Implementation of the SMART Information Retrieval System *

Chris Buckley

TR 85-686

May 1985

Department of Computer Science
Cornell University
Ithaca, New York 14853

* This study was supported in part by the National Science Foundation under grant IST 83-16166.

Implementation of the SMART Information Retrieval System

Chris Buckley
Department of Computer Science
Cornell University
Ithaca, New York 14853

This study was supported in part by the National Science Foundation under grant IST 83-16166.

1. Introduction

The SMART information retrieval package is a set of programs composing a fully automatic document retrieval system. It allows easy creation, maintenance, and use of on-line document collections. As more information is being kept on-line every day; it becomes more essential to have methods of easy, natural access to the information. The SMART package is primarily a tool for investigating some of these methods. In addition, it is quite usable itself for many applications.

The current SMART system is a collection of programs written in 'C' under the UNIX operating system. It is operational at Cornell University on at least three machines: DEC VAX 11-750, DEC VAX 11-780, and SMI SUN (a 68000 based super micro-computer).

What is described here is only the latest in a long series of SMART implementations; the earliest one being in the early 1960's [1,2,3,4,5]. This, however, is the first implementation allowing actual practical use of the system by a naive user. Previous versions were for experimental purposes only.

This new version naturally draws very heavily on the older versions for its algorithms, although almost no code remains from those versions. A special debt is owed to Ed Fox's implementation which immediately preceded this one [6]. His was the first UNIX implementation, and many of the lessons learned during his work were very useful here.

The current implementation of the SMART system is covered in the remainder of this paper. First, the features and goals of the system are described. The information retrieval process in general is then related to the particular modules and programs within the system. The overall approach to accessing information and parameters is discussed, followed by a brief look at some of the internal data structures used. There is a short section on concurrency, consistency, and protection features. The conclusion discusses the future of the SMART system: what should be done (or re-done) and what could be done.

2. Goals, Features, and Requirements

This implementation of SMART contains few new or radical concepts. Instead, it attempts to provide a solid framework for future work in information retrieval. The two major goals of the current version are to

1. Provide a flexible experimental system for research in information retrieval. See [6] for a discussion of desirable system capabilities and design principles for experimental work.

2. Provide a fast, portable, interactive environment for actual users.

These two goals naturally conflict with each other; the current SMART design is an attempt to satisfy each as much as possible.

The system is concerned with three major types of users: the experimenters, the database administrators, and the naive users. The experimenters need the ability to easily change system parameters and to easily add or replace program modules. The database administrators must be able to create and maintain a collection of documents without worrying about the peculiarities of the particular collection. It should be possible to initially specify the features of the collection and not worry about them again. The users need to be able to enter a query and view the results without knowing anything about the internal parameters of the system, being aware only of the collection features which are relevant to them (such as the type of information contained in a document). An interactive help facility is necessary for the casual user.

The current system is a first step in satisfying these goals. The major lack at the moment is a satisfactory user interface. There is a usable interface here at Cornell, but more work is needed.

In no particular order. the major features of the SMART system are:

2.1. Size

The system consists of roughly 200 source files with a total of 35,000 lines of code. A fair amount of this code will not be used in any one application; there are several complete modules that only an experimenter or a database administrator with a particular application in mind would use.

2.2. Simplicity

Access to the main UNIX data files is all straightforward, as is the access to the internal data structures. In particular, no attempts were made to get the maximum performance out of disk operations. This is a great boon to the experimenter who needs to modify the system, but of course it means non-optimal performance for the casual user.

2.3. Uniform access to UNIX data files

The core of the SMART system is a group of utility procedures designed to efficiently access the collection and retrieval files needed to work with the system. Each type of UNIX data file is considered a distinct object with its own instantiation of access procedures. These are described in detail later.

2.4. Interactive

The system is designed to be used for small to medium scale collections, and offers reasonable speed and support for these actual applications. Large collections (greater than 200,000 documents) can be handled, but pose additional efficiency problems. It is not clear that a UNIX environment (in which a programmer does not have direct control over disk accesses and locations of blocks on disks) is suitable for large collections.

2.5. Flexibility

The design of the SMART system concentrates on two types of flexibility. The first is complete flexibility at a number of levels in specifying the parameters for all operations. All parameters have reasonable default values. In addition they (possibly) can be given values within a collection dependent specification file. This means a database administrator can tailor the parameters to one particular database application. These values, in turn, can be over-ridden at command execution time by specifying a parameter and its value on the command line.

At the program design level, flexibility is achieved by allowing very easy expansion of the most commonly used modules. For example, if an experimenter wishes to add a new procedure for computing the similarity between two vectors, two lines in one "data" file needs to be changed and the retrieval program needs to be re-linked

2.6. Speed

SMART is not blindingly fast, but it is quick enough to be used as an actual retrieval system. The performance figures depend greatly on the exact situation, but to give a rough estimation of speed:

1. Indexing is done at 2,000 to 2,500 characters per CPU second.

2. Sequential retrieval is done at about 500 document-query similarity computations per CPU second. This improves by a factor of 4 for experimental runs where a large batch of queries are run at once.

