Some Considerations for Implementing the SMART Information Retrieval System Under UNIX

Edward A. Fox*

83-560
September 1983

*Department of Computer Science
Cornell University
Ithaca, New York 14853

now at:
Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24061

This work was supported in part by the National Science Foundation, under grant IST-81-08696.

TABLE OF CONTENTS

SOME CONSIDERATIONS FOR IMPLEMENTING THE SMART INFORMATION RETRIEVAL SYSTEM UNDER UNIX

Edward A. Fox*

Abstract

Since the early 1960's the SMART project has tested out new ideas in information science aimed at fully automatic document retrieval. Beginning in 1980 development of an enhanced and generalized version of SMART has progressed at Cornell. The current implementation is in the C language and runs under the UNIX operating system on a VAX 11/780 computer.

The history of SMART is outlined. Considerations that led to the current design are described. Since SMART now allows multiple concept types to be manipulated in connection with an extended vector representation, storage and processing issues are discussed, including use of INGRES relations. Clustering algorithms are presented and run parameters are given for document clustering and subsequent clustered searching. SMART experiments (e.g. with p-norm queries, or probabilistic methods) can be compared using the evaluation package. The S statistical package can be applied to performing other special analysis and descriptive tasks. Finally, to illustrate the usefulness of these facilities, an outline is given of current SMART activities and of future plans.

*Department of Computer Science, Cornell University, Ithaca, NY 14853; now at Dept. of Computer Science, Virginia Tech, Blacksburg, VA 24061. This work was supported in part by the National Science Foundation, under grant IST-81-08696.

1. Introduction

Development of the SMART system has proceeded since 1961 [Salton 1980]. This author began implementing SMART under the UNIX[1] operating system in 1980 on Cornell University Department of Computer Science's computers. Preliminary design notions were discussed in [Fox 1981]. This report brings the status of SMART up to date, describing the running system as used for the experiments detailed in [Fox 1983b].

Section 2 concerns the history of SMART. Early versions are mentioned; for more elaboration see [Salton 1971a, 1975, 1980]. The current UNIX version is also introduced.

The design of SMART is covered in section 3. Except for clustering and evaluation, all of the basic package is dealt with. Beginning with points gleaned from the IBM SMART system [SMART Project 1974], reasoning is given of how tools available under UNIX could be applied to retrieval tasks. Design principles are then listed, and the capabilities desired for the system are enunciated. Issues of data storage are considered, including use of the INGRES relational database package. With data requirements taken care of, program modularization is discussed. Searching, ranking, retrieval, and indexing are each separately considered.

Section 4 focuses on the formation and manipulation of extended vectors. A brief summary is given of the model introduced in [Fox 1983b], which has been tested using two new experimental collections [Fox 1983a]. A general procedure is given for constructing similar document collections, followed by a description of implementation issues as they relate to SMART software. Two types of manipulation of these extended vectors, clustering and clustered searching, are then considered.

The SMART evaluation package is discussed next. Of particular interest may be the integration of special software with the general purpose S statistical package. Justification is also given for the

[1]UNIX is a trademark of Bell Laboratories.

various algorithms and tests incorporated.

This report concludes with a sketch of recent studies being made with the SMART software, and with mention of ideas for future projects. The interested reader is advised to view the latter purely as illustrating some of the potential applications for SMART in upcoming years.

2. History of Implementations

2.1. Early Versions

In the late 1950's, automatic text analysis began to be seriously studied. In 1961, the SMART (System for Mechanical Analysis and Retrieval of Text) project began, with the object of designing a fully automatic document retrieval system and of testing out new ideas in information science.

In SMART, an automatic indexing component constructs stored representations of documents. Then a search component selects those documents which seem most similar to a user's query. Finally, an evaluation component allows rating of batches of results, using relevance judgments obtained from the user population, thereby facilitating comparisons between various indexing and search schemes [Salton 1980].

SMART was originally intended for operation in a time-sharing environment. In 1964 a preliminary design suited for such operation was suggested [Rocchio 1964] and implementation work began for a version to run under M.I.T.'s CTSS [Cane 1964]. In 1966, Lesk concluded that fully automatic documentation centers were feasible with current technology [Lesk 1966]. In 1970, a detailed design for an IBM 360 based conversational system was prepared [Williamson & Williamson 1970]. Later, a partial implementation for the IBM 370 TSO environment was undertaken.

The most complete early version of SMART was developed for batch operation on the IBM 370 at Cornell, and was completed around 1974 [SMART Project 1974]. The last major project using that system at Cornell was completed in 1980, and involved indexing, clustering, searching, and evaluation runs to show how SMART methods could be beneficially applied to indexing of business correspondence [Nodtvedt 1980].

The IBM SMART system had been programmed in FORTRAN IV with some portions in assembly language. Auxiliary programs, such as for clustering, were coded in a variety of other languages, especially in PL/I. Since running the system resulted in expensive computer charges, and since modifying or extending the generally undocumented monolithic system seemed inadvisable, it was decided to implement a generalized version of SMART using more modern methods and less expensive hardware.

2.2. UNIX Implementation

By 1980, several components of the SMART system had been reprogrammed in the C language [Kernighan & Ritchie 1978] to run under the UNIX operating system [Ritchie & Thompson 1974] on a Digital Equipment Corporation PDP[2] 11/60 computer. Later that year the software was transferred to a VAX[3] 11/780, and work since then has been on that hardware. There are now plans to also port the programs to a Motorola 68000 series based microcomputer running UNIX.

A number of proposals and tools have been considered to aid in the development effort. On the VAX, a collection of routines implementing the operators of the relational algebra [Date 1982] were programmed to allow occasional manipulation of relations; many of the smaller data files were stored as relations to aid in standardization. To generalize that approach, the idea of relational processing of abstract data types was advocated for retrieval operations [Fox 1981]. Though such a development scheme was theoretically appealing, there was not enough time or programming support to carry the project to completion in such a manner.

---

[2]PDP is a trademark of Digital Equipment Corporation.

[3]VAX is a trademark of Digital Equipment Corporation.

In 1981, an early version of the INGRES relational database management system [Stonebraker et.al. 1976] was obtained for use at Cornell. In 1982, the S statistical package [Becker & Chambers 1981] was also installed on the VAX. INGRES served as the backbone for data storage and manipulation in all situations where efficiency was not of crucial importance, and S became a handy tool for exploratory data analysis and for statistical calculations. The histograms, scatter plots, and much of the statistical reporting in [Fox 1983a, 1983b] were all done using S.

Though various such tools speeded up and supplemented the retrieval experimentation, the main part of SMART should be implementable on any UNIX system with sufficient storage and adequate memory management. Indeed, a version of SMART, used for a number of standard operations when efficiency is essential, requires neither INGRES nor S. One possible plan for the future is to have retrieval experiments carried out on a dedicated microcomputer that runs UNIX. Then the statistical evaluation and the comparison of results among runs could be done on a VAX which supports INGRES and S.

3. Design

3.1. IBM SMART System

The IBM SMART system, completed in the mid 1970's by a group of students and programmers at Cornell, was designed so as to be able to carry out the necessary processing without requiring a great deal of memory or machine resources [Williamson et al. 1971]. Further, it was programmed in FORTRAN, since members of the team were proficient in that language, even though the language standard then current lacked good tools for structured programming, text processing, or manipulation of files on secondary storage units.

SMART was executed as a single program, which brought in appropriate procedures as overlays when needed. Communication between various subprograms was through data files on secondary storage, or, more often, through variables held in a globally accessible common area of high speed memory. Unfortunately, to save space, portions of that common area were reused for varying purposes, making the data flow and interconnections exceedingly difficult to follow.

In 1981 an undergraduate student studied these SMART programs in order to prepare an overview of the system. The task was rather difficult and confusing, and so it was decided that further analysis of that software package would not be encouraged. However, the general structure deduced, part of which is shown in Figure 1, did provide useful guidance for other development efforts.

As can be seen in Figure 1, an executive routine leads to data initialization and then calls in, one by one, other procedures required to carry out the batch job submitted. Some of the data provided specifies values of parameters required by the processing routines; the remaining parameters are given appropriate settings by default. In any case, the list of variables needing specification was useful information in helping plan the description of modules for a subsequent implementation.

3.2. Use of UNIX

In 1980, it was decided that future research studies by this author would make use of the experimental facilities provided by the National Science Foundation for the Department of Computer Science at Cornell, since there were no charges for time or storage space, and since available development tools were easier to work with than those on the IBM system. The UNIX[4] operating system, in particular, encouraged interactive program development and testing of new ideas, using quick to develop prototype implementations; it was also very effective at performing the necessary text processing operations.

[4]UNIX is a trademark of Bell Laboratories.

Figure 1: Structure of IBM 370 SMART System

Executive Routine – calls initialization routine
Data Set Initialization – start or restart runs, and call other routines
- Text Input Routines
  - Store Input
  - Convert Text to Vectors
  - Perform Partial Syntactic Analysis
  - Recode Vectors for New Dictionary
- Vector Input Routines
  - Load in or Modify a Vector Collection
  - Read in Results of Previous Searches
  - List out a Vector Collection
- Search Routines
  - Modify Queries for Feedback Runs
  - Specify Which Documents should be Correlated
  - Correlate Documents against Query
  - Sort on Correlation and Assign Ranks
  - Prepare Results for Printing
- Clustering Routines
  - Define Newly Specified Cluster
  - Cluster via Dattola's Algorithm
  - Cluster via Rocchio's Algorithm
- Averaging Routines
  - Compute Various Averages
  - Print Output and Graphs
  - Run and Print Statistical Tests

For example, to index a document, one program could take a typical text input file and return a list of single words. A second could remove "stop" words, while a third could replace words with their stems. Other routines could sort lists, or number the lines of a file. By taking the output of one run and "piping" it into another program, two or more processes could execute virtually simultaneously, avoiding the need for intermediary files to be actually materialized in their entirety.

With the UNIX tools provided and a moderate number of new ones, a simple system for indexing documents was devised. To make it easier for others to use, the steps were recorded in a file that could thereafter be executed as a "shell script;" UNIX allows procedures to be written in a system command language or shell and those executable files can later be invoked as if they were a single program. Eventually, to provide capabilities needed to handle more complex document formats, and to allow more efficient operation as required for indexing thousands of documents, several sections of the indexing script were redone as single programs. This methodology of connecting available tools to develop prototypes that test new ideas, and of subsequently adapting and tuning those early versions for more efficient operation, is very much in accord with the practice of stepwise refinement [Wirth 1971].

3.3. Design Principles

To formalize these general notions, a number of design principles were explicitly stated to guide further planning:

(1) There should be a network of programs rather than a single large one.

(2) Communication between programs should be through data files.

(3) Communication between procedures and functions should be through argument lists and return values.

(4) Lists of default settings of parameters should be stored in text form in data files, to facilitate making changes.

(5) Different shell scripts should be developed to call the various programs automatically, for all standard tasks.

(6) The same program (or at least many of the same procedures) should be usable, with minor extensions perhaps, for experimental studies and for "production" retrieval runs. Thus, separate versions should be avoided as much as possible, and one should be able to run large batch experiments, interactively carry out a test search, or try out a new type of similarity function, all with

the standard system.

(7) Old SMART data files should be converted into new forms that might be required for indexing or other additional processing.

(8) Emphasis should be on getting new methods to work, or reimplementing old SMART components, as expeditiously as possible. All available tools should be exploited, including INGRES and S, and coding from scratch should only be done as a last resort.

(9) Tuning should be done as needed. First, sections of shell scripts should be replaced by specialized, more efficient programs to accomplish the same function. Then, algorithms and data structures should be revised to reduce processing and data manipulation costs.

3.4. Retrieval Capabilities

The actual retrieval software was also planned so as to provide a number of capabilities in a single coherent system:

(1) The ranking of retrieved documents was to be done separately from the actual search.

(2) Search methods should include sequential and clustered techniques first and manipulation of inverted files later.

(3) Vector and Boolean systems should be easily modelled.

(4) Probabilistic methods should be incorporated.

(5) A number of similarity functions should be included, such as:

a- cosine correlation
b- vector inner product
c- Boolean expression evaluation
d- p-norm expression [Salton, Fox & Wu 1982] interpretation
e- probabilistic weighting where terms are assumed independent,
where term dependencies follow a tree structure, or where

the BLE scheme ([Duda & Hart 1973] and [Yu, Luk & Siu 1979]) with singles, pairs, and triples of terms are used.

(6) As in the SIRE system ([McGill et al. 1976], [McGill & Noreault 1977], and [Noreault, Koll & McGill 1977]), it should be possible to rank the documents retrieved by a Boolean search according to some other similarity function.

(7) Ranking of documents should be possible for any type of search method, for both initial and feedback searches.

(8) Ranking for feedback runs should be possible using any one of several evaluation methods [Chang, Cirillo & Razon 1971]:

a- Residual collection
b- Test and control
c- Full freezing
d- Partial freezing

(9) Evaluation results should be stored for later printing and analysis. Printout options should include:

a- Recall-precision results for each query
b- Recall-precision averages for a batch of queries
c- Other statistics for each query or query collection
d- Statistical test results comparing the effectiveness of several retrieval trials.

(10) Statistical comparisons should at least include ([Siegel 1956], [Snedecor & Cochran 1980]):

a- T test
b- Sign test
c- Wilcoxon signed rank test.

(11) Clustering runs should allow gathering of information about search efficiency.

(12) An interactive version should provide reasonable responsiveness for:

a- Submission of p-norm, Boolean, or natural language query forms
b- Clustered or sequential searches
c- Display of portions of text from selected top-ranked documents
d- Improvement of current query by feedback processing
e- Subsequent searching, and repetition of steps (a)-(d).

(13) Document and query representations should allow use of multiple concept types, and composite similarity functions should be specifiable.

(14) Indexing should allow different processing for the various portions of incoming records, to account for formatting peculiarities and to enable separation into multiple concept types. Normalization of names (e.g., use last name followed by first initial) and dates should be automatic in appropriate places.

(15) Words should be able to be grouped, as in phrases appearing in keyword lists, or to be stemmed, or to have final "s" removed.

(16) A standard stop word list should be used as the default choice, but users should be allowed to substitute other lists instead.

