij},i), the smallest member of Z, from Z.
(b) Add (D_{ij+1},i) to Z if stream i is not exhausted.
(c) Set w equal to w_i.
(d) Remove (D_{kl},k), the smallest member of Z, from Z.
(e) If $D_{ij} = D_{kl}$ set w to $w + w_k$ and go back to (d), otherwise:
(f) w is now $f(D_{ij})$ and (D_{kl},k) will act as the next (D_{ij},i) for step (a).

This process gives rise to another series of 2-tuples of the form $(f(D_{ij}), D_{ij})$, or more simply $(f(D), D)$, and from them the set M may be constructed. M is initially empty, and has a maximum size, given by n. The procedure is as follows:

(a) If D is in set S discard it, otherwise:
(b) If the size of M is less than n, add $(f(D), D)$ to M, otherwise:
(c) (Size of M equals n.) If $(f(D), D)$ is greater than the smallest tuple in M, let it replace that tuple, otherwise discard D.

This must be repeated for each tuple $(f(D), D)$.

In the computation of the w_i, n_i, the length of the vector $D(T_i)$, is kept in the index file *TD*; N, the size of the document collection, is kept at the base of the index of file *DT*; r_i, the number of documents in R indexed by T_i, is found by reading $T(D_i)$ into primary storage for each D_i in R and simply counting along these term vectors; R, the size of set R, is simply kept as a statistic with the set. Note that it is assumed that the vectors $T(D_i)$ are sufficiently small to fit into primary storage. No such assumption may be made about the vectors $D(T_i)$.

This is a simple method for forming set M, and it is not always optimal. Suppose for example that Q is

$$Q = (T_1, T_2)$$

where T_1 is a common term with $n_1 = 10000$ and T_2 is a rare term with $n_2 = 10$. Suppose also that $w_2 > w_1$. A cheaper way of constructing M both in terms of cpu time and the number of accesses to secondary storage might be to read in the vector

$$D(T_2) = (D_1, D_2, \ldots D_{10})$$

and from this read in the term vectors $T(D_1)$, $T(D_2)$, . . . $T(D_{10})$, and find which of D_1, . . . D_{10} are indexed by T_1. M consists of those documents of D (T_2) indexed by T_1, followed by those documents of D (T_2) not indexed by T_1, and then followed by any other documents indexed by T_1 only.

Nevertheless it seems wise to stick to the simple strategy in constructing M, or something similar. To find an optimal strategy which handles the general case would seem to be difficult, and it is not clear that in practice the savings would ever be very great.

The choice of data structure to represent the various sets will depend on how they are being used. So for example while M is being constructed the most efficient representation is a heap, and once it has been constructed a linear vector is more convenient. But these sets never become very large, and it is probably a mistake to worry too much about efficiency here. A linear vector for all of these structures is probably adequate, although in practice we have used tree structures and linear vectors.

9. EMBEDDING CUPID IN A RELATIONAL DATABASE

Many aspects of CUPID have not been properly developed as it stands at the moment. The data is essentially static, so that if new documents are added to the collection, the process of setting up the direct access files must be repeated almost from scratch. The storage management is fairly primitive. So for example in evaluating DQ, the merge of the k streams of the D (T_i) requires space for k blocks of the file DT in core, and there is no provision for the case when k is so large that the blocks cannot be accommodated. Again no attempt is made to keep a pool of recently or frequently used blocks in core. The CUPID command language is primitive and none too user friendly, although the Dublin implementation improves on the Cambridge one here.

An attractive way of solving all these problems is to try and implement CUPID in a DBMS. The DBMS, if suitably powerful, could at once provide a suitable filing mechanism for the data with provision for dynamic updating, could provide good store management and, conceivably, a friendly user interface. It could also give the user access to the usual 'fact retrieval' facilities of a DBMS, which could be a useful complement to the IR methods offered by CUPID. Among DBMSs the relational approach would seem to be the most appropriate. The ease with which so many of the operations surrounding CUPID can be expressed in relational terms will by now have become apparent, and in fact almost all the operations performed by CUPID can be expressed in a relational algebra with a few natural extensions.

We will give an example of this, again with the DQ command which constructs the set M from the query. Using the notation developed in section 2, suppose relation V has the form $[T,w]$ and consists of the list of query terms T_i with their associated weights w_i. W is formed by joining V with TD,

$$W[T,w,D] := V[T,w] *(1)\, TD\,[T,D] \tag{9.1}$$

X is a simple projection of W,

$$X[D,w,T] := W[T,w,D] \rightarrow 3,2,1 \tag{9.2}$$

$TOT\ (R,n)$ is a special function which totals the numbers in the nth column of relation R over constant values in columns 1 to $n-1$, while columns $n+1$ upwards are discarded. Then

$$Y[D,w] := TOT(X[D,w,T], 2) \tag{9.3}$$

creates as Y a set of documents with their associated f-values in column 2. These can be ranked, going from lowest to highest f-value, by the projection

$$Z[w,D] := Y[D,w] \rightarrow 2,1 \tag{9.4}$$

and then if M has maximum size n, the last n tuples of Z give the set M. It may be supposed that the last n tuples can be extracted from a relation by a special function $LAST$,

$$M[w,D] := LAST(n, Z[w,D]) \tag{9.5}$$

This may be a very reasonable approach, but there could be problems of efficiency here. If the document collection is very large and some of the query terms are very common, relation W will be large. The sort of W in (9.2) could then be expensive. Relation Y should be much smaller than relation W, but again the sort in (9.4) could be expensive. If we think of the DBMS generating M directly from V, then it may find short cuts which make it unnecessary to generate all the relations W to Z. Nonetheless, it is not easy to imagine that it could from an expression of the form

$$M := LAST(n, TOT(((V{*}(1)\ TD) \rightarrow 3,2,1), 2) \rightarrow 2,1)$$

which combines (9.1) to (9.5), produce, as a means of evaluating M, a process equivalent to the one given in section 8.