3. Inverted file retrieval is much harder to quantify since it is heavily dependent on the length of the query. A typical user's query on a medium size collection takes almost 1.5 CPU seconds to return the top documents.

These figures were obtained on a DEC VAX 11-780 computer with a floating point accelerator.

In a typical multi-purpose, multi-user, single CPU environment, actual elapsed time is dominated by the time needed to load the SMART programs. It takes considerably longer to fetch the executable programs from disk than the programs themselves take! In an environment dedicated to information retrieval, this overhead will not exist.

2.7. Disk Space Requirements

The disk space needed for the indexed collections obviously depends directly on the size of the collection. A rough estimate would be that the indexed collection will take up .8 times the space of the text version of the collection. This includes a dictionary, display information, and both an inverted file and a sequential representation of the indexed documents. Note that both of these representations are not necessarily required, but each is useful for particular applications (eg. inverted files make fast retrieval possible, sequential files are needed for good feedback methods.) If a database administrator is willing to sacrifice some functionality and effectiveness, the indexed collection disk space requirement can be lowered to .2 times the text version.

The source of the SMART package is about 1.1 Megabytes, not counting object files and object libraries. The executable programs take up about 3.5 Megabytes.

2.8. Memory Requirements SMART should be run on an operating system with virtual memory, or on hardware with large amounts of physical memory. Most of the programs require less than .5 Megabytes of memory for program and data space. There are currently two bottlenecks in the indexing module though. At the moment, the entire dictionary must be kept in-core when anything in it is being changed and the dictionary can reach 2 Megabytes. Also, the program which converts a sequential document representation into an inverted file representation needs significant memory for working space. There is no inherent reason why these programs need to take this much space and future versions may not. Until then, the collection indexing process will require a large amount of virtual space. The processes an ordinary user would run (retrieval, feedback, query indexing, display) require only a limited amount of space (< .5 Megabyte) so their impact on the system is much less.

3. Information Retrieval

Before going any farther, the general model of information retrieval used in the SMART system needs to be discussed. For further details about information retrieval models, Salton and McGill [7] provide a solid introductory text.

An on-line collection of *documents* is pre-supposed. These *documents* can be anything from electronic messages to programming manual entries to full technical journal articles. Every document (potentially) contains several distinct kinds of information. Possibilities include date of publication, author of the document, receiver of the document, a supplied list of keywords, a supplied placement of the document within hierarchical categories, the title, the abstract, a list of other documents that are cited, etc. Several types of additional information are contained in the text of the document; for example: dates, proper nouns, times, numbers, and normal text words.

Each of the information classes above may be useful to a user submitting a query, but there must be some way of distinguishing each classification type within the document representation. This leads to the association of a classification type (or *ctype* for short) for each document concept.

The documents in the collection are automatically *indexed* with a document representative being assigned to the corresponding document. This representative contains the system's idea of the important concepts found in the document. The representative consists of a list of concepts, the ctype of each concept, and weights for each concept; the weight giving a value to the importance of the concept.

Users come to the SMART system with an information need and try to convey this need to the system. Their initial statement of their need can be a piece of natural language text, a query using Boolean connectives (AND, OR), a list of keywords, etc. The system assigns a *query* representative for the need, either a simple list of concepts and weights like the document representatives, or something a bit more involved which gives more structure to the representative.

A retrieval function within the system then calculates the *similarity* of the query representative to each of the document representatives. (In practice, not every document needs to be examined - depending on the similarity function.) The documents are presented to the user in order of their similarity to the query. It is hoped that the similarity order will have some correspondence to likelihood that the user will judge the document useful.

At this point, the user has the option to examine some of the top retrieved documents, and give a judgement of whether the documents were *relevant* to their information need. If the user desires more documents, a new query representative can be automatically constructed from the old representative and some of the concepts occurring in the relevant documents. This process is known as *relevance feedback*. The new *feedback* query can then be compared against the document collection and more documents can be retrieved for the user. This process continues until the user has as many documents as they desire.

The study of information retrieval concerns itself with the numerous methods which can be used to accomplish the above procedure. There have been many models of the information retrieval process proposed over the years and many different methods of implementing these models. The SMART system was designed to experimentally evaluate these methods and models.

4. The Levels of Smart

In considering the implementation of SMART, it helps to look at the system as being composed of four levels of programs and procedures. Going from the "highest" level to the "lowest", they are:

1. The *user request* level.
2. The *task implementation* level.
3. The *object access* level.
4. The *database access* level

A user submits some kind of request to the topmost level of the system. This request could be adding documents to the collection, retrieving documents from the collection, or one of several other actions. The topmost level gathers needed information from the user, for example, a query. The system then decides what tasks need to be done in order to accomplish the user's request. In the case of a retrieval operation the separate tasks might include indexing the user's query, retrieving some documents, displaying the documents to the user, and constructing a feedback query. The topmost level invokes the appropriate program for each task.

The programs in the task implementation level are each responsible for performing one task. The arguments to a program indicate the location of the input and output data. Within the task programs, the access to data is through *relational file objects*. (The precise meaning of this term is explained later.) Each object class consists of the data structure for a tuple of this object class, along with a set of procedures for finding, reading, and writing tuples of this class. In the retrieval example, the retrieval program will access document representatives by reading tuples from a *vector* relational object. It will then output the ids of the most similar documents by writing tuples to a relational object of class *top-ranked*.