(17) Automatic identification of statistical phrases (groups of words that co-occur often enough, appear close together in the same sentence, and each have proper collection wide frequency) should be possible. To allow this, positional information on each word in the collection must be available.

(18) Auxiliary programs should be able to process data used by the retrieval system. These should include routines for:

a- Clustering
b- Construction of feedback queries
c- Parsing and conversion from infix Boolean or p-norm expressions
to prefix forms and later to a fast internal representation.

In accord with these principles, the UNIX SMART system was programmed in stages over about a three year period. In the following sections, details of the current design are given. First, underlying data handling issues are described and then the various main programs are outlined.

3.5. Data Storage Schemes

Information retrieval calls for storing and accessing large amounts of data. Though specialized structures like compact vectors or inverted files are probably still the most appropriate ones to use for searching, there are many intermediate results and requirements for processing of smaller data groupings where relations can be effectively used. Furthermore, as relational systems become more efficient, with new hardware mechanisms and better algorithms available, and as text retrieval systems are generalized to handle structured as well as textual information, relations may be utilized even further.

In an earlier study [Fox 1981], relational processing was proposed for all aspects of retrieval work. For a practical implementation of SMART with existing software tools, however, several different data representations were employed.

First, standard UNIX test files were selected for a wide variety of purposes. The original form of document and query collections were provided that way. Parameters for complicated runs, such as clustering, were kept in this form so as to be easily edited. Finally, the output of runs almost always included textual output, for ease of interpretation.

Second, specialized files were employed. Compact forms of vectors, whether of queries, documents, or cluster centroids, were recorded as lists of

<integer, real number>

pairs in binary format to save space and speed up processing. Pointers to text files, giving the starting position and field lengths for each collection document, were stored in a fast access format to enable quick presentation of retrieved documents to on-line searchers. A final example is that co-occurrence data for probabilistic retrieval methods was also organized in a specialized format.

A third scheme for handling retrieval data was that of storing relations as text files. A good example is during the indexing phase, where UNIX programs can pipe results from one process to the next and thus save on intermediate storage. Starting with a stream of data organized as tuples, each a

<document number, string containing a word>

pair, eventually a file containing tuples like

<document number, concept type, concept number, document frequency>

is produced.

These textual relations are very useful, since each attribute can be of variable length, and since processing textual forms is exceptionally easy under UNIX. One handy transformation is that of compressing or expanding between relation and list forms and vice versa. Thus, a document collection of many

<document number, concept number>

pairs could be compressed into N groupings of form

<document number, list of concept numbers, 0 to end list>

where there are N distinct documents. Later, the groupings could be stored as compact vectors, or could be expanded back into first normal form relations.

The fourth and final format of data storage employed is that of relations stored in a database of the INGRES relational system. One database was created for each document collection. Routines or procedures were generally available to convert INGRES relations to text file type relations. Some of the relations stored in INGRES are shown in Figures 2 through 4.

3.6. INGRES Relations

Much about a system of computer programs can be understood when the detailed form of the data being manipulated is known. Hence, a description of the relations utilized in the SMART search and ranking system should be of particular value. These relations are of special interest since they are simpler to specify than are other secondary storage files – one need only know the purpose of the relation and details about the attributes which make it up.

Figure 2 covers the three relations employed for p-norm queries[5]. Utilizing these illustrates compromising between efficiency and ease of representation. When Boolean or p-norm queries are parsed, they are placed into an internal structure which is a tree form of the p-norm expression.

Figure 2: INGRES Relations Used for P-Norm Queries

1) Tree description: details of parse tree of query representation
   - no_cons – number of concepts in the query
   - no_nodes – number of nodes in the tree
   - bool – whether the query is interpreted as strict Boolean
   - root – node number of the root of the tree

2) Concepts: details about each concept in the query
   - qid – query identifier
   - con – concept number
   - ctype – concept type

3) Nodes: details about each operator or concept node in tree
   - type – "concept," or else which operator (AND,NOT,OR)
   - c – if type is "concept," then concept number
   - ctype – if type is "concept," then concept type
   - p – if type is "operator," then p-value attached
   - wt – relative weight attached to this node
   - child – number of the node which is its first child
   - sibl – number of the node which is its first sibling

---

[5]P-norm queries were first defined in [Wu 1981]. Experimental validation of their usefulness is given in [Salton, Fox & Wu 1982], [Salton, Buckley & Fox 1983], [Salton, Fox, Buckley & Voorhees 1983], and in great detail in [Fox 1983b].

Storage is done by recording essential data: about each query tree, about the leaves or concepts of the trees, and about the set of nodes in the tree. Modification of the trees is typically accomplished by reading the tree back into an internal structure, making revisions (such as is required for Boolean feedback) and storing the new tree instead of the old. Alternatively, the relational form can be directly changed.

To carry out a search, the query is loaded from the relations back into internal form, and then is rapidly evaluated for each document. The internal structure allows fast interpretation of the query and simplifies revisions; the relational storage scheme is slow to load and store, but makes accessing of the data much simpler to specify and eases the workload when structural modifications are required.

Figure 3 is about the four relations involved in searching and ranking. There are two key outputs of the programs: a top-ranked set, and ranking information about all relevant documents. For interactive searches, the user asks for, say, the ten best documents. For batch experiments with relevance feedback, the top ten to twenty documents may be selected for presentation to the user. When recall-precision charts or graphs are desired, it is crucial to have the rank of each relevant document so precision at each recall level can be calculated and averaged over all queries in the set.

For cluster searches it is useful to have information about how much of the document and centroid files have had to be examined in order to present the user with the desired number of documents. And, for query collections that are in relational form, the obvious format for storing relations is used.

Figure 4 deals with the input information required to commence an experimental run. The first relation connects to all other relations, so that with the "run_name" specified, one has the key required to find out about the document and query collections to use, and about other parameters which must be known. "sim_type" selects the desired similarity function, while "dwt" and "qwt" identify the chosen concept weighting schemes. When multiple concept types are present, their number is given by

Figure 3: INGRES Relations Used for Search & Ranking

1) Top-ranked set: identifies the n most similar documents to query
   - iter – feedback iteration step now in effect
   - qid – identifier of query being processed
   - sim – similarity of document to the query
   - rank – rank of the document in retrieved set where $1 \leq$ rank $\leq n$
   - rel – whether this document is judged relevant to query
2) Relevant ranks: ranks for all relevant documents, for each search
   - iter – feedback iteration step now in effect
   - qid – identifier of query being processed
   - did – relevant document identifier
   - rank – rank of this document after retrieval where ties are randomly broken
   - sim – similarity of document to the query
3) Cluster search efficiency: counts of vectors retrieved
   - run_name – description of this experimental run
   - iter – feedback iteration number
   - qid – identifier of query being processed
   - cent_total – total number of centroids examined
   - cent_ll – number of low level centroids examined
   - doc_total – total number of documents examined
   - doc_ret – number of documents actually presented to user
4) Query vectors: relational form of vector queries
   - qid – query identifier
   - con – concept number
   - ctype – concept type
   - wt – weight for current concept

Figure 4: INGRES Relations Required to Specify Runs

1) Parameters for search run

run_name - name or title used to refer to this search run
da_title - title of document collection selected
qa_title - title of query collection selected
sim_type - what kind of similarity function to use
dwt - what kind of document concept weighting to use
qwt - what kind of query concept weighting to use
p-value - if given, *p* to overide that on all operators
thes - name of thesaurus to use (not yet implemented)
iter - feedback iteration number
no_pairs - number of pairs to include in BLE expansion
no_triples - number of triples to include in BLE expansion
fdbk_type - what feedback evaluation (e.g., freezing) to use
no_tr - number of documents to remember in top ranked set for feedback presentation (e.g., 10)
sim0_opt - should relevant documents with 0 similarity be randomly assigned ranks or get worst possible
rel_ranks - name of relation to record ranks of relevants
top_ranks - name of relation to record information about top-ranked documents for feedback presentation
no_wanted - number desired from cluster search
numb_rels - estimated number relevant per query, for relative recall evaluations
use_nrels - indication to use "numb_rels", since complete relevance data is unavailable

2) Document collection details

a_title - title of collection
no_docs - number of documents
no_ctypes - number of different concepts used in indexing
tree - UNIX tree path name for directory of data files
centroid_fa- name of fast access storage file for centroids
centroid_cw- name of concept-weight storage file for centroids
fast_access- name of fast access document vector file
con_wt - name of concept-weight document vector file

3) Query collection details

a_title - title of collection
no_q - number of queries
q_type - query type: e.g., p-norm or vector
q_rels - name of relation with relevance information

Figure 4 continued: INGRES Relations Required to Specify Runs

4) Concept type multipliers for combined retrieval formula

run_name	– name of run (matches that in relation 1 above)
ctype	– concept type
ct_mul	– multiplier or coefficient for this concept type
sim_type	– similarity function to use for concept type
dwt	– document weighting scheme for this concept type
qwt	– query weighting scheme for this concept type

5) Relevance information for a document/query collection pair

qid	– query identifier
did	– document identifier for a document relevant to this query

---

the "no_ctypes" attribute of the document collection relation, and the weighting coefficient for each is present, along with other attributes, in the fourth relation shown. The third relation is given to describe the query collection, and refers, through the "q_rels" attribute, to an appropriate instantiation of the fifth relation given, which contains user supplied relevance decisions.

The first relation in Figure 4 includes a number of special parameters to facilitate experimentation. Repeated runs can be easily made by simply changing "pvalue", or by varying "no_wanted" to broaden a cluster search through calling for more retrieved documents. Similarly, BLE[6] runs could be varied by changing "no_pairs" and "no_triples". For feedback runs, the number to retrieve and present to the user is given as "no_tr", and the iteration number by "iter", while "fdbk_type" details which of the four feedback evaluation schemes is selected. Normally "sim_opt" calls for random ranking of nonretrieved relevants among the nonretrieved set and "numb_rels" is not used since complete relevance information is provided. Entries "top_ranks" and "rel_ranks" refer to the output relations

[6]The Bahadur Lazarsfield expansion (BLE) allows a binary valued function to be represented by a terminating series of probabilistic expressions of single items, pairs, triples, etc. [Duda & Hart 1973]. A truncated form can be used to compute term weights and other values for information retrieval purposes [Yu, Luk & Siu 1979].

defined at the beginning of Figure 3.

Collection specification relations include information about special format files containing appropriate document and query representations. Since either vector or p-norm queries are allowable, the attributes required to specify all the differing file descriptions are here omitted from those listed for the query collection relation. For document collections, however, the five names that can be specified are each listed. The first one, "tree", gives the name of the directory containing the four remaining files, two for cluster centroids and two for document vectors. Two files are required for each since the actual concept-weight pairs are stored in a compressed binary file while pointers and subvector length data needed for fast access to the vectors are provided in a second file.

Figures 2 through 4 thus describe the currently used INGRES relations. Relations were incorporated in an early version of the clustering program; that version was quickly implemented and worked well but was so slow that tuning required all relational processing be replaced by special access procedures. The result was a least two orders of magnitude faster, since many simple file references required relational command parsing and interpretation when INGRES was utilized.

To facilitate SMART processing on systems that lack INGRES, a version of the retrieval program is available that does not require any of the relations described. Rather, special text files are formatted so as to have all the necessary data described above. Some of the processing, such as feedback ranking, which is fairly involved, can not yet be handled by this faster version, but for tasks that can be accomplished with it, the speed benefits more than offset the difficulty of having to specify a desired run in a more complicated manner.

3.7. Program for Search and Ranking

The basic retrieval system had to handle feedback ranking and a number of similarity methods as well as having documents represented with multiple concept types. In order to minimize the future maintenance required, a design with a number of modules was chosen. Thus, separate modules were coded to process centroid and document files used for cluster searching, or to manipulate p-norm queries.

Figure 5 is an outline of the various routines in the search and ranking system. There are six main groupings of procedures. The basic components constitute the first group, which includes the main program. The overall search algorithms are present as well, along with the standard vector similarity functions. Selection of top ranked documents and determination of the ranks of all relevant documents is also carried out here.

The second grouping includes general utility routines. Vector retrieval from other INGRES or fast-access forms is allowed. For ranking, a randomization routine is provided. Finally, general sorting and binary search procedures are present.

The third grouping is split into two sets of clustering related modules to hide underlying data representations from higher level routines. One set handles the nodes of a cluster tree. The other manipulates the heap of centroids and documents that have been retrieved but are being kept to allow subsequent broadening of an initial depth first cluster search.

The remaining three groupings deal with similarity computation for p-norm, tree dependence, and BLE approaches. P-norm or Boolean queries are stored in three relations and so the data must be loaded into a fast internal form prior to examination of documents. The actual similarity computation for each document can then proceed, with the necessary steps of handling concept-weight pairs, taking p^{th} roots and raising to the p^{th} power, evaluating the special AND and OR formula, and finally deter-

Figure 5: Routines for Search and Ranking

Basic Components
- Main program
- Sequential and clustered searching
- Vector similarity functions: inner product, cosine
- Top ranked handling
- Ranking of relevant documents

Utility Routines
- Retrieving vectors in INGRES form
- Retrieving vectors in fast-access form
- Sorting
- Binary searching
- Randomizing ranks

Clustered File Handling
- Accessing nodes of cluster tree
- Manipulating of heap for broadening a depth first search

P-norm Similarity Computation
- Loading tree into internal form
- Evaluating a (concept-weight pair) term in a clause
- Computing powers and roots
- Evaluating an AND or OR clause
- Computing overall similarity for a document

Tree Dependence Similarity Computation
- Setting up query expansion desired
- Computing similarity for a document

BLE Similarity Computation
- Setting up query expansion desired
- Obtaining co-occurrence data
- Computing coefficients for single terms, and for pairs and triples
- Computing similarity for a document

mining the overall similarity for the document.

Tree dependence and BLE similarity computations are similar to each other in that the proper setup is required, co-occurrence information must be gathered, and then probabilities can be estimated.

Similarity of a document can then be determined according to the appropriate model.

3.8. Retrieval Related Programs

To supplement the basic search and retrieval program, a number of separate routines were required. The indexing and evaluation packages are described in the next subsection and in section 5, respectively. But components for clustering, initial or feedback query construction, and on-line interactive use are discussed immediately below.