But other approaches may be open to us. Suppose we begin with a series of relations $V1$, $V2$ . . . Vk, where each Vi has the form $[T,w]$ and contains the single 2-tuple $[T_i,w_i]$. If Wi is defined by

$$Wi\,[T,w,D] := Vi\,[T,w] *(1)\, TD\,[T,D] \tag{9.6}$$

then Wi consists of the elements of the vector $D\,(T_i)$ each prefixed by the fixed pair (T_i,w_i), in other words:

$$(T_i,\ w_i,\ D_{i1})$$
$$(T_i,\ w_i,\ D_{i2})$$
$$\ldots$$

If for each i, Xi is formed by the projection

$$Xi\,[D,w,T] := Wi\,[T,w,D] \rightarrow 3,2,1 \qquad (9.7)$$

and then X is formed from the Xi by a union of the Xi,

$$X\,[D,w,T] := X1\,[D,w,T] + \ldots + Xk\,[D,w,T] \qquad (9.8)$$

('+' indicating set union) then the X of (9.8) is the same as the X of (9.2). But operations (9.6) to (9.8) may be much faster than (9.1) and (9.2), since the projection in (9.7) does not involve any sorting of the tuples in Wi. The database might be able to detect that no sorting needs to be done, since it is possible to deduce that columns 1 and 2 of Wi must be constant from the fact that Vi has a cardinality of one, or if it does invoke a sort, this may run very fast since it effectively has no work to do. Alternatively it may be possible in the relational algebra to tell the system in the expression (9.7) that no sorting is required.

Finally it may be possible to construct M directly from X by a rather specialized function which works like the algorithm given in section 8:

$$M\,[w,D] := BEST\,(n, X\,[D,w,T])$$

The discussion has centered here around the formation of the set M since this is in fact the trickiest operation to deal with. The other operations, such as evaluating the parameters r_i from the set R, or ranking the terms which index the documents in R by decreasing g-value, turn out to be much more straightforward. I believe that we can conclude that a system like CUPID could be implemented in a suitably powerful relational DBMS with a relational algebra query language which can be extended by providing such special functions as *TOT*(. . .), *LAST*(. . .), *BEST*(. . .) and so on. This is an interesting result since possible tie-ups between IR work and the more recent developments in the DBMS area are too often dismissed on the grounds that DBMSs are concerned with fact retrieval only.

10. CONCLUSIONS

In a conventional IR system, queries are expressed in terms of Boolean logic (which the user is required to learn), and employs query terms which need to be chosen very carefully, often out of predefined lists and sometimes from a manually constructed thesaurus. The user will, in the latter case, need to know how to find his way around the thesaurus in order to examine and extract terms. The retrieved documents will usually be presented to the user as an unranked set, and some skill needs to be exercised to retrieve sets of a manageable size. In a system like CUPID, queries can be formulated in natural language, and the whole apparatus of Boolean logic and predefined lists of query terms is bypassed. This makes CUPID quite easy to use. It is not very difficult to implement, and could be useful as a practical IR system, as a teaching aid in the IR field, and as a vehicle for testing out IR ideas in the probabilistic model, particularly those involving user interaction. As a practical realization of a theoretical IR model, it will I hope help narrow the gap between IR theory and practice which currently exists.

ACKNOWLEDGEMENTS

My thanks are due to the Museum Documentation Association for their permission to use the machine readable version of their library catalogue, and for their help in setting it up for use by CUPID. I must also thank Professor van Rijsbergen of University College, Dublin for many valuable discussions and suggestions.

REFERENCES

Date, C. J. (1981) *An Introduction to Database Systems.* Third edition. Massachusetts: Addison Wesley.

King, T. J. (1979) *The design of a relational database management system for historical records.* Ph.D. Thesis. University of Cambridge.

Lennon, M., Peirce, D. S., Tarry, B. D. and Willett, P. (1981) An evaluation of some conflation algorithms for information retrieval. *Journal of Information Science 3* (4), 177–183.

Porter, M. F. (1980a) *GOS Reference Manual.* Duxford, Cambridgeshire: Museum Documentation Association.

Porter, M. F. (1980b) *How to use GOS.* Duxford, Cambridgeshire: Museum Documentation Association.

Porter, M. F. (1980c) *Description of the internal structure of GOS.* Duxford, Cambridgeshire: Museum Documentation Association.

Porter, M. F. (1980d) An algorithm for suffix stripping. *Program 14* (3), 130–137.

Robertson, S. E. and Sparck Jones, K. (1976) Relevance weighting of search terms. *Journal of the American Society for Information Science 27*, 129–146.

Van Rijsbergen, C. J. (1980) *Information Retrieval.* Second edition. London: Butterworths.