The object access level is comprised of the procedures for accessing relational file objects and the procedures for reading specification files. These procedures implement the only access programs have to long-lived data (outside of the original text documents). Logically, the relational object procedures invoke database access procedures to read and write data on UNIX files. Each set of database procedures puts information on disk using a different storage method. For example, *top-ranked* tuples are physically stored on disk in sorted order in one file, while *vector* tuples are stored in two files; the first a direct access file giving pointers to locations in the second file.

The database access level is the lowest logical level in the SMART hierarchy. Note that this level is completely unknown to the top two levels of the hierarchy. A task program has no knowledge of the storage method used for any object. In practice, if there is only one relational object using a database access level procedure, the implementation of the database access procedure is combined with the object access procedure. An example of this would be the dictionary relational object. The disk storage method is so tuned to the requirements of the relational object that a separation of the two levels would not be worth it.

There is actually one lower level that was designed, but has not yet been implemented. The database log level is responsible for keeping track of all changes to a database object. This log allows easy abortion of a partially completed transaction. Unfortunately, not all versions of UNIX support the primitives needed to implement this level. As a consequence, the more expensive alternative of "shadowing" entire relations was adopted as the rollback mechanism.

Figure I shows some of the programs and procedures at each level of the SMART system. It may be worthwhile referring back to this figure during the next few sections which describe each level in more detail.

5. User Request Level Programs

The main programs which deal with users are still in a state of flux. The high level information given below should be correct, but the reader is warned that the details may be out of date. Also, the menu driven sub-system described below may not be available outside of Cornell. The reader is referred to "The SMART User's Manual" for the current facts about the local implementation.

There are two sub-levels of executable programs within the user request level. The bottom level is composed of shell scripts combining some of the task implementation programs into one user oriented function. These include:

smart_create - Create and initialize a new collection
smart_enter - Add the designated new documents to the collection
smart_add - Construct and add a single document to the collection
smart_delete - Delete the designated documents from the collection
smart_retrieve - Display useful documents in response to a user's query
smart_demo - The same as **smart_retrieve** except the system gives a running explanation of the actions taken on a particular collection.
smart_run - Perform an experimental retrieval and evaluation

This level is implemented as shell scripts to provide the most flexibility possible. Application programs can use this level directly. Experimental programs may find it useful to adapt one of these programs to fit the specific need of the experiment.

The top level is composed of a single program, **smart**. **Smart** is the single entry point into the SMART system for the casual user with a normal information retrieval application. If **smart** is called with no arguments, then the user will be put into a menu-driven subsystem which interactively offers and explains the options available to the user throughout the invocation. The menu system is designed for either a casual user or a casual experimenter (eg. student in an