The clustering process is described in section 4 below; it is a hierarchical tree building process [Williamson 1974] that requires $O(n \log n)$ steps to cluster n documents. A roughly balanced tree with all leaves (i.e. documents) at the same level results. When nodes become too large, they are split – an $O(n^2)$ approach to find a good division of the n children of the node helps the end result have good quality, but since the maximum node size is bounded (e.g., $n<20$), the overall complexity is not increased. A single program performs the clustering, and other supplemental programs give various printouts, or compress the tree for faster searching and reduced storage size.

Preparation of initial feedback queries is a more difficult task. Each query representation – p-norm, vector, or probabilistic scheme – requires separate processing to utilize given feedback data or to initially construct queries.

For p-norm queries, there are the initial routines for parsing Syracuse Diatom[7] searches or prefix forms of user questions, and obtaining p-norm relations. Given query vectors, one routine will construct frequency range p-norm queries based on a suitable parameter file; see [Fox 1983b Chapter 4] for the algorithm. Another routine will construct automatic Boolean or p-norm queries from vectors and

[7]Diatom is a retrieval system using inverted files to process Boolean queries developed at the Syracuse University School of Information Studies. See the discussion about creating the INSPEC Boolean query collection in [Fox 1983b Appendix C].

parameters, using the DNF scheme.

This DNF construction program can also make use of a p-norm query collection, along with feedback information, to produce Boolean or p-norm feedback queries. A good deal of coding by several people was involved in the various versions of this program; the basic approach and algorithms have been given in [Fox 1983b Chapters 4 and 5], [Salton, Buckley & Fox 1983], and [Salton, Fox, Buckley & Voorhees 1983].

Feedback processing of vector queries required a number of other programs. Note that by using all relevance information, one can run searches that give retrospective or upper bound performance – some of the comparisons in [Fox 1983b Chapters 3-5] include those cases. Otherwise, feedback will normally include information from a short initial search, as was the case with extended vector term relevance runs described in [Fox 1983b Chapter 8].

Feedback programs for vector queries were of several types. The one mentioned in the later part of [Fox 1983b Chapter 8] used the term relevance scheme, and produced a new vector query that could then be run using the inner product similarity function. The other two versions produced queries in tree dependence and BLE forms, and so the feedback search required use of those two similarity computation techniques.

As can be seen, a goodly number of different programs are involved in query preparation, searching, and feedback. However, while this situation is fine for experimentation, it is inappropriate for on-line searching. Hence, a repackaging of the entire system was undertaken to facilitate on-line operation.

Rather than have separate programs for that purpose, the routines already described had some minor changes made and were simply invoked in the proper order by a new UNIX shell script. A few additional programs were also needed, to display portions of the original document texts when documents were retrieved, and to accept user feedback information. One version began with a p-norm query, while another started with a natural language statement. Figure 6 illustrates the processing for the later.

It is hoped that eventually the on-line system, now used primarily for experimentation or demonstrations, can be of practical use when applied to other collections, and can be employed to gather realistic queries and relevance information for use in future studies.

3.9. Indexing Package

The indexing component of SMART has evolved considerably since early versions of the system were devised. Initially, only collections previously indexed, such as the ADI and Medlars documents and queries, had to be processed. Later, for generality, a standard document organization was devised and a two step process was required. Any collection format would be preprocessed by a special routine or parser and converted into the standard one. Then the usual SMART indexing package could be applied to produce document vectors and other by-products.

Figure 6: On-Line SMART Operation

(1) Accept user supplied natural language text form of question.
(2) Index the text to produce a query vector.
(3) Run retrieval and ranking program to retrieve 10 documents (using clustered or sequential search).
(4) Show user the document numbers and similarities in descending order.
(5) Display desired portions of the text of documents retrieved, and request feedback information on each.
(6) Store relevance judgment information for future use.
(7) Construct feedback query if desired.
(8) If feedback is desired, return to step 3. Otherwise return to step 1 if more queries are available.

The basic approach outlined above in section 3.2 is followed in indexing, but a number of extensions were necessary. These can be seen when one examines the various phases required for indexing.

First, several preprocessors were called for. A simple, fast one was provided for single queries submitted to the on-line system. A basic one was provided to take previous SMART collections and make minor revisions and changes. Several types of electronic mail parsers were also devised. Eventually, a generalization of this procedure was suggested; it will be mentioned later in section 6.

Second, a collection in standard SMART format was processed. Figure 7 illustrates that structure, which is appropriate for query or document collections. In calling for indexing, weights could be attached to entries in the various sections, and sections could be ignored if desired. Concept types were assigned to basic entities in a section – times, dates, words, author names, keyword phrases, etc. Since different parsing techniques could be applied to each section, words or dates in several parts of the document could be distinguished and each given different concept types. Stemming options, stop word lists, and other choices to make could also vary depending on the characteristics of each section.

Third, the primary output from a standard collection was a textual relation identifying at least:

1) document number
2) concept type
3) textual entity – e.g., stem or author name or date

To facilitate formation of statistical phrases, to allow separate indexing of paragraphs or even sentences, and to enable use of metrical operations (e.g., "w" for within same sentence) – all ideas for future enhancements – it was also possible to have other information produced as output such as:

1) paragraph number in document
2) sentence number in paragraph
3) word number in sentence.

Note that for certain textual entities such as author names, a special output canonical form was adopted – e.g.,:

Figure 7: Standard SMART Indexing Structure

.I n (to identify the n^{th} document)
.T
title lines
.W
words in abstract, text, or message
.K
key words or key phrases
.A
author name lines
.B
bibliographic information like date
.C
categories or descriptors
.N
name or other descriptive comments
.FN (to terminate an entire collection)

<last-name>_<first-initial> (for example, Salton_G).

Stemming or final "s" removal was accomplished in this phase if requested, and stop words were removed as found.

Fourth, for document collections, the unique entries of each concept type were collected together, sorted, and a dictionary constructed. For dynamic collections it would be better to use hashing to add new terms to the dictionary as they occur, or to look up their concept type and number in case they have occurred before. Lacking that implementation, however, the indexing output need be further processed to replace each actual textual entity with the appropriate concept number in that concept type's dictionary.

Fifth, the raw data, including document number, concept type, and concept number at least, must be combined to produce vectors. Multiple occurrences of concepts in a document are noted, and the term frequencies or count of such repetitions is determined. Additional data, such as that from processing co-citations or other bibliographic information, is also combined with the result of previous indexing steps, and the end product, a vector with multiple subvectors, is stored in the appropriate internal form. Intermediate information, separately recorded for each concept type, allows compilation of collection frequency and thence inverse document frequency weights, so the final vectors may include "tf*idf" weighting (see [Fox 1983b Chapter 2]) appropriate for either p-norm or vector matching.

Finally, with the document collections processed, query collections can be handled in a similar fashion. The dictionary need only be read from, not added to, and document term weighting procedures are not relevant. The rest of the operations are applied in much the same fashion as document indexing. P-norm queries do not require special splitting into words, identification of concept types, or checking for stop words, but words must still be looked up in appropriate dictionaries. Eventually each query is placed in the correct form for searching.

4. Creation and Clustering of Extended Vectors

4.1. Background

In the vector space model as implemented by previous versions of SMART [Salton 1971a], each document is represented by a single vector, usually made up exclusively of terms. Though in some experiments (e.g., [Salton 1963, 1971b], [Michelson et al. 1971]) the vector had bibliographic information in addition to term concepts, there was no clear separation into subvectors as is proposed in [Fox 1983b].

Using additional factual information such as author name or perhaps publication date has precedent in the areas of database processing and in the operation of many common Boolean retrieval services. File management and database systems always relate a number of fields or attributes to a given real-world entity; an article must have a title, journal name, author name (or names), and date of publication. Table 1 shows a partial listing of fields used in the INSPEC database, provided as files 12 and 13 in the DIALOG system[8]

Database and Boolean systems commonly allow explicit references to each subfield or attribute. Queries typically include expressions with syntax such as

Table 1: Fields in INSPEC Database

Abbreviation	Field Name	Example
AB	Abstract	This is a very long abstract ...
DE	Descriptor	Personal Computing
ID	Identifier	Choice
TI	Title	Microprocessor Systems
AN	Abstract Number	A77087732
AU	Author	Runge, H.
BN	ISBN No.	0 9505171
CC	Class Code	A9260G
CL	Conference Loc.	Bucharest
CO	CODEN	Meruaz
CS	Corporate Source	Agard
CT	Conference Title	Acoustical ...
CY	Conference Year	1979
DT	Doc. Type	Journal Paper
JN	Journal Name	Acta Astron ...
LA	Language	Dutch
PA	Patent Assignee	British Leyland
PC	Patent Country	UK
PN	Patent No.	30
PU	Publisher	Academic
PY	Publication Year	1977
RN	Report No.	AD-168564
F	Subfile	C
TC	Treatment Code	Applic
UD	Update	8014

[8]Sheet from February 1981 DIALOG (trademark) brochure about INSPEC database.

$$\text{AUTHOR} = \text{Smith} \quad \text{or} \quad \text{Article.author} = \text{Jones}.$$

Range searches on publication date are often employed to restrict the size of retrieved sets, and are based on testing of the date field.

There is no straightforward way to accommodate these needs in the simple vector model – for example, one cannot distinguish a name appearing in the author field from the same name in the abstract unless special indexing techniques are employed to simulate the separation of concepts into different types.

Consequently, the extended vector model has been proposed and experimentally tested [Fox 1983b] using two collections in computer and information science [Fox 1983a]. The discussion below deals first with procedures for creating extended vectors (specifically, the bibliographic components) and for utilizing them inside the SMART system. The rest of this section then focuses on how extended vectors can be clustered and searched by new programs specially added to SMART for those purposes.

4.2. Constructing Bibliographic Submatrices

Now that the role of bibliographic information in enhancing document representations has been explained, it seems appropriate to describe algorithms for obtaining the requisite submatrices.

4.2.1. Input Data

Though it is conceivable that co-citation or other already processed data could be supplied initially, it seems more sensible to specify the processing steps required if only basic raw data is supplied.

In the case of bibliographic information, that data will be a set of tuples describing which documents are cited by others:

$$\text{CITED} = \{ (d_i, d_j) \mid d_i \rightarrow d_j \} \qquad \text{for } 1 \leq i,j \leq N. \tag{4-1}$$

The set captures the details of which of the N collection articles are cited by others; the arrow in (4-1) points from referring to cited article.

4.2.2. Initial Subvectors

To obtain the $\vec{\ln}$ subvectors, one begins by supplementing CITED with its transpose and the diagonal. Sorting puts the tuples in proper order, and subvectors are then easily built by identifying repeating groups instead of leaving the data in relational first normal form. Figure 8 shows the steps required.

Figure 8: Constructing $\vec{\ln}$ Subvectors

```
1. /* use CITED */
 ln := CITED

2. /* add transpose */
 For each (d_i, d_j) in CITED
     add (d_j, d_i) to ln

3. /* add diagonal */
 For k := 1 to N
     add (d_k, d_k) to ln

4. /* sort */
 Sort ln tuples on first and
 then second attributes.

5. /* make vectors */
 Construct vectors by converting from relational first
 normal form to vector form of repeating groups.
```

Forming the $\underline{bc}$ submatrix requires a bit more processing. Cited tuples are sorted on the second attribute and then coupled articles are identified. The result is also sorted and duplicates are counted. Finally $\underline{bc}$ is obtained. Figure 9 gives the details.

Figure 9: Constructing $\vec{bc}$ Subvectors

1. C := ϕ

2. Sort CITED on second attribute,
   i.e., d_j of tuple (d_i, d_j).
   Cited can then be partitioned into p groups,
   each with a distinct d_j.

3. For each of the p groups, do the following:
   F := $\{ (d_i \mid d_i$ is the first element of some tuple in the group $\}$
   X := $\{ (d_i, d_j) \mid d_i \in F,\ d_j \in F \}$
   Add entries in X to C.

4. Sort C on first and then second attributes.

5. Construct BC by counting the number of identical tuples of C,
   i.e., if there are n tuples (d_i, d_j) in C then have
   (d_i, d_j, n) in BC. Such a BC tuple designates that
   documents d_i and d_j have n units of coupling and
   so BC will be the relational form of $\underline{bc}$.

The $\underline{cc}$ submatrix is formed in a similar fashion, though sorting is done in reverse order, as can be seen in Figure 10.

Figure 10: Constructing $\vec{cc}$ Subvectors

1. C := ϕ

2. Sort CITED on first attribute,
   i.e., d_i of tuple (d_i, d_j).
   Cited can then be partitioned into p groups,
   each with a distinct d_i.

3. For each of the p groups, do the following:
   S := { $(d_j \mid d_j$ is the second element of some tuple in the group }
   X := { $(d_i, d_j) \mid d_i \in S, d_j \in S$ }
   Add entries in X to C.

4. Sort C on first and then second attributes.

5. Construct CC by counting the number of identical tuples of C,
   i.e., if there are n tuples (d_i, d_j) in C then have
   (d_i, d_j, n) in CC. Such a CC tuple designates that
   documents d_i and d_j are co-cited n times and
   so CC will be the relational form of $\underline{cc}$.

4.2.3. Updating Subvectors

Now that it is clear how bibliographically based subvectors can be initially constructed for a new collection, the subsequent issue of updating such subvectors as new articles are added to a collection must be considered. Clearly the bibliographic subvectors can be rebuilt from scratch, but for minor additions that is much too expensive a process to perform.

Two situations typically arise in connection with adding new bibliographic data. The first is when a new article is added, and references in its bibliography suddenly become available. The second situation often results from the first, in that when a new document is added, references to it from other documents already processed will finally be meaningful and can be considered.

Regarding the first situation, let d_a be the newly added document. Let A be the new tuples,

$$A = \{ (d_n, d_i) | \; d_n \rightarrow d_i \}.$$

To update ln, one adds A, the transpose of (the full matrix form of) A, and all the diagonal entries (d_a, d_a).

To update cc, one produces a set S of all documents referred to by d_a,

S := { d_j | d_j is the second element of some tuple in A }

and then identifies all possible pairs of entries in S,

X := { (d_i, d_j) | $d_i \in$S, $d_j \in$S }.

Given X, one adds 1 to the appropriate cell of cc for each entry in X – i.e., (d_i, d_j) in X calls for adding one to the value for row i and column j of cc, and if $i \neq j$, also adding 1 to the entry in row j and column i.