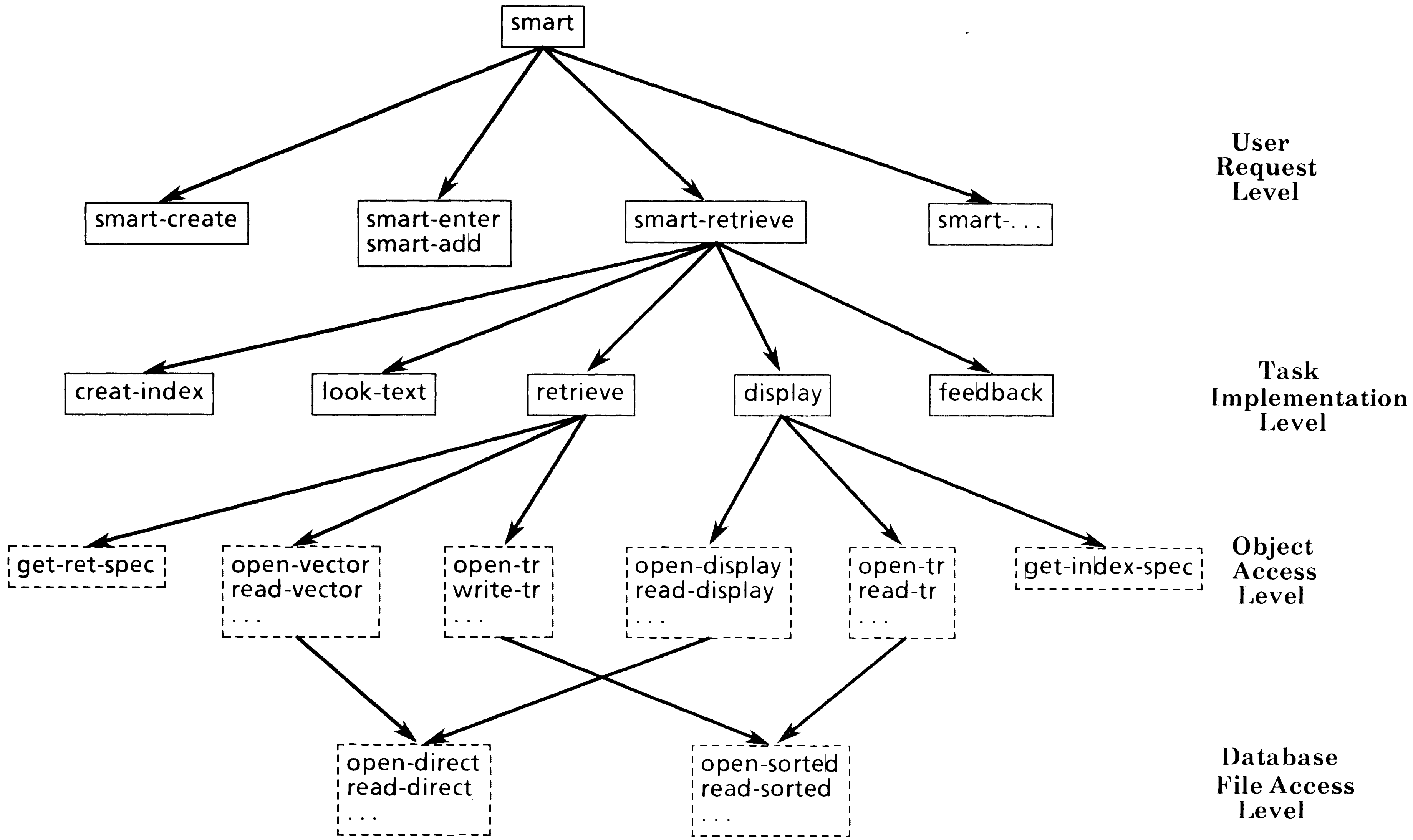

Figure I. Partial View of SMART Control Structure

information retrieval class). The most useful feature is a help window describing the current option being considered.

If the user knows what to do, **smart** can also be called with arguments. In this case the menu system is by-passed completely, and the programs in the bottom level are called directly by **smart**. For example,

smart retrieve cacm

will find the *cacm* document collection, and then invoke **smart_retrieve** on it. **Smart** is also responsible for performing any concurrency locking required by the particular action to be done. This is described in more detail later.

The duality between the menu system and the direct invocation of programs seems to be essential. It is difficult for a novice to use SMART unless explicitly offered the available options and information about each option. An experienced user does not want the overhead of the menu system, and needs to have access at times to the lower level programs. The current system makes both easily available, at least at the top level.

6. Task Implementation Level

There are a number of logically separate modules of the implementation. Each of these is composed of several programs. The indexing module takes a document (or query) in its original form and prepares it for use in retrieval. The retrieval module takes an indexed query (or queries) and a document collection and returns a list of those documents that best match the query. The display module takes a list of documents and presents it to the user. The user can view the documents and optionally decide whether each document seen is relevant to them. The feedback module takes the list of documents that the user has seen and constructs a new query based on the original query and those documents that the user liked. The evaluation module, used only for experiments, takes retrieval results and a list of relevant documents and evaluates how well the retrieval performed.

There are also a fair number of miscellaneous programs that do odd jobs like creation and maintenance of the databases and others that allow access to the information in the databases.

6.1. Indexing Module

Four stages are defined for a typical document that is to be added to an existing collection. As it enters the system, it is in its original form, which is also the form in which it is to be displayed to the user after the document is retrieved. The documents in a particular collection may have a different structure from the documents in any other collection. This structure is represented by dividing the document into logical *sections*. Different keywords are used to indicate the different sections of a document; different parsing methods may be used on each section; different types of information may be made available for retrieval. One general indexing mechanism cannot handle all of the varieties of document forms.

6.1.1. Pre-parsed Form

The SMART system takes care of the idiosyncrasies of each document collection in two ways. The first is by defining an index specification file for each collection. This file specifies what sections might be found in each document and how each section is to be parsed. The type of information available within each section is also given here.

The second way is by having a simple, collection-dependent program responsible for breaking each document into its component sections. This program, called the **pre-parser**, must

1. Recognize the beginning of a new document (if there is more than one per file in this collection) and associate the section representing start of document with this particular filename and location within the file.

2. At the beginning of each new recognized section, give the location within the file of the section. This allows each section to be displayed separately, if desired.

3. Recognize the end of the document and give the location of the end.

4. **Pre_parser** must take care of any collection dependant parsing that the database administrator wishes done for this collection. For example, in a collection of electronic mail it may be difficult to separate the name of the sender from the name of the sender's machine. **Pre_parser** would be responsible for that.

Other than distinguishing sections (and possibly performing special parsing tasks), **pre_parser** just passes the document through to the next stage.

6.1.2. Document Indexed Form

The program **creat_index** takes the document in pre-parsed form and parses it, section by section. The parsing method to be used on each section is given by the index specification file. **Creat_index**

1. Breaks the document into individual tokens (or *concepts*)

2. Determines the type of each concept (proper noun, word, date, number, etc.)

3. Checks to see if the concept is on a negative word list. This is a list of words (of the appropriate type) which do not convey any information that would be useful for retrieval. For example the words "the", "of", and "an" all convey no information about the content of a document. Any concept on the negative list will not added to the document representative.