To update bc with the information available in set A about d_a, one must do a bit more elaborate processing. Any article d_j in the bibliography of d_a can be the cause of coupling d_a with some other article d_i. But to identify such instances one must know that $d_i \rightarrow d_j$. Unfortunately, it is not possible to tell whether $d_i \rightarrow d_j$ or $d_j \rightarrow d_i$ from any of the available submatrices. Hence one must keep in an accessible form the original CITED tuples. Assuming they are sorted on the second attribute, one can update bc with d_a information as shown in Figure 11.

Figure 11: Updating <u>*bc*</u> with Added Document d_a

```
For each document d_j referred to by d_a:
    For each document d_i such that (d_i, d_j) is in CITED:
        begin
          add 1 to the entry in row i, column a
          if n ≠ i, add 1 to the entry in row a, column i
        end
```

The second update situation arises from the fact that though already processed articles may refer to d_a, such data would have been ignored since d_a was not previously in the collection. Obviously, if such data was not ignored then no action would be required. In a dynamic environment, therefore, it is advisable to include all suitable documents in the various submatrices at the beginning to avoid having to perform substantial extra work later.

Nevertheless, it is possible to properly deal directly with this second case for newly added documents. To update <u>ln</u>, one adds all tuples (d_i, d_a) and (d_a, d_i), for $i \neq a$, as well as (d_a, d_a), based on occurrences in CITED.

To update <u>*cc*</u>, one constructs set MISSED containing all tuples in CITED of form (d_i, d_a). A new set CITED$'$ is formed from MISSED so that

$$\text{CITED}' = \{ (d_i, d_j) \mid (d_i, d_a) \in \text{MISSED} \}.$$

CITED$'$ is used instead of CITED in the algorithm of Figure 10, leading to a new relation CC$'$. Finally, the co-citation submatrix <u>*cc*</u> is updated by replacing entries with the new values found in corresponding tuples of CC$'$.

Regarding $\underline{bc}$, one presumes that since d_a never appeared in the output but was just used internally in the algorithm then all $\underline{bc}$ entries are indeed correct and the effect of d_a has already been clearly recorded.

4.2.4. Construction Algorithm Complexity

The various algorithms given above describe how bibliographic submatrices can be formed from the available raw data. That data is simply the set of reference information, CITED, which lists all articles cited by a given article. For a collection of N articles, CITED could conceivably be of size $O(N^2)$, but since the average number of references is fairly small, say on the order of 15, one would presume $|\text{CITED}| \leq 15 * \text{N}$.

Constructing the various submatrices involves several sorts, so the complexity is at least $O(N \log N)$ steps. In identifying instances of bibliographic coupling and co-citation, one may need to consider all possible pairs of elements in a particular group. Again, it is theoretically possible that some article is cited by $k * N$ other articles, for small k such as .01 (i.e., 1% of the articles), but it seems more reasonable to assume that some realistic fixed upper bound could be found, such as 10,000. Adopting such an assumption, therefore, each part of the various algorithms requires no more elaborate processing than the sorting phases, so the overall complexity is $O(N \log N)$.

One concludes that the necessary bibliographic submatrices can be easily formed from initial reference information, can be updated as new articles are added, and that the various processes can proceed without an extraordinary amount of computation.

4.3. Implementation in SMART

The multiple concept type vector approach calls for a number of generalizations to techniques used in simple vector processing systems. Points raised that may be of particular interest to database or retrieval system researchers will be highlighted.

To provide an overall framework to the discussion, notation and terminology from relational database theory will be employed. Some of the ideas developed here relate to ideas proposed earlier [Fox 1981].

A document collection C in the usual vector scheme can easily be described by the relation

$$C = \text{Collection (did, con, wt)} \tag{4-2}$$

where

did = the document identifier, i.e., index in range 1 ... N
con = concept number, i.e., concept's index in dictionary
wt = concept weight, i.e., binary or real valued weight
assigned to this concept in this document.

Vector approaches might sort on *did* and treat <con,wt> pairs as a repeating group to eliminate duplicate storage of *did* values; inverted file approaches would do the same type of space savings trick after sorting on *con*.

To shift to the multiple concept type scheme, one simply defines another attribute, "ctype" for "concept type", and enlarges the collection relation:

$$C = \text{Collection (did, ctype, con, wt)} \tag{4-3}$$

where

ctype = an integer code designating the type of concept
that con represents.

An example is when 0 is used for authors, 1 for terms, and 2 for date of publication.

The queries can also be defined with multiple concept types, as in

$$Q \;=\; \text{Queries (qid, ctype, con, wt)} \tag{4-4}$$

where a whole batch of queries may be considered together. Similarity computations require knowing when a match occurs; the condition is, for two concepts specified by ($\text{ctype}_1, \text{con}_1$) and ($\text{ctype}_2, \text{con}_2$), that

$$\text{ctype}_1 = \text{ctype}_2 \quad \text{and} \quad \text{con}_1 = \text{con}_2 . \tag{4-5}$$

An overall similarity computation between document i and query j might then be defined, assuming cosine is used on each of S subvectors, as

$$SIM(\vec{D}_i, \vec{Q}_j) \;=\; \sum_{ct=1}^{S} coeff_{ct} \frac{a_{ct}}{\sqrt{b_{ct} * c_{ct}}} \tag{4-6}$$

where, using pseudo SEQUEL [Chamberlin et al. 1976] statements:

```
a_ct = SUM ( collection.wt * queries.wt )
         WHERE collection.ctype = queries.ctype = ct
          AND collection.con   = queries.con
          AND collection.did   = i
          AND queries.qid      = j

b_ct = SUM ( (collection.wt)^2 )
         WHERE collection.did   = i
          AND collection.ctype = ct

c_ct = SUM ( (queries.wt)^2 )
         WHERE queries.qid     = j
          AND queries.ctype   = ct .
```

The reader is referred to [Fox 81] for further discussion on enhancing the relational model to better express retrieval tasks such as the above.

At various points in SMART processing, vectors are viewed as being present in relations such as (4-3) or (4-4). By simply specifying the appropriate ctype and concept, one can readily perform exact matches as well as general similarity computations. [Fox 1983b Chapter 8] discusses how p-norm methods can be applied in the context of extended vector feedback using such matching methods.

To actually process a user supplied query one must begin with the natural language form. Consequently, an explanation of indexing procedures is now in order. Indeed, the indexing process is one where particular attention is required for properly handling multiple concept types.

Certain concept types such as the bibliographically derived ones are processed separately and loaded directly into the Collection relation. Thus, given raw data such as that document 5734 refers to 3256 and 4681, one can add tuples as shown in Figure 12.

Figure 12: Collection Tuples Based on Knowing
$5734 \rightarrow 3256$, $5734 \rightarrow 4681$

did	ctype	con	wt
5734	ln	3256	1
5734	ln	4681	1
3256	ln	5734	1
4681	ln	5734	1
3256	cc	4681	1
4681	cc	3256	1

No dictionary is required for such concept types, but appropriate algorithms must be followed to initially construct and later update the proper tuples. Recall section 3 for further details on the relations and for discussion of SMART indexing of multiple concept type collections.

Other concept types, such as terms and authors, do require dictionaries. Viewed in relational notation, one has

Dict = Dictionary (ctype, con, text) (4-7) where "text" is the actual textual form of the word, name, date, or other. Typically, a document database as originally supplied by a vendor or other source will come with separate fields for each principal kind of data item, so indexing techniques need only tag the elements of each field with the appropriate ctype when the input is parsed. Certain concept types, such as terms, will imply that a particular stop word list and analysis procedure such as stemming should be applied prior to dictionary lookup or entry. Indeed, different handling of each concept type is possible so that names of authors can be suitably reformatted, dates can be put in a single standard form, and other normalizations can take place.

These methods have been applied to the task of processing electronic mail messages, where specially formulated parsing rules are applied to incoming files, and sender, recipient and other useful concept types are identified from the available context. Transformation is first made to a standard SMART collection input form, namely that of Figure 10. Many of the concept types lead eventually to a dictionary like (4-7) and a document vector collection like (4-3).

Figure 13: SMART Standard Collection Input Form

```
.I <did or qid>

.W
   words of text, for ctype tm

.A
   author lines

.K
   key words or phrases

.B
   bibliographic information like date

.C
   thesaurus category

.T
   title
      .
      .
      .

.FN <line that finishes the collection>
```

Document and natural language queries are both indexed according to the above system, so suitable vector collections are eventually obtained. P-norm queries are somewhat easier to process since by convention each match expression must include its concept type (or else the default type, say term, is assumed). Concept types with dictionary entries can then have query elements readily looked up while other types specify the exact concept numbers. The procedure is illustrated in Figure 14.

Figure 14: Example of P-Norm Query Processing with
Multiple Concept Types

1. User submits initial query

   author.Harper
   AND^{2}
   (probabilistic AND^{1} retrieval AND^{1} estimation)
   AND^{2}
   (date.1980 OR^{∞} date.1981)

2. System considers terms (ctype=0), authors (ctype=1), and dates (ctype=2) and interprets the query using <ctype,con> values such as

   <1,396>
   AND^{2}
   (<0,5738> AND^{1} <0,7325> AND^{1} <0,1528>)
   AND^{2}
   (<2,256> OR^{∞} <2,257>) .

3. Documents 5387, 5623, and 6542 are retrieved with highest similarity to the query.

4. User requests articles linked, bibliographically coupled, or co-cited with document 5387 in 1980 or 1981, where ctypes 3, 4, and 5 respectively designate those three types of information,

   (<3,5387> OR^{2} <4,5387> OR^{2} <5,5387>)
   AND^{3}
   (<2,256> OR^{∞} <2,257>) .

5. System responds with retrieval results, and the searching process terminates.

4.4. Clustering and Searching

Since queries are often made up only of terms, and certainly lack special bibliographic information such as co-citation linkages, there is no obvious direct way to utilize such data in a search. However, if searches retrieved groups instead of single documents and the groups were formed in part due to bibliographic connections then the value of extended vectors might be demonstrated. This approach was explored in [Fox 1983b Chapter 7].

Clustering is simply a process for creating groupings, and clustered search allows retrieval of groups in response to query submission. The subsections below provide a little background about clustering methods and then go into the details of clustering as incorporated into the current SMART system.

4.4.1. Background

The application of clustering methods to retrieval has been explored by a number of researchers. For an overview, refer to [Fox 1983b Chapter 7] and the references identified there. For the study described in [Fox 1983b] it was necessary to employ an efficient technique yielding reasonably good results, so a modified version of the Williamson [1974] algorithm was selected.

Williamson developed an effective and highly efficient clustering and search system employing heuristic methods to construct a cluster tree in time $O(n * \log n)$. Though lacking the theoretically pleasing qualities of the single link method his algorithm seemed to be stable and yield good results as attested by a number of experimental trials. The resulting tree is easily controlled by parameter settings to obtain a desired degree of overlap, a limitation of cluster size at each level of the tree, and a specific constraint on the branching ratios. Since the tree is built by adding documents one by one, collection growth is easily accommodated. Finally, when addition to a given node forces that node to exceed the allowable size then it is split apart only after computing the full similarity matrix for all of its children – a technique that enables reliable control and yields a good overall end-product. Williamson's algorithm served as the basis for the method employed to cluster extended-vectors as described below. With the aid of a Cornell graduate student, Robert Harper, the clustering program was implemented under UNIX. Clustered search procedures were then integrated into the rest of the SMART system.

4.4.2. Clustering with Bibliographic Data

The extended vector model was introduced whereby various types of bibliographic data were utilized to supplement the standard term vectors. One motivation for the model was the success of clustering studies considering bibliographic data instead of or in addition to terms. For more details see [Fox 1983b Chapter 7].

Apparently, better clustering can be expected when supplemental data is also considered. Hence it seems appropriate to consider algorithms for clustering that allow inclusion of the various components of an extended vector representation.

4.4.3. Algorithms

The algorithms in subsequent subsections are based in general on the work of Williamson as described in his Ph.D. thesis [1974]. The procedures were later adapted for experiments requiring control over the degree of overlap between low level clusters and for comparisons with other storage and retrieval schemes [Salton & Wu 1981].

In the current implementation, a number of further changes have been made. In converting the programs to run under UNIX a modular approach was employed and a much more comprehensible end product produced. Simultaneously the procedures were generalized to handle the requirements of clustering extended vectors.

Following the precedent of the original description by Williamson, the clustering and search algorithms will be separately discussed below even though many aspects of searching are integrated into the clustering phase. Where possible, redundant explanations will be omitted.

4.4.4. Clustering

The clustering algorithm produces a hierarchy where all N documents in a collection end up as leaves of a multilevel tree. Interior nodes are associated with cluster centroids which represent all the documents in the subtree below them. Viewed another way, a given centroid summarizes all the information contained in the children immediately below regardless of whether those are documents or other centroids.

Clustering proceeds by adding documents one by one starting with an initially empty tree. The addition process involves a search for the proper place to insert the new document and a subsequent adjustment of the tree to first include the new entry and secondly conform to the various constraints enforced during the build operation. In particular, adding a document may require a low level node or "twig" to be split and splitting may be recursively repeated all the way up to the root. This way the tree stays relatively balanced and all documents are the same distance from the root.

Now that the general procedure has been summarized the various parameters involved will be described.

4.4.4.1. Parameters

Table 2 gives the definitions of the general parameters involved in clustering.