4. If desired, stems the concept by removing suffixes. This allows words which are different forms of the same word to match each other. For example, "clustering" and "clustered" will both be reduced to "clust".

5. Outputs a tuple giving <concept, type of concept, document id>. Op-

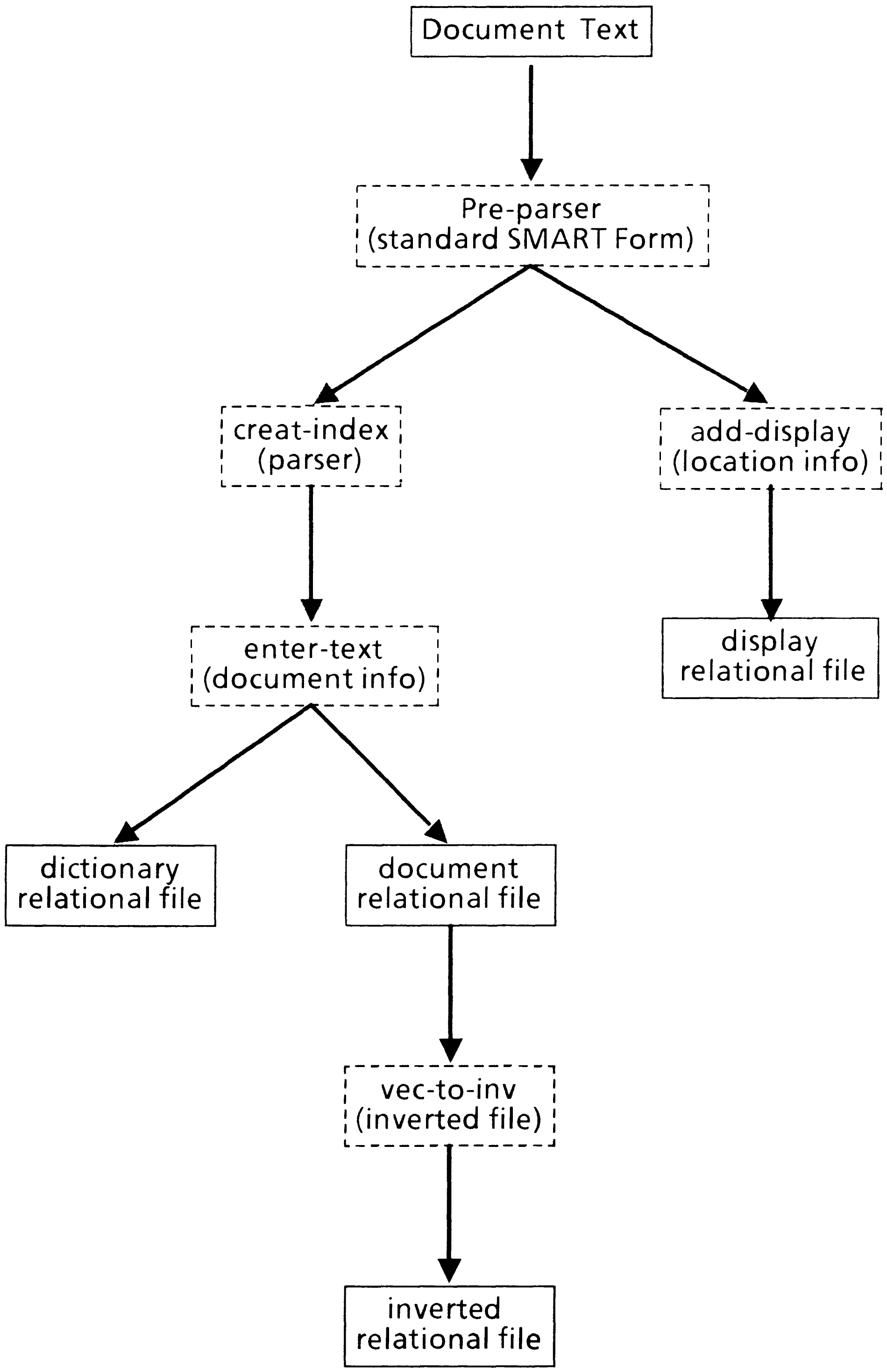

Figure 2. Document Indexing

tionally, additional lexicographic information can be output giving the paragraph number (within the document), sentence number within the paragraph, and word number within the sentence. This information could be used to form phrases, but isn't used in the standard SMART indexing process.

6.1.3. Collection Form The program **enter_text** takes the output of **creat_index** and actually adds the document and concepts to the existing document collection. For each document, **enter_text**

1. Enters each new concept in the collection dictionary (or increments the document collection frequency count if the concept is already in the dictionary and has not been incremented for this document yet).

2. Computes a document weight for each concept in the document which represents the importance of the concept in deciding whether this document is relevant to a user's query. The particular weighting scheme used for a document is collection dependent and is given in the index specification file.

3. Add the document to the document collection.

4. If desired, each <concept, document> pair can be added to the inverted file as well.

6.1.4. Query Indexing

Natural language queries are indexed in a fashion very similar to that of documents. **Pre_parser** and **creat_index** are invoked on the query exactly as if it were a document. Instead of **enter_text**, the program **look_text** is invoked. **Look_text** performs the same actions as **enter_text** except that the dictionary is not changed and the new query (queries) is entered in a new query collection.