Table 2: General Parameters for Clustering

Name	Description
ADDBOT	value in formula controlling overlap, namely (COVER-SUBTOP)/(SIZE+ADDBOT) > MAXCOV where COVER = coverage value based on NUMCOV scheme, and SIZE = number of members in cluster, so as to force dropping of larger trial clusters.
DEFBY DEFNEW DEFTOT	These three are definition test parameters; a doc. is accepted as a trial cluster center if its neighbors pass this after the concentration test – both check density using statistical measures. Concentration requires at least TSTTOT docs (including the cluster center) to have corr. over TSTBY standard deviations larger than the mean off-diagonal corr., and TSTNEW percent of these must not be in another trial cluster. Definition parameters are less strict and let in other less similar children – they "define" the cluster.
DIVIDE	max no. of kids before split is forced (e.g., 20)
INIFOV	"in if over", no. of stdv's above mean off-diag. corr. required for a doc. to be added to a twig.
MAXCOV	max coverage ratio allowed – clusters proposed are deleted when their elements are mostly covered by already existing clusters, in descending order of ADDBOT ratio.
MINSPLIT	min no. of kids before a split is attempted
MNLOS	min no. of kids required to form a centroid of loose docs., if they cannot be added anywhere (see INIFOV).
NUMCOV	true=>numerical overlap (compute COVER based on the number of overlap occurrences) false=>logical overlap (compute COVER based on the number of children having any overlap)
SUBTOP	overlap control value – see explanation of ADDBOT
TEST_ALL	should all documents be tested as trial cluster centers, or should only unassigned ones be tried
TESTBY TSTNEW TSTTOT	These three are concentration test parameters; a doc. is accepted as a trial cluster center if its neighbors pass this and then the definition test – see the DEFBY,DEFNEW,DEFTOT explanation above.

These control the splitting operation (parameters MINSPLIT, DIVIDE), the decision process for identifying candidate new clusters during splitting (TSTTOT, TESTBY, TSTNEW, DEFTOT, DEFBY, and DEFNEW), and the determination of whether a candidate cluster is not needed due to overlapping with others by too much (MAXCOV, NUMCOV, ADDBOT, and SUBTOP).

Table 3 describes parameters required by the extended vector approach.

Table 3: Combined Retrieval Parameters for Each Concept Type

similarity coefficient ≡ coefficient used for a given concept type before adding it to arrive at overall similarity, based on formula:

$$\text{combined similarity} = \sum_{\text{all types } t} coeff_t * sim_t$$

similarity computation method ≡ specification of function to compute similarity: cos correlation, inner product, normalized inner product (i.e., divided by sum of vector values)

weighting method ≡ use binary or real values

maximum subvector length ≡ length of this subvector that must not be exceeded; if it is, then low frequency values in the subvector are deleted to shorten it to within bounds

subvector deletion frequency: initial value and increment ≡ when subvector must be shortened, all entries below the initial value are deleted, and for subsequent deletions the increment is added to the cutoff previously used

---

The first three values described specify how the overall similarity between documents can be determined based on available subvectors – relative weighting, similarity function, and whether real valued weights are allowed. The last two parameters allow special processing when a centroid subvector gets too long and must be shortened to fit into available space. Given these various definitions, the actual clustering algorithm can be better understood.

4.4.4.2. Actual Procedures in Outline Form

Figures 15 through 22 describe in simple terms the various routines involved in clustering. The main procedure shown in Figure 15 obtains the requisite parameter values and repeatedly adds new documents to the current tree. In actuality there is also a parameter which indicates whether the tree is new and should be initialized as empty; other parameters specify which documents should be added if a partial clustering has already taken place. The procedure therefore allows clustering to proceed in steps and can even accept a list of document numbers to allow arbitrary control over the order of addition of new documents.

Figure 15: Main Procedure

```
BEGIN
        Initialize tree if initially empty;
        Get parameter settings from input and default file;
        WHILE (more documents to add)
                add(doc);
END;
```

Adding documents (Figure 16) simply involves finding the proper place to insert, attaching the incoming entry appropriately, and recursively splitting overly large nodes from that point up the tree as needed.

Figure 16: Add a Document

```
PROC add(doc);
        parent := find(doc):
        attach(parent,doc);
        split(parent);
END add;
```

Find (Figure 17) is essentially a depth first search, a simpler version of the search algorithm given in section 4.4.5. Starting at the root the most appropriate branch is chosen and followed downward until a twig is reached. Since each centroid passed will have the new document below it, such centroids have the new document appropriately added in. If a new centroid becomes overly large, unimportant or low frequency concepts are deleted until the total vector length is within bounds. Finally, the correct twig is identified and returned as the result of "find".

Figure 17: Find Correct Insertion Point for a Document

```
PROC find(doc);
        WHILE (current place is not a twig)
                BEGIN
                        select most similar child;
                        add vector for doc to centroid of
                           child and shorten if too long;
                END
        RETURN (current place);
END find;
```

Attaching a document (Figure 18) is straightforward but splitting (Figure 19) is more complex. In the generality of cases a split is unnecessary and so no action is needed. If a node has enough children to warrant trying to split, that will be investigated but no action is taken if the suggested splitting is not suitable. On the other hand if the node must be split the best splitting is identified and unconditionally accepted.

Figure 18: Attach document To Its New Parent

```
PROC attach(node, child);
        modify tree representation so child is
                attached to its parent node;
END attach;
```

Figure 19: Splitting a Node

```
PROC split(node);
        WHILE (node has too many children)
                BEGIN
                        div_cent(node,no_kids(node));
                        build new centroids based on
                                specification given
                                as div_cent result;
                        node := parent(node);
                END
END split;
```

Splitting is accomplished by procedures div_cent (Figure 20), cleave (Figure 21), and uncour (Figure 22). First a complete similarity matrix is formed based on the pairwise combined similarity values. "Cleave" then identifies a plausible clustering except that no limit on overlap is considered. "Uncour" compensates for that by first deleting clusters that exhibit too much overlap with remaining clusters, and secondly by assigning the others to a "garbage" or "orphan" cluster.

Figure 20: Dividing Up a Centroid by Clustering its Children

```
PROC div_cent(node, no_children);
        compute child x child similarity matrix;
        cleave(sim_mat, no_children);
END div_cent;
```

The splitting by "cleave" could employ single link or other methods but presently is based on the similarity distribution. By comparing the similarity of a pair of children to the average off diagonal similarity value and seeing if it is a sufficient number of standard deviations away from the mean, highly correlated pairs are identified. Candidate clusters which pass the concentration test are those formed by having enough highly correlated pairs in the proposed cluster. Once such a core has been identified other slightly less tightly bound children that can be added in are used to define the final candidate cluster and the second test can then be passed. For example, one might require two tightly connected nodes and five nodes connected by at least slightly more than the average correlation.

Figure 21: Splitting Up an Overly Large Cluster

```
PROC cleave(sim_mat, no_children);
        use sim_mat to compute statistics such as average and
                standard deviation of similarity values;
        FOR (each child)
                IF (passes concentration test)
                THEN BEGIN
                        attach other nodes that pass the
                                looser criterion test;
                        IF (passes definition test)
                        THEN accept cluster for output;
                END
        uncour();
END cleave;
```

"Uncour" repeatedly considers the remaining cluster that is most heavily covered by other clusters. If the overlap is too much, it is deleted. Eventually only clusters that pass all appropriate tests are accepted.

Figure 22: Limiting Cluster Overlap

```
PROC uncour();
        compute "cover" for each candidate cluster;
        WHILE (no. of clusters > 0)
                BEGIN
                        heavy_cover := most heavily covered cluster;
                        IF (heavy_cover isn't too covered)
                        THEN BREAK;
                        delete heavy_cover and correct
                                all cover counts appropriately;
                END
        collect any orphans;
        compress resulting description matrix;
END uncour;
```

The above description and the figures given should be adequate to give a general understanding of how the clustering process is carried out.

4.4.4.3. Linearized Cluster Tree

Salton and Wu [1981] demonstrated that if the cluster tree is appropriately stored and a prefetch scheme is adopted for secondary access then fairly efficient searching can take place. Therefore, a separate program was written to "linearize" the cluster tree.

Specifically the tree is traversed from left to right, one level at a time, starting at the top and proceeding down to the leaves. Thus the root is stored first, followed by all nodes below the root, followed by the level below, etc.

If one assumes that prefetch or read ahead buffering takes place by the secondary memory system then a much reduced number of page faults will be expected as compared to if a least recently used (LRU) page replacement policy is followed. Whenever the search procedure calls for examination of the children of the current node, centroids for all those documents will be stored sequentially and will probably be read in by one secondary storage access. Salton and Wu [1981] found this approach to be much more efficient than the LRU paging method.

4.4.5. Searching

Searching a clustered tree is very much like finding the correct place to add a new document – see Figure 17 and the attendant description. However, searching has a different objective; instead of finding the single twig where insertion should follow one would like to retrieve and rank documents so that all relevant documents, regardless of what cluster they appear in, are retrieved as soon as possible.

Figure 23 describes the general procedure. A heap is maintained so that productive but so far untraversed paths can be remembered and considered in due course. Heap entries are sorted based on their "effective similarity" value for subsequent searching so that the heap top always is the best docu-

---

Figure 23: Main Search Procedure

```
BEGIN
        add root to initially empty heap;
        no_retrieved := 0;
        WHILE (no_retrieved < desired_no_retrieved
                & heap is not empty)
                BEGIN
                        heap_top := pop(heap);
                        IF (heap_top is a document)
                        THEN BEGIN
                                increment no_retrieved;
                                save heap_top to rank for user;
                        END
                        ELSE add children of heap_top to heap;
                END
END
```

ment or centroid to examine. An adjustment is made in computing "effective similarityρq values of centroids; if a document and a centroid have the same similarity to a query, it is likely that the centroid is "hiding" a document even more similar to the query and so the effective similarity of the centroid is slightly increased. Another adjustment is made for documents in the same low level cluster since it is desirable that they be retrieved together. Specifically, following the cluster hypothesis that similar documents are probably similar to the same queries, the better half of the documents in the same cluster are assigned an effective heap similarity which is a weighted average of their actual similarity to the query and the similarity of the "best" document in the cluster. Thus, a cluster with documents having similarities

1.0, .5, .4, .3, .2, .1

have the similarities for the first three (i.e., those in the better half) adjusted by formula

$$0.6 * sim \quad + \quad 0.4 * sim_{best},$$

leading to new effective similarities

1.0, .7, .64, .3, .2, .1.

The result of all of these adjustments is that a broad search is carried out instead of a narrow depth first search. Further, most of the documents in a retrieved cluster are presented to the user fairly quickly though those with negligible query-document similarity will be bypassed till much later. All in all, precision should be fairly high as should recall though more centroids and documents than usual must be considered. By varying the formula for the two adjustments described above, some degree of control can be exercised over the balance between effectiveness and efficiency.

5. Evaluation

This section describes the SMART evaluation package. Background and methods are discussed, but the detailed algorithms are omitted since they have been described elsewhere. What is important here is to see which of a wide variety of techniques have been selected as being appropriate to current requirements.

5.1. Background

A great deal has been written about the evaluation of information retrieval work. Some of the relevant literature will be surveyed in this section.

5.1.1. Recall/Precision

Recall and precision are two measures that have been discussed for many years. Their use was encouraged by the findings of the Aslib-Cranfield projects. Lawani [1970] gives a good early summary of these studies which began in 1957 – some of the most significant early work in indexing and retrieval evaluation.

Assuming that it is known whether or not a document is relevant to a given query, and that each query will, in a certain situation, retrieve a certain number of documents, then recall and precision are defined, according to the scheme in Table 4, by equations (5-1) and (5-2).

Table 4: Contingency Table for Defining Recall, Precision

	Relevant	Non-relevant
Retrieved	a	b
Not Retrieved	c	d

$$\text{recall} = \frac{a}{a + c} \tag{5-1}$$

$$\text{precision} = \frac{a}{a + b} \tag{5-2}$$

Recall and precision have been used in many studies since Cranfield, and so they are reported by the current evaluation system along with other measures.

5.1.2. Summaries Available

Chapter 5, entitled "Retrieval Evaluation", of [Salton & McGill 1983] gives an up to date summary of the major issues of that subject. Included also is an excellent list of references for further study.

Additional general discussion is included in [Salton 1975]. Derivations and details of many formulas were published previously in [Salton 1968]. Another early explanation is given in [Salton & Lesk 1968]. Actual methods used in the IBM SMART system are described in [Salton 1971a]. Thorough coverage of the "E" measure is in [Van Rijsbergen 1979].

Statistical techniques in general are described in [Snedecor & Cochrane 1980]. An approach to data analysis, and discussion of algorithms that relate, is found in [Chambers 1977]. Finally, the "S" statistical package, which implements Chambers' ideas, and which has been used for [Fox 1983b], is explained in [Becker & Chambers 1981].

5.1.3. Key Issues

Before giving details of how evaluation has been carried out with the SMART system, it is appropriate to discuss the various perspectives and related trade offs possible. For example, the two measures, recall and precision, defined in equations (5-1) and (5-2), were viewed as very important by workers in the Cranfield projects. They can be used to analyze various elements of retrieval systems.

First, the method of indexing – e.g., whether multiple concept types are useful, or which weighting schemes are best – can be examined. Second, indexing provisions for improving recall and precision – e.g., adding in lexically related words or expanding queries through feedback – can be tested. Finally, the quality of searches can be measured – e.g., comparison of Boolean versus p-norm versus vector representations, or the contrast between manually and automatically formulated Boolean expressions.

Given that recall and precision measures are adopted, the next issue is how to summarize the results for a set of searches into a single recall-precision graph. Various possibilities exist for handling Boolean or vector retrieval results for each query. Interpolation methods are sometimes needed to give a continuous relationship. Averaging can be done from either user or systems perspectives.

Once single curves are obtained, the next issue is how to compare several curves to see which reflects the better performance. The t-test, sign test, and Wilcoxon signed rank test have all been suggested, and can be used, as long as one is aware of the assumptions which their use implies.

Rather than comparing curves, however, it would be more convenient to have a single value that characterized each curve. Many such measures have been proposed, and each has its adherents.

Choosing one is another crucial decision, though it is wisest to utilize such a measure for preliminary analysis and presentation purposes, and to use more complete information when making detailed comparisons.

Decisions made in regard to each of the above issues, as relates to evaluation results reported in [Fox 1983b], are explained in the next section.

5.2. Methods Utilized

Since the earliest partial version of SMART was made operational under UNIX, it has been possible to produce recall-precision tables and graphs to present the results of experimental runs. As a wider range of searches was made possible, the SMART evaluation package expanded into a more flexible and comprehensive set of routines. The current features are described below, along with some of the specifics of implementation.

5.2.1. Similarity Based Ranking

Recall and precision are defined in terms of the number of documents retrieved by a query. When the similarity between a query and document can be any one of many possible values, such as when the cosine or p-norm formula are utilized, then a recall-precision pair can be computed for any threshold on similarity.