Boolean queries (or Pnorm queries - an extension of Boolean queries) are parsed by a different set of programs, since there is no way to preserve both the section structure of a collection document and the logical structure of the Boolean query. However, the same basic actions are performed for the Boolean query - negative word removal, stemming, dictionary lookup. The major difference is that the tree structure of the Boolean query must be preserved and represented. The programs **ppre_parse** and **pindex** index the Boolean queries.

6.1.5. Display Creation

One additional phase in the indexing of a document is the addition of the document to a relation giving the physical location and structure of the document. This information is used when it comes time to display a document to the user. The program **add_display** takes the output of **pre_parser** and adds the file location and location of sections within the file of each document to the display relation.

6.2. Retrieval Module

There is only one program, **retrieve**, in the retrieval module, but it is a very flexible program! **Retrieve** runs an indexed query collection (possibly consisting of just one query) against an indexed document collection, calculating (theoretically) the similarity between each document and each query. The output is either a list of the documents which most closely match each query or a list of a given set of documents and the ranks which would be assigned them if the documents were sorted in decreasing order of similarity to the query. In an experimental research setting, this set of documents would be the known relevant documents for each query and The ranks of these relevant documents are used to evaluate the effectiveness of different retrieval methods.

All of the options of **retrieve** are given in the retrieval specification file passed to it. These options include information like

1. Type of input query (vector, boolean tree, pnorm)

2. Retrieval method to be used (discussed below)

3. Type of output desired (just top documents, ranks of relevant documents, both)

4. The location of the input (document collection, query) and the output.

5. Etc.

The parameters whose values can be specified within the specification file are given reasonable default values. For most operational runs, as opposed to experimental runs, the specification file consists of a single line telling what collection is to be used. On the other hand, a complicated experimental run that, say, uses a different matching function for every type of information in the query, could run to 30 lines of parameters.

The various retrieval methods form the heart of **retrieve**. To allow complete flexibility, there are three levels of retrieval methods: the collection access level, the vector access level, and the ctype access level.

6.2.1. Collection Level The collection access level allows control over which documents will be compared to the query, and what data structures will be used in doing so. Five types of collection access methods are currently implemented (or being implemented):

1. Sequential Access. Every document in the collection is compared with the query. This is slow (typical speeds are 500 similarities per CPU second) but for experimental purposes is often the best method.

2. Inverted Access for Vector Queries. Like SIRE, this method uses only an inverted file. For each query term, the documents containing that term are found, along with the weight of that term in each document. The inner product of the query and each document can be calculated without ever refering to the document vector itself. This offers a great saving in time, since the only concepts looked at are those which have an influence on a