From a SMART run, the results are available as tuples:

$$(\text{did, sim, rel, rank}) \tag{5-3}$$

where

did	means	document identifier
sim	is the	similarity between query and document

rel	tells	whether this document is relevant to query
rank	is the	rank of this document compared to others,

after query-document similarities have been determined. Based on such a relation, one could have a recall-precision pair for each rank position, assuming that retrieval could be controlled to stop at any setting of the number of top ranked documents desired.

In order to arrive at the relation shown in (5-3), several steps are called for. First, similarity values are computed. For clustered searches or strict Boolean cases, documents that are not retrieved are assumed to have a similarity of zero. Second, in computing ranks ties are broken by randomization. Thus, for Boolean queries relevant nonretrieved documents are assigned random positions among the set of nonretrieved. The final step, that of assigning ranks to relevant documents, is actually done by sorting the retrieved set on similarity and incrementing ranks of those relevants in a special updating pass through the tuples of retrieved documents.

5.2.2. Query Statistics

Given the relation in (5-3), one only needs some general additional information to compute a variety of statistics. First, precision at twenty one recall levels is determined. Basically, the precision and recall associated with every relevant document is found and precision at other recall levels is interpolated leftward [Salton & McGill 1983 page 167]. For example, if precision at recall .1 is .5 and at recall .25 is .4, then a precision of .5 is used for the open interval (0,.1] while .4 is used for (.1,.25]. According to this scheme, precision can be determined at recall levels 0, 0.05, 0.10, ..., 0.95, 1.0. Those values are retained for averaging with the results of other queries.

Figure 24 lists all the statistics filed away for each query of each experimental run. Eleven precision values are recorded while the other 10 are simply computed, used for averaging with other queries, but not filed away for each query. Normalized recall and precision, and log precision, are all computed and recorded as in earlier versions of the SMART system.

Figure 24: Query Statistics

Experimental test run name
Query identifier
Precision for recall levels:
0.0, 0.1, 0.2, 0.3, 0.4, 0.5
0.6, 0.7, 0.8, 0.9, 1.0
Normalized recall
Normalized precision
Log precision
Recall, precision for retrieved
set sizes: 10, 30
E values for all 6 cases, where
$\beta = 0.5, 1.0$, or 2.0
retrieved set size = 10 or 30

For compatibility with reported work by Van Rijsbergen and others, the E value is also given. For retrieved sets of size 10 and 30, recall and precision are determined. With those values, and for situations where recall and precision are assumed to have relative significance given by ratios 1:2, 1:1, and 2:1, the E measure is reported.

The main values considered useful, however, are the recall level averages. E values are relatively difficult to interpret, and there seems not to be great reduction in information if six different numbers must be reported to illustrate what performance has been achieved. Further, the basic values in determining E, the recall and precision for retrieved set sizes 10 and 30, can be rather deceiving if only 5 or 15 documents are retrieved by a Boolean query, or if a complete ranking of a thousand documents takes place via cosine similarity when 50 or more relevant documents exist.

In any case, the various statistics shown in Figure 24 are determined and saved in relation "q_stats_q" for later statistical comparisons, as discussed below.

5.2.3. User Oriented Averages

When a system is to be evaluated, there must be some way to combine the results from a number of queries to arrive at average precision values. The approach taken is to simply compute the mean of each separate statistic, averaging over the queries present. In other words, all the single query statistics of Figure 24, plus the other 10 precision values that were originally computed, are simply summarized by their means.

Williamson [1974] advocates a different type of averaging for cluster searches. However, if queries each have roughly the same number of relevant documents, and enough documents (e.g., 30) are retrieved by each search, his system oriented average scheme does not present enough advantage over the user oriented averaging to warrant adopting that type of presentation which is much harder to understand intuitively.

5.2.4. Single Value Measure

For comparison of many results, and for presentation of differences between schemes, it is very useful to have a single statistic that can capture the overall level of search effectiveness. The measure adopted is simply the average for precision values at recall levels 0.25, 0.50, and 0.75.

While this measure lacks the appeal of theoretically pleasing schemes such as the E measure, it captures a characteristic of query performance that is easily understood – how precise the identification of relevant documents is for an average type of search.

5.2.5. Statistical Comparison

Given results for each query in an experimental run, a comparison with another run can be made by considering values pairwise. That is, each statistic is available for two different cases, call them A and B, for each query.

Standard paired tests can therefore be applied, and for each measure, e.g., precision at recall 1.0 or log precision, it can be seen whether schemes A and B are different. Various tests can be applied, depending on what sort of assumptions can be made about the distribution of values being compared. The t-test is well known, but many feel that since results compared are probably not normally distributed then using the t-test is not justified. Furthermore, it is only sensitive to the magnitude of difference between different schemes, whereas in retrieval situations there may be a small actual difference which may nevertheless be uniformly maintained at all recall levels.

The sign test is certainly usable, since it is based on whether the difference between sets of values is almost always of the same sign. And the set of values considered is the difference in precision between two trials at a number of recall levels. However, the sign test is not very powerful, in the statistical sense, in that two distributions which most people would agree are distinct might not be shown to be significantly different by the sign test.

The Wilcoxon signed rank test is probably the best available for the retrieval system comparisons being made. It has good statistical power, and yet the assumptions made about distributions being tested are not unreasonable.

These three tests are described in earlier works about the SMART system [Salton 1971a, 1975]. The next section illustrates their use.

5.2.6. Sample Evaluation Output

Figures 25 through 29 are based on output from the evaluation package and illustrate the form and content of reports of data and statistical comparisons.

Figure 25: Example of Standard Recall-Precision Chart

Medlars – Probabilistic Retrieval Using Maximum Spanning Tree
Feedback after Initial Cosine Match, tf*idf weighting.

Results for the following cases are shown below:
1) Test case: Query Expansion Scheme 1.
2) Test case: Query Expansion Scheme 2.

recall level	precision for cases– 1	2
0.00	0.8095	0.7168
0.05	0.7917	0.7152
0.10	0.7301	0.6410
0.15	0.6331	0.5800
0.20	0.6046	0.5307
0.25	0.5552	0.5073
0.30	0.4925	0.4674
0.35	0.4585	0.4022
0.40	0.3961	0.3592
0.45	0.3282	0.3299
0.50	0.3234	0.3161
0.55	0.2647	0.2426
0.60	0.2579	0.2234
0.65	0.2353	0.1635
0.70	0.1609	0.1183
0.75	0.1202	0.0926
0.80	0.1013	0.0798
0.85	0.0872	0.0741
0.90	0.0788	0.0610
0.95	0.0694	0.0551
1.00	0.0499	0.0533

average prec and % change vs. base case, for levels– .25, .50, .75

prec=	0.3329	0.3053
% prec change=		-8.3

recall	base	percentage change from base case precision
0.00	0.8095	-11.5
0.25	0.5552	-8.6
0.50	0.3234	-2.3
0.75	0.1202	-23.0
1.00	0.0499	6.8
av, 21 levels		-12.6

Figure 26: Example of Average Statistics Chart

Statistics averages for the following cases are shown below:

1) Test case: Query Expansion Scheme 1.
2) Test case: Query Expansion Scheme 2.

statistic	# queries used	1	2
norm r	30	0.8360	0.8352
norm p	30	0.6243	0.6152
rank r	30	0.0775	0.0660
log p	30	0.4565	0.4383
prec 10	30	0.3900	0.3533
prec 30	30	0.2078	0.2089
recll 10	30	0.3363	0.3010
recll 30	30	0.4819	0.4943
E,.5,10	30	0.6458	0.6820
E,.5,30	30	0.7721	0.7712
E,1,10	30	0.6728	0.7084
E,1,30	30	0.7292	0.7286
E,2,10	30	0.6772	0.7126
E,2,30	30	0.6509	0.6494

Figure 27: Example of Paired t-Test Chart

PAIRED t TEST:

1) Test case: Query Expansion Scheme 1.
vs.
2) Test case: Query Expansion Scheme 2.
based on 30 queries

			PROB. OF ERROR IF REJECT H0 (A=B	
STATISTIC	t	df	1 TAIL	2 TAIL
normal. recall	-0.07	29	0.5277	0.9447
normal. prec.	-0.51	29	0.6931	0.6139
rank recall	-1.11	29	0.8626	0.2747
log precision	-1.33	29	0.9031	0.1938
precision(10)	-1.78	29	0.9571	0.0858
precision(30)	0.13	29	0.4477	0.8954
recall(10)	-1.37	29	0.9089	0.1822
recall(30)	0.39	29	0.3482	0.6964
- E(.5;10)	-1.85	29	0.9629	0.0742
- E(.5;30)	0.10	29	0.4613	0.9225
- E(1;10)	-1.81	29	0.9600	0.0801
- E(1;30)	0.06	29	0.4777	0.9553
- E(2;10)	-1.62	29	0.9422	0.1156
- E(2;30)	0.09	29	0.4660	0.9319
prec at 0.0	-1.79	29	0.9577	0.0845
prec at 0.1	-1.76	29	0.9556	0.0888
prec at 0.2	-1.83	29	0.9610	0.0781
prec at 0.3	-0.54	29	0.7035	0.5931
prec at 0.4	-0.89	29	0.8104	0.3792
prec at 0.5	-0.27	29	0.6051	0.7897
prec at 0.6	-1.25	29	0.8898	0.2203
prec at 0.7	-2.13	29	0.9792	0.0416
prec at 0.8	-1.74	29	0.9535	0.0930
prec at 0.9	-1.70	29	0.9499	0.1002
prec at 1.0	0.16	29	0.4382	0.8764

Figure 28: Example of Sign Test Chart

SIGN TEST:

1) Test case: Query Expansion Scheme 1.
vs.
2) Test case: Query Expansion Scheme 2.
based on 30 queries

	NO. OF QUERIES FAVORING			PROB. OF ERR. IF REJECT H0 (A=B)			
				TIES INCLUDED		TIES NOT INCLUDED	
STATISTIC	A	B	TIED	1 TAIL	2 TAIL	1 TAIL	2 TAIL
normal. recall	14	16	0	0.4276	0.8551	0.4276	0.8551
normal. prec.	15	15	0	0.5724	1.0000	0.5724	1.0000
rank recall	14	16	0	0.4276	0.8551	0.4276	0.8551
log precision	15	15	0	0.5724	1.0000	0.5724	1.0000
precision(10)	13	6	11	0.9997	0.0019	0.9668	0.1687
precision(30)	9	12	9	0.8994	0.3613	0.3313	0.6625
recall(10)	13	6	11	0.9997	0.0019	0.9668	0.1687
recall(30)	9	12	9	0.8994	0.3613	0.3313	0.6625
- E(.5;10)	13	6	11	0.9997	0.0019	0.9668	0.1687
- E(.5;30)	9	12	9	0.8994	0.3613	0.3313	0.6625
- E(1;10)	13	6	11	0.9997	0.0019	0.9668	0.1687
- E(1;30)	9	12	9	0.8994	0.3613	0.3313	0.6625
- E(2;10)	13	6	11	0.9997	0.0019	0.9668	0.1687
- E(2;30)	9	12	9	0.8994	0.3613	0.3313	0.6625
prec at 0.0	9	5	16	0.9999	0.0005	0.9093	0.4227
prec at 0.1	12	7	11	0.9990	0.0062	0.9157	0.3588
prec at 0.2	14	8	8	0.9969	0.0176	0.9322	0.2864
prec at 0.3	11	14	5	0.7081	0.8551	0.3446	0.6892
prec at 0.4	12	16	2	0.4276	0.8551	0.2854	0.5708
prec at 0.5	12	15	3	0.5724	1.0000	0.3502	0.7003
prec at 0.6	15	13	2	0.8193	0.5839	0.7146	0.8501
prec at 0.7	16	13	1	0.8193	0.5839	0.7712	0.7103
prec at 0.8	16	12	2	0.8994	0.3613	0.8276	0.5708
prec at 0.9	17	11	2	0.9498	0.2012	0.9071	0.3447
prec at 1.0	18	8	4	0.9969	0.0176	0.9845	0.0776

Figure 29: Example of Wilcoxon Paired Sign-Rank Test Chart

WILCOXON PAIRED SIGN-RANK TEST:

1) Test case: Q1 = Q0'
vs.
2) Test case: Q1 = Q0' v Q0'
based on 30 queries

	RANK SUM OF QUERIES FAVORING		NORMAL DEVIATE		PROB. OF ERROR IF REJECT H0 (A=B)	
STATISTIC	A	B	1TAIL	2TAIL	1TAIL	2TAIL
normal. recall	210.0	255.0	0.45	0.45	0.3255	0.6509
normal. prec.	259.0	206.0	-0.56	0.53	0.7107	0.5928
rank recall	248.0	217.0	-0.33	0.31	0.6290	0.7577
log precision	264.0	201.0	-0.66	0.64	0.7448	0.5237
precision(10)	133.5	56.5	-1.57	1.53	0.9417	0.1262
precision(30)	112.0	119.0	0.10	0.10	0.4585	0.9170
recall(10)	130.0	60.0	-1.43	1.39	0.9234	0.1650
recall(30)	109.5	121.5	0.19	0.19	0.4242	0.8484
- E(.5;10)	129.0	61.0	-1.39	1.35	0.9175	0.1776
- E(.5;30)	111.0	120.0	0.14	0.14	0.4447	0.8894
- E(1;10)	132.5	57.5	-1.53	1.49	0.9369	0.1365
- E(1;30)	111.0	120.0	0.14	0.14	0.4447	0.8894
- E(2;10)	132.5	57.5	-1.53	1.49	0.9369	0.1365
- E(2;30)	109.0	122.0	0.21	0.21	0.4174	0.8348
prec at 0.0	79.0	26.0	-1.69	1.63	0.9550	0.1026
prec at 0.1	133.0	57.0	-1.55	1.51	0.9393	0.1313
prec at 0.2	176.0	77.0	-1.62	1.59	0.9477	0.1117
prec at 0.3	153.0	172.0	0.24	0.24	0.4043	0.8087
prec at 0.4	183.0	223.0	0.44	0.44	0.3285	0.6570
prec at 0.5	167.0	211.0	0.52	0.52	0.3027	0.6055
prec at 0.6	214.0	192.0	-0.26	0.24	0.6033	0.8110
prec at 0.7	266.0	169.0	-1.06	1.04	0.8553	0.2993
prec at 0.8	236.0	170.0	-0.76	0.74	0.7772	0.4593
prec at 0.9	231.0	175.0	-0.65	0.63	0.7418	0.5312
prec at 1.0	214.0	137.0	-0.99	0.97	0.8390	0.3345

Figure 25 shows the standard recall-precision output for two experimental runs that will hereafter be referred to as A and B. At the top, an overall heading is given, followed by a short description of each case. Then, the main part of this report is given – the precision of each case for recall values between 0.0 and 1.0, in increments of 0.05.

Following that chart, the average precision measure is given for cases A and B. To facilitate comparisons, the percent change against the base, that is case A, is shown for all other cases.

The bottom part of the output shows percentage changes at some of the major recall levels, between the base and other case. Finally, the percentage changes are themselves averaged. The average percent change, for all 21 levels, however, misrepresents cases where two curves are fairly close but then widely diverge at only part of the curve. It is interesting but not recommended for presentation purposes since the values at high recall levels are often improperly emphasized. And emphasis of those values is not sensible since precision at that end of the curve is often based largely on random breaking of ties among nonretrieved documents.

Figure 26, the second part of the reporting on cases A and B, omits precision values at recall levels but otherwise shows averages for all of the statistics recorded for each query of each case. By inspecting this table line by line, one can see where A is better than B, where they are tied, and where B is better than A. Note that the E values are lower for better performance, whereas other statistics show increases as effectiveness goes up.

Figures 27 through 29 are the actual statistical comparison reports. For each, all statistics recorded on a per query basis are considered separately. Probabilities shown give the likelihood of error if one rejects the null hypothesis that A = B. In other words, they show how likely it is that differences observed between the two cases are due to random error. Generally one would like these probabilities to be less than 0.01 or else 0.05 or certainly 0.10, for 1, 5, or 10% confidence levels, respectively.

Two other points of terminology are important to explain. First, there is the matter of ties. When two cases are compared, should the ties be ignored or should they be included? To allow either interpretation, and to show the differing results based on each assumption, both sets of values are shown.

Second, there is the matter of using one or two tailed tests. In one tailed tests there should be some fact in the experimental design that implies that the result distributions are bounded in some special way. Thus, to be as conservative as possible, one might use a two tailed test with ties not included.

In the earlier version of SMART, a composite statistic, using Chi square, was computed to summarize the results of the individual tests given. Such a composite value should be based upon independent measures. If one requires a single probability, it seems that a consistent grouping of the most independent values available should be used – such as probabilities for the eleven different precision values at the selected recall levels.

Figure 27 gives results for the t-test, Figure 28 for the sign test, and Figure 29 for the Wilcoxon signed rank test. If one of these three must be selected, then the last mentioned is probably the most appropriate. To give convincing proof of significance, however, it is useful to report agreement between two or three of the methods when such occurs.

The sample output shown should serve as a key to interpretation of SMART output. It is also possible to produce other reports, such as graphs of recall level averages. These and other more general capabilities are touched on in the next section.

5.3. S Interface

5.3.1. S Package

Numerous statistical packages are available nowadays. For interactive users of the UNIX system, the S statistical package [Becker & Chambers 1981], whose underlying approach, philosophy, and algorithms are described in [Chambers 1977], provides a comprehensive facility for data analysis. Numerous parts of the package have been employed to produce output for [Fox 1983b].

S is both a language and a collection of routines. In keeping with the overall philosophy of UNIX, intermediary results can be easily manipulated in a variety of ways and one can add in tailored code using the S or FORTRAN languages when available programs are not quite appropriate. However, it is generally possible for someone familiar with S to accomplish in a short period of time a large number of data reduction and analysis tasks, using the package as is.

Arrays are basic elements of data in S, and a variety of operators are available to work with them. These are generally high level, and the style of use is somewhat like that of APL. Special structures, such as matrices and time series, are easily created and commonly used by routines provided – automatic setting of default values allows casual use, while specialized applications can be accommodated since there are frequently a large number of options that can be specified.

The regression capabilities are adequate; sample results were shown in [Fox 1983b Chapter 3]. An optional interface to GLIM can be used to allow even more complex types of linear models to be studied [Becker & Chambers 1981].

5.3.2. Use of S

Raw data created from SMART runs, when put in textual form, can be loaded into S. Typically, one creates a file that looks like a vector or a matrix, and loads that into the corresponding S data structure.

After that, one can manipulate the data inside S, creating other structures as needed, and obtaining outputs from a variety of runs. In [Fox 1983b], output from each of the routines listed in Table 5 have been included.

Table 5: Principal S Routines Used for Processing and Output

- Dendogram Printing
- Histogram
- Leaps and Bounds Regression
- Multidimensional Scaling
- Plot
- Regression
- Scatter plot
- Single Link Clustering
- Statistical Distributions

Another way to use S is to call on functions in the S library. New routines can be added to the public or local libraries. Systems such as SMART can utilize such functions, to avoid having to program algorithms already working in the S package.

To illustrate this approach, Figure 30 shows how the one and two tailed versions of the t-test are coded in the S language. Many built in functions, such as LENGTH, mean, ptdis, abs, sqrt, and var, are callable, and that simplified the task. The t-test routines shown were accessed from the SMART evaluation programs to produce the output of Figure 27.

Figure 30: Examples of S Functions –
Description, Sample Calls, Definitions for 1, 2 Tails Versions

ttest: Function to compute a student's t-test

ttest(dat)

ARGUMENTS:
dat: Numeric structure. NA's not allowed.

VALUE:
A 5-element vector. The first element is the mean of dat. The second element is the standard error of the mean. The third element is the value of Student's t. The fourth element is the two-tailed significance level determined by the t-value and the degrees of freedom. The fifth element is the degrees of freedom.
If there are not at least two non-NA data values, the vector returned will contain all zeroes.
If the standard error of the mean is zero, the returned t-value is arbitrarily set to 99999.

EXAMPLES:

```
ttest(vect - 5.0)        # Tests the mean of vect against 5.0
?row(matrix,ttest)       # Returns a 5 by nrow(matrix) array describing
# t-tests for each row of matrix
```

This is the ttest which returns a two tailed significance level.

```
FUNCTION ttest(dat/REAL,NAOK/)
STATIC(real mean,var,ptdis)
STRUCTURE(res/REAL,5/)

for (i=1; i<=5; i=i+1)  NASET(res[i])
if (LENGTH(dat) < 2)  RETURN(res)
res[1] = mean(dat,LENGTH(dat))
res[2] = sqrt(var(dat,LENGTH(dat)) / LENGTH(dat))
if (res[2] > 0.0)  res[3] = res[1]/res[2]
else  res[3] = 99999.
res[5] = LENGTH(dat) - 1
res[4] = (1.0 - ptdis(abs(res[3]),res[5]))*2.0
RETURN(res)
END
```

(see function for 1 tail test on next page)

Figure 30 continued : Examples of S Functions –
Description, Sample Calls, Definitions for 1, 2 Tails Versions

This is the ttest which returns a one tailed significance level.

```
FUNCTION ttest(dat/REAL,NAOK/)
STATIC(real mean,var,ptdis)
STRUCTURE(res/REAL,5/)

for (i=1; i<=5; i=i+1)  NASET(res[i])
if (LENGTH(dat) < 2)  RETURN(res)
res[1] = mean(dat,LENGTH(dat))
res[2] = sqrt(var(dat,LENGTH(dat)) / LENGTH(dat))
if (res[2] > 0.0)  res[3] = res[1]/res[2]
else  res[3] = 99999.
res[5] = LENGTH(dat) - 1
res[4] = .5 - ptdis(abs(res[3],res[5])
RETURN(res)
END
```

It can be seen that S is a very handy tool to facilitate experimentation using the SMART system. Data can be prepared, examined, and presented to help in better understanding the retrieval process.

5.4. Discussion and Conclusion

In order to evaluate the results of SMART runs, and to compare the behavior of different methods, it was essential to have an evaluation component for the UNIX SMART system. Ellen Voorhees continued the development of that capability beyond the initial limited version, and interfaced the S package to facilitate use of its functions.

The current evaluation facility follows the original SMART system in many respects. It adds in the E measures, for two levels of retrieved set size and for three different relative valuations of the importance of recall versus precision. By using the S package and INGRES system, it encourages manipulation of available data beyond that required for casual use. Thus, regression methods have been applied in a variety of ways that would have been very difficult to engineer using the IBM 370 version of SMART. Methods are basicly the same, but ease of use is improved.

Some might argue over certain of the methods employed in computing results that have been presented in [Fox 1983b]. The most unusual procedure followed is in connection with strict Boolean queries. Since comparison with p-norm and vector methods was required, the use of recall-precision values was chosen, even though a Boolean query typically has only one recall-precision pair associated with it. However, to a user examining the set retrieved by a Boolean query, when those documents are viewed in sequence, a recall-precision curve is actually being followed. As long as the number retrieved is not too small and the number relevant not too large, this method seems appropriate.

A second point that might be debated is in connection with handling of nonretrieved relevant documents. These are assumed to be randomly distributed among the nonretrieved set. As long as there are not too many nonretrieved relevants, this should not be very much of a concern. Besides, to achieve full recall one would have to delve into that nonretrieved set, and with zero similarity between query and relevant documents, it is not unreasonable to assume that discoveries of these relevant documents would be random events.

A final point of concern to some might be the use of an average precision value for a single valued overall measure. However, by selecting recall levels .25, .50, and .75, one seems to capture the most important region of the recall precision curve, and that curve is still one of the best ways to analyze retrieval behavior. By avoiding use of precision for very low and very high recall levels, one eliminates problems associated with having precision based on too few or too many documents. Precision values at low recall levels are unrealistic since users will certainly be willing to view at least 10 or more retrieved documents, and so values at recall levels typically associated with fewer than that many retrieved are unimportant. On the other hand, high recall levels are rarely achieved in practice since users typically stop before locating all of the relevant material. If they must have close to 100% recall, those precision values will have a good deal of uncertainty since the persistent searcher must view a large number of documents in what amounts to an almost random or hit and miss series of examinations of retrieved sets. Finally, a user can understand what "average precision" means; an average user would be concerned with the middle portion of the recall-precision curve in most typical searches.

The other decisions made in implementing the evaluation package seem to be generally beyond question. Various possible interpretations can be made with the several tests given and the options available, such as how to handle ties, or whether to consider one or two tailed distributions. Thus, a user with a particular requirement or idea about appropriate evaluation techniques should be able to simply select the results of interest among those given.

All in all, the SMART evaluation package should be of value to users of the SMART system. Future additions and enhancements should not be difficult to make, what with the extensible nature of S, and the easy to access data available in relational form from experimental runs.

6. Future Plans

The SMART system has been evolving for over two decades, and will no doubt continue to do so for many years to come. As in the past, it will be improved so as to better carry out the tasks presently performed, and will be extended to enable testing of new ideas and methods.

Future work can therefore be grouped into a number of areas. One is that of consolidating completed work and arriving at a packaged form so that various people requesting copies can be given a coherent, documented, tested basic system. Secondly, there are projects underway at Cornell that will lead to improvements or adaptations to the current system. Finally, there are extensions suggested by the work carried out for [Fox 1983b].

6.1. Packaging

In order to quickly package the current software in an effective form, three types of efforts are needed. First, a good deal of documentation must be written. While certain components have been described in this report, many portions have not been dealt with. Also, documentation is required at a more detailed level than the general overview of data and algorithms given here.

Second, the system should be better split into a grouping of routines that do not require any special software tools beyond standard UNIX, versus the rest, which make use of INGRES, S, or other add-on packages. Such a division could be made almost immediately, and, without a great deal of work a number of programs now using INGRES could be modified so as not to require that support.

Finally, a cleanup of current software is required. Untested or duplicate versions should be omitted, and only essential, reliable elements included. All of the related programs should be identified and shell scripts provided to facilitate interconnection. Test collections should be finalized so every future user has the same starting point for experimentation. At that point, copies of the current system could be released, and subsequent development work might be geared toward periodic, perhaps yearly, subsequent updates.

6.2. Cornell Projects

Many projects have been in process for several years at Cornell, relating to the SMART system. Since most will result in additions to the basic system, it is of value to briefly summarize the most important ones here.

6.2.1. UNIX Retrieval Tools

One Cornell project relating to SMART is that of providing a usable system for on-line use by UNIX users. The first application suggested was that of allowing searching of the UNIX manual.

Later, news broadcasts and electronic mail collections could be handled.

Ellen Voorhees has been working on the indexing aspects of these projects. Chris Buckley has been working on providing a set of retrieval tools to enable a system programmer to establish a new collection, based on data in some consistent format; subsequently, documents could be added, deleted, and retrieved. The components to accomplish most of this are already part of the SMART system, but those need to be properly documented and packaged, as described above.

The principal new tool that has been developed provides a flexible way to get data into a form that can be handled by the SMART indexing package. Two things are involved:

(1) Having a specification file that describes each concept type, how it is identified in the original input, and how it should be handled by the indexing system.
(2) Having a special high-level parser that will convert the original input into a form that can be accommodated by the SMART indexing routines.

Some other problems are harder to face and may require considerable effort to resolve. The current indexing process works with a static collection that is analyzed all at once. If documents are added and deleted afterwards, provision must be made for changing the dictionary and for re-computing statistically based document weights. If clustered files are consistently used, the current storage techniques developed for documents should be flexible enough. Hence, no problems are expected there. However, to have clustering work on totally new collections, some better, perhaps experimentally based, techniques to choose proper parameter settings may be required. Furthermore, once dynamic modification of collection files is allowed, provision must be made for maintaining data consistency in spite of hardware or software failures, even when retrieval and updating are carried out simultaneously.

6.2.2. Phrase and Thesaurus Construction

Identification of statistical phrases in the indexing process has been worked on by Ellen Voorhees. The volume of data that needs to be handled is rather large, making experimentation difficult. Furthermore, if is not easy to evaluate the effectiveness of various types of phrase addition methods, since query collections are typically rather short.

A related task is that of allowing thesauri to be included in SMART. Automatic procedures for identifying thesaurus categories could be programmed. However, problems similar to those mentioned above about handling phrases must be addressed. Further, categories suitable for use in the collection as a whole can be difficult to isolate; a scheme whereby categories can be dynamically identified as an integral part of the search process might be wiser to investigate.