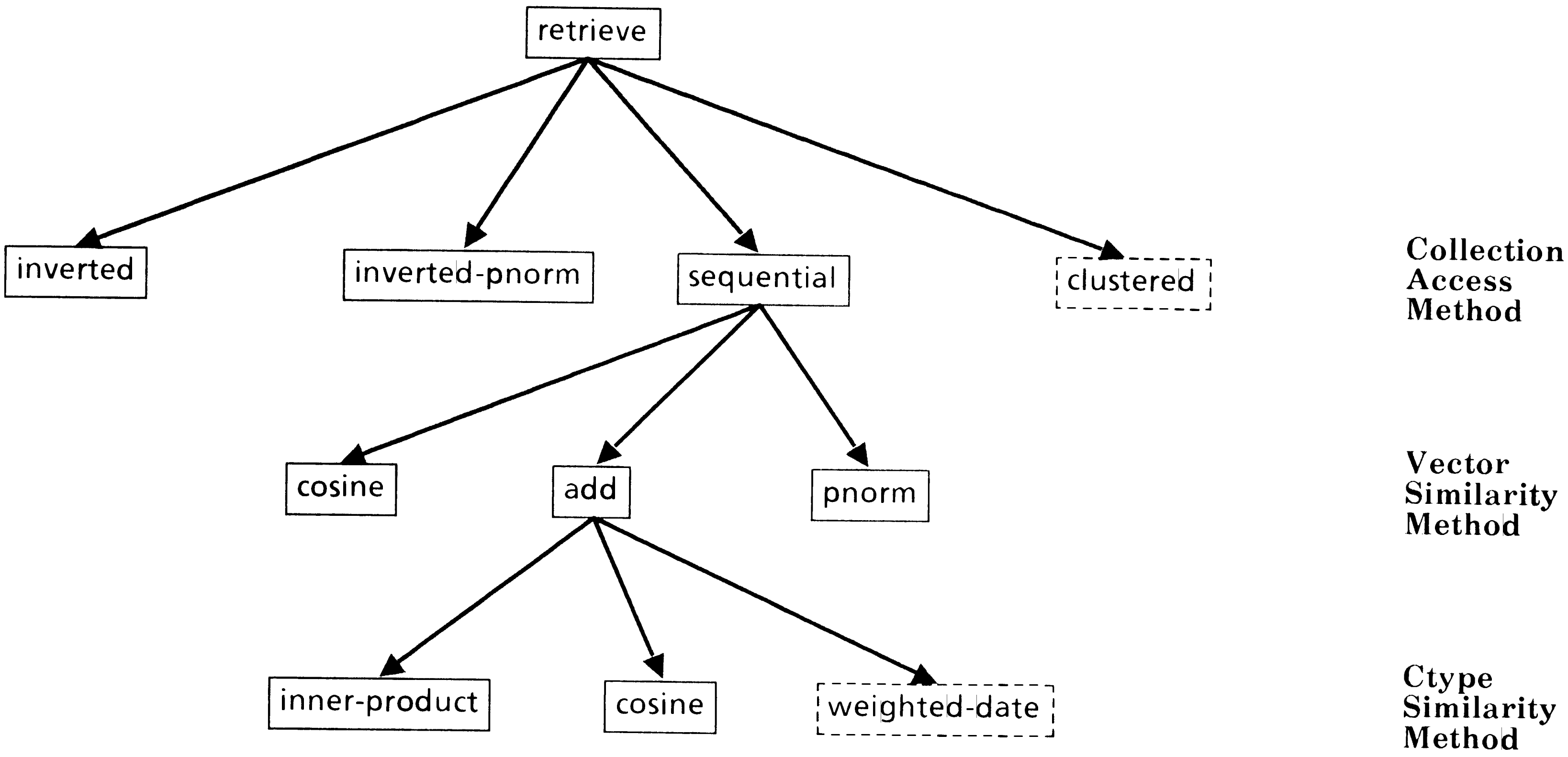

Figure 3. Levels of Retrieve

document's similarity. This method is quite useful in practice because of its efficiency, but isn't flexible enough to be used for many experimental purposes.

3. Inverted Access for Tree Queries. If the original query is a boolean query, or an extended boolean query (eg. Pnorm), then an inverted file can still be used, although the operations performed are more of a list manipulation nature than an inner product.

(4). Clustered Access. If the document collection has been hierarchically clustered so that very similar documents are grouped together into clusters, and these clusters are further grouped, etc; then the collection can be searched efficiently by only looking at clusters which match the query well. Theoretically, most of the documents in the collection will never be looked at. Again, this is a notion which has achieved a lot of attention because of its efficiency, but is not a good experimental test-bed since it is difficult to evaluate different types of retrieval functions. An experimenter can never be certain whether a retrieval function performs poorly because it actually was poor, or because the vagaries of the clusters dominated the results.

(5). Mixed Retrieval. This combines inverted file access with vector access. One application is if a query has some types of information which are required to be matched (eg. date, or author) but has other types of information for which the "best" match is desired. The inverted file is used to obtain a list of documents which match the factual information. Each document on the list has a vector similarity function calculated for it and the ranking of the documents is based on this similarity value.

6.2.2. Vector Level

The vector access methods calculate a similarity between the query vector and a document vector. Note that the inverted file collection access methods will never invoke the vector level. There are currently 2 types of vector similarity functions:

1. Weighted Addition. Each type of information contained in a query can have a separate retrieval function defined for it (see the **Ctype Level** descriptions below). This method produces the weighted sum of each of the ctype sub-vector simlarity function results. The weights assigned to each type of information can be set in the retrieval specification file.

2. Extended Boolean (and Boolean). The extended boolean model works more naturally with a vector approach than with the inverted file approaches. The type of information of a concept is irrelevant here, so a ctype sub-vector function is not needed.

6.2.3. Ctype Level

This level is the one that people typically think of when talking about a vector information retrieval system. The similarity between the parts of a query vector and document vector which contain the same kind of information is calculated here.

In most older information retrieval system models, there is only one type of information; so the functions contained here would logically be put at the vector level. The ctype similarity functions implemented are:

1. Inner Product - The similarity is the sum over the terms that are in both the query and the document of the product of the query weight and the document weight.

2. Cosine - The usual cosine similarity function as given, say, in Salton and McGill *Introduction to Modern Information Retrieval.* It is closely related to the inner product function, but the vectors involved are normalized by their length so that a long document (with lots of terms that can match a query) will not have an "unfair" advantage over a short document.

3. Co-ordination Level - Returns the number of terms matched in the document and the query.

4. Partial Matching - General catch-all for a number of routines which do partial matching of factual information. The normal example is a date, where the closer the document date is to a date given in a query, the better.

6.3. Display Module

After documents are retrieved in response to a user's query, they need to be displayed to the user. The **display** program takes the output of **retrieve** and shows the top documents to the user. Normally, the titles of all the retrieved documents are presented and the user is asked which documents they wish to see. The title of a document is assumed to be the first words appearing in the section of the document designated as the title_section. Title_section is a parameter appearing in the index specification file.