6.2.3. Probabilistic Retrieval

Chris Buckley has worked on improving the performances of various probabilistic retrieval schemes. Instead of assuming that terms occur independently, it is posited that there are probabilistic dependencies among terms that govern how they will co-occur in documents. When dependencies are accounted for, either the tree or the BLE approaches can be utilized.

Considerable work has gone on to arrive at good techniques of estimation. One problem is that of consistently handling trivial cases, such as when no relevant retrieved documents contain the term of interest.

Another area of concern is that of properly considering dependencies between pairs and triples of terms. Some theoretical insights by Yu suggest that improvements can result from following certain principles when working with such groupings. Experimentation needs to be continued to confirm those insights [Yu et al. 1983]; very careful handling of probabilities and estimation methods is required to make this successful. It is also vital that such approaches be thoroughly tested in the more realistic predictive mode, in addition to retrospective or upper bound analysis.

6.2.4. Office Automation

Earlier work described in [Nodtvedt 1980] suggested that retrieval methods could be applied to business correspondence. A number of collections of electronic mail have been collected, and it is hoped that involved users will submit queries and make relevance judgments.

Experimentation is planned to determine the effectiveness of automatic indexing and retrieval methods for such collections of messages. Using multiple concept types should be of benefit. Some scheme to identify links between documents, like bibliographic references in article collections, should result in significant gains in retrieval effectiveness. In addition, the patterns of connection should be of interest in their own right.

6.3. Extensions

A number of further developments of the SMART system can be carried out to extend work done in connection with earlier studies of this author.

One idea is to enhance the DNF system for Boolean and p-norm feedback [Salton, Fox, Buckley & Voorhees 1983] to allow more flexible query construction, so that automatic phrase and thesaurus category identification can occur dynamically. If positional information is recorded about concepts, then pairs and triples whose elements occur near to each other can be favored. Further, low frequency concepts that appear nearby can be thought of as being in the same thesaurus category, and can be connected with an OR instead of an AND operator.

With positional information, the p-norm method can be applied to metrical operators besides the normal Boolean ones, and membership functions for such clauses can decrease in proportion to the actual distances between concepts. Thus if one employed "X w Y" to request having concepts X and Y within the same sentence, that function might yield a value of 1.0 when X and Y do appear in the same sentence, but only give 0.5 if they are located in adjacent sentences instead.

Furthermore, vectors could be computed for smaller items than just documents. This would be of particular value in full text retrieval applications. Thus, if one were to retrieve passages in the *Bible*, a vector might be computed for each verse and for each chapter. A link would connect verses and chapters, and users could specify whether they would like to find relevant verses or relevant chapters.

Another extension would be to employ certain artificial intelligence techniques to help in the feedback process. Though it is not yet feasible to do in-depth analysis and store the resulting representations for large collections of documents, it might be possible to do some type of special analysis of retrieved relevant documents during the feedback process. By developing an expert system [Small,S. 1980] for this situation, and integrating it into the feedback loop, one might obtain significant improvements.

Regarding the SMART implementation, better design methods might be usable. The idea of combining relations and abstract data types [Fox 1981] may eventually be developed further, and with a reasonably efficient implementation might be usable for additional enhancement of the SMART system.

In any case, the SMART storage and search methods need to be generalized. An option to use inverted files should supplement the clustered approach, and dynamic, secure versions of both algorithms should be developed. If possible, the clustering and cluster search schemes should be generalized so as to automatically improve and adapt, and so fewer parameters are required.

Combining document retrieval and database handling is another useful area of development. P-norm queries could be directly used in the database environment – preliminary experiments could be carried out through the use of different concept types for each attribute of a relation. If results are promising, simpler inclusion of such data into SMART can be provided by allowing vector creation where no indexing is required at all. And indexing collections where most of the data is in relational form could also be simplified.

With suitable data, p-norm feedback where multiple concept types are available could be tested. Such an approach might even be appropriate for approximate match searches using database queries. Special membership functions could be utilized for concept types such as dates, so that similarity goes down as the separation between two dates increases. Algorithms for identifying differences between strings [Wagner 1974] or for general handling of spelling problems [Peterson 1980] could be used to determine membership functions for names included in questions submitted. Even numeric ranges could have associated functions for indicating closeness of matches based on the actual data distributions. If one views data types as values [Demers & Donahue 1983], then this kind of special processing for various concept types might be abstracted and generalized through a more sophisticated theory-based perspective.

A final extension to mention for the SMART system is that of improving the man-machine interaction. It is hoped that the methodology presented in [Ehrich 1982] can be adapted to SMART, so that dialogue management aspects can be cleanly separated from the retrieval component, enabling each to be improved and tested in a relatively independent fashion. Then, enhancements to SMART to enable a better user interface, connected in with a coordinated approach to handling objects present in the office of the future, might be of considerable value; such an easy to use electronic storage and retrieval system could provide considerably more functionality than would a simple mechanization of traditional file cabinet techniques.

6.4. Conclusion

Though the current SMART system has demonstrated its usefulness for experimental studies, there are many other investigations yet to be completed, and many ideas for further work. The current design is aimed at incorporating some of the latest techniques in information storage and retrieval, and the present implementation provides a considerable number of useful capabilities. It is hoped that this summary of the state of the project as of the middle of 1983 will aid others interested in the history or use of SMART software.

REFERENCES

[Becker & Chambers 1981]
Becker, Richard A. and John M. Chambers. *S: A Language and System for Data Analysis.* Bell Laboratories, 1981.

[Cane 1964]
Çane, Mark. Adapting SMART to the M.I.T. Compatible Time-Sharing System. Information Storage and Retrieval, Vol. 8, Cornell University Department of Computer Science, Ithaca, New York, 1964.

[Chambers 1977]
Chambers, John M. *Computational methods for data analysis.* John Wiley & Sons, New York, 1977.

[Date 1982]
Date, C. J. *An Introduction to Database Systems. Volume I.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[Demers & Donahue 1983]
Demers, Alan and James Donahue. Making Variables Abstract: An Equational Theory for Russell. Technical Report 82-534, Cornell University Department of Computer Science, Ithaca, New York, 1983.

[Duda & Hart 1973]
Duda, R. O. and Peter E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York, 1973.

[Ehrich 1982]
Ehrich, R. W. DMS: A System for Defining and Managing Human-Computer Dialogues. In Proceedings of Analysis, Design, and Evaluation of Man-Machine Systems IFAC/IFIP/IFORS/IEA Conference, Federal Republic of Germany, 1982, pages 367-374.

[Fox 1981]
Fox, Edward A. Implementing SMART for Minicomputers Via Relational Processing with Abstract Data Types. In Joint Proceedings of SIGSMALL Symposium on Small Systems and SIGMOD Workshop on Small Data Base Systems. *ACM SIGSMALL Newsletter*, Vol. 7, No. 2, October 1981.

[Fox 1983a]
Fox, Edward A. Characterization of Two New Experimental Collections in Computer and Information Science Containing Textual and Bibliographic Concepts. Technical Report 83-561, Cornell University Department of Computer Science, Ithaca, New York, July 1983.

[Fox 1983b]
Fox, Edward A. Extending the Boolean and Vector Space Models of Information Retrieval with P-Norm Queries and Multiple Concept Types. Cornell University Ph.D. Thesis, University Microfilms, Ann Arbor, Michagan, August 1983.

[Kernighan & Ritchie 1978]
Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Lawani 1970]
Lawani, S. M. *The Aslib-Cranfield Studies on the Evaluation of Indexing Systems*. University of Ibadan Institute of Librarianship, Ibadan, Nigeria, 1970.

[Lesk 1966]
Lesk, M. E. Design Considerations for Time Shared Automatic Documentation Centers. Information Storage and Retrieval, Vol. 11, Cornell University Department of Computer Science, Ithaca, New York, 1966.

[McGill et al. 1976]
McGill, Michael J. et al. Syracuse Information Retrieval Experiment (SIRE): Design of an On-Line Bibliographic Retrieval System. *ACM SIGIR Forum*, Vol. 10, No. 4, Spring 1976, pages 37-44.

[McGill & Noreault 1977]
McGill, Michael J. and Terry Noreault. Syracuse Information Retrieval Experiment (SIRE): Rationale and Basic System Design. Report, Syracuse University School of Information Studies, Syracuse, NY, May 1977.

[Michelson et al. 1971]
Michelson, D. et al. An Experiment in the Use of Bibliographic Data As a Source of Relevance Feedback in Information Retrieval. In *The SMART Retrieval System, Experiments in Automatic Document Processing*, edited by Gerard Salton, Prentice Hall, Inc., Englewood Cliffs, N.J., 1971.

[Nodtvedt 1980]
Nodtvedt, Einar. Information Retrieval in the Business Environment. Technical Report 80-447, Cornell University Department of Computer Science, Ithaca, New York, 1980.

[Noreault, Koll & McGill 1977]
Noreault, Terry, Matthew Koll and Michael J. McGill. Automatic Ranked Output from Boolean Searches in SIRE. *Journal of the American Society for Information Science*, Vol. 28, No. 6, November 1977, pages 333-339.

[Peterson 1980]
Peterson, James L. Computer Programs for Detecting and Correcting Spelling Errors. *Communications of the ACM*, Vol. 23, No. 12, December 1980, pages 676-687.

[Ritchie & Thompson 1974]
Ritchie, D. M. and K. Thompson. The UNIX Time Sharing System. *Communications of the ACM*, Vol. 17, No. 7, 1974, pages 365-375.

[Rocchio 1964]
Rocchio, Jr., Joseph J. Possible Time-Sharing Organization for a SMART Retrieval System. Information Storage and Retrieval, Vol. 7, Cornell University Department of Computer Science, Ithaca, New York, 1964.

[Salton 1963]
Salton, Gerard. Associative Document Retrieval Techniques using Bibliographic Information. *Journal of the ACM,* Vol. 10, No. 4, October 1963, pages 440-457.

[Salton 1968]
Salton, Gerard. *Automatic Information Organization and Retrieval.* McGraw-Hill, New York, 1968.

[Salton & Lesk 1968]
Salton, Gerard and Michael E. Lesk. Computer Evaluation of Indexing and Text Processing. *Journal of the ACM,* Vol. 15, No. 1, January 1968, pages 8-36.

[Salton 1971a]
Salton, Gerard, editor. *The SMART Retrieval System: Experiments in Automatic Document Processing.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1971.

[Salton 1971b]
Salton, Gerard. Automatic Indexing Using Bibliographic Citations. *Journal of Documentation,* Vol. 27, No. 2, June 1971, pages 98-110.

[Salton 1975]
Salton, Gerard. *Dynamic Information and Library Processing.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[Salton 1980]
Salton, Gerard. The SMART System 1961-1976: Experiments in Dynamic Document Processing. In *Encyclopedia of Library and Information Science, Vol. 28,* 1980, pages 1-36.

[Salton & Wu 1981]
Salton, Gerard and Harry Wu. A Comparison of Inverted File Searching with Cluster Tree Search Operations. Unpublished Report, Cornell University Department of Computer Science, Ithaca, New York, 1981.

[Salton, Fox & Wu 1982]
Salton, Gerard, E. Fox and H. Wu. Extended Boolean Information Retrieval. Technical Report 82-511, Cornell University Department of Computer Science, Ithaca, New York, August 1982.

[Salton, Buckley & Fox 1983]
Salton, Gerard, C. Buckley and E. A. Fox. Automatic Query Formulations in Information Retrieval. *Journal of the ASIS*, Vol. 34, No. 4, July 1983, pages 262-280.

[Salton, Fox, Buckley & Voorhees 1983]
Salton, Gerard, E. A. Fox, C. Buckley and E. Voorhees. Boolean Query Formulation with Relevance Feedback. Technical Report 83-539, Cornell University Department of Computer Science, Ithaca, New York, January 1983.

[Salton & McGill 1983]
Salton, Gerard and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[Siegel 1956]
Siegel, Sidney. *Non-parametric Statistics for the Behavioral Sciences.* McGraw-Hill, New York, 1956.

[Small,S. 1980]
Small, Steven. Word Expert Parsing: A Theory of Distributed Word-Based Natural Language Understanding. TR-954, Department of Computer Science, Univ. of Maryland, College Park, Maryland, September 1980.

[Williamson 1974]
Williamson, Robert Edward. *Real-Time Document Retrieval.* Cornell University Ph.D. Thesis, June 1974.

[Wirth 1971]
Wirth, N. Program Development by Stepwise Refinement. *Communcations of the ACM,* Vol. 14, No. 4, April 1971.

[Yu, Luk & Siu 1979]
Yu, C. T., W. S. Luk and M. K. Siu. On Models of Information Retrieval Processes. *Information Systems,* Vol. 4, No. 3, 1979, pages 205-218.

[Yu et al. 1983]
Yu, C. T., C. Buckley, K. Lam and G. Salton. A Generalized Term Dependence Model in Information Retrieval. Technical Report TR83-543, Cornell University Department of Computer Science, Ithaca, New York, February 1983.

[SMART Project 1974]
SMART Project. User's Manual for the SMART Information Retrieval System. Technical Report No. 71-95 Revised April 1974, Cornell University Department of Computer Science, Ithaca, New York, 1974.

[Snedecor & Cochran 1980]
Snedecor, George W. and William G. Cochran. *Statistical Methods: Seventh Edition.* The Iowa State University Press, Ames, Iowa, 1980.

[Stonebraker et al. 1976]
Stonebraker, Michael et al. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976.

[Van Rijsbergen 1979]
Van Rijsbergen, C. J. *Information Retrieval: Second Edition.* Butterworths, London, 1979.

[Wagner 1974]
Wagner, R. A. Order-n Correction for Regular Languages. *Communications of the ACM*, Vol. 17, No. 5, May 1974.

[Williamson & Williamson 1970]
Williamson, D. and R. Williamson. A Prototype On-Line Document Retrieval System. Information Storage and Retrieval, Vol. 18, Cornell University Department of Computer Science, Ithaca, New York, 1970.

[Williamson et al. 1971]
Williamson, D., R. Williamson and M. E. Lesk. The Cornell Implementation of the SMART System. In *The SMART Retrieval System, Experiments in Automatic Document Processing*, edited by Gerard Salton, Prentice Hall, Inc., Englewood Cliffs, N.J., 1971.