The user has available a number of actions that can be taken while displaying documents:

q(uit) : exit display, updating any necessary files
x(it) : exit display, without updating anything
h(eaders) : Display the menu headers again
+, n(ext) : Display the next document
c(urrent) : Display the current document again
<did> : Display document number <did>
e(dit) : edit current document, replacing old version in collection
w(rite) file : append copy of current document to the named file
? : Display this help message
d(elete) : Delete current document from collection
u(ndelete): Un-delete a previously deleted document
!command : Invoke a shell to execute command

The **delete** option will delete the current document from the entire database. This assumes that the user has write permission for the database. When the user finally quits display (using **quit**), the system will run a program to delete the designated documents from all relations within the database. Note that while running display, the user can change their mind and decide not to delete a document. This is done by **undeleting** a document, or simple using **xit** to leave **display**.

The **edit** option allows a user with write permission to "alter" a document in the collection. The document is copied to a separate file, the user edits that file, and after the user **quits**, the old document is deleted from the collection and the new document is added. Note that the actual physical old document is never touched and remains in the UNIX file system until removed. Only the collection references to the old document are removed.

The **write** option allows the user to make a copy of the document in a designated file.

The **!command** option gives the user the chance to execute any UNIX command from within **display**. An example of this would be **writing** a document to a file and then using the mail command to send the document to someone else.

One of the options at the initial invocation time of **display** is for the system to request relevance judgements from the user. If this option is used, the user is asked whether or not that document is useful, after each document is displayed to the user. The results are stored and eventually written to an output relation. A feedback query is constructed from the original query and those documents that the user considered relevant. This feedback query can be run to retrieve more documents (if the user desires).

Another option at invocation time of **display** is the choice of sections to display. Designated sections of retrieved documents can be displayed. This can function either as a security measure, or just as a way to avoid looking at non-useful sections of a document.

6.4. Feedback Module

The programs in the feedback module all take an initital query and some relevance judgments supplied by the user to form a new query. This query will be run by **retrieve** again to return additional documents to the user.

There are potentially at least as many feedback methods (and parameters) as retrieval methods. Parameters and methods are given by a feedback specification file. Most of the methods are still in the experimental stage, so only a few broad types of feedback will be mentioned here.

Boolean (Pnorm) feedback constructs a new boolean query composed of terms seen in the relevant documents. The clauses of the new query are constructed by estimating the number of documents that are expected to be retrieved by terms in a prospective clause. A "good" clause should not retrieve too many or too few documents. This is based on work done by Ed Fox.

Probabilistic feedback estimates the probability of an occurrence of a term (or occurrence of two or three terms) in a relevant document. These probabilities, based upon the seen relevant documents, are used to estimate the probability of relevance of the rest of the documents in the collection.

Term relevance weighting is perhaps the standard feedback approach. The weights of terms occurring in the relevant documents are combined (possibly) with the weights of terms in the original query to construct a new query.

6.5. Evaluation Module

The programs in the evaluation module are normally of interest only to the person doing information retrieval research. Given a list of ranks of relevant documents from a particular retrieval, an overall evaluation of how well that retrieval method worked can be done. A large number of evaluation criteria are calculated by the SMART system since there is no agreement within the information retrieval community on any one evaluation method. The most important method commonly used is the recall-precision chart, which measures the precision (how many retrieved documents were relevant) against the recall (how many of the relevant documents were retrieved). Other measures used are normalized recall, normalized precision, rank recall, log precision, precision or recall after 10 documents, precision or recall after 30 documents, and various parameterizations of van Rijsbergen's E measure. The interested reader is referred to Salton and McGill, *An Introduction to Modern Information Retrieval* for details of these measures.

Evaluate is the major program in this module. It is the one which takes the list of relevant documents and produces a relation which gives a query by query evaluation of the retrieval method using all of the above evaluatin measures.

The program **print_results** takes one or more evaluation relations prepared by **evaluate** and outputs a file ready to be printed containing the averages of the evaluations over all the queries in the collection. If there is more than one evaluation relation given as an argument, the results are presented in a head to head fashion to allow easy comparison.

Several programs implementing various significance tests exist. The Wilcoxon test, t-test, and sign test all attempt to answer the question "Is retrieval method A better than retrieval method B?" (or more precisely, "Can we rule out the hypothesis that method A is equivalent to method B?"). Note that none of these tests are really valid for information retrieval purposes. All of them contain assumptions about result distributions that are not necessarily true for retrieval evaluation results. The sign test probably contains the least objectionable assumptions and should be used if this type of test is needed.

There are subtleties present when the results of feedback queries need to be evaluated. The problem is what to do with the documents judged relevant after the initial query, since these items are used to construct the new feedback query. It is obviously unfair to allow the change of rank of these documents to influence the evaluation of the retrieval (or feedback) method. Somehow the ranks need to be *frozen*. This is the purpose of the **freeze** program. It allows for the freezing of ranks according to several different schemes. Each of the schemes has advantages and drawbacks, so no one method is accepted for being the "right" way to evaluate feedback runs.

7. SMART Specification Files

The specification files are designed to provide a uniform method of supplying parameter values for the various programs in the SMART system. In a flexible experimental system like SMART, the task of organizing, remembering, and setting all of the possible parameters is non-trivial. There is no way a system administrator or experimenter can be expected to set these parameters unless there is a systematic way to change just the parameters that directly affect the database or experiment they are working on.

The format of a specification file is a series of lines of the form

<parameter_field> <parameter_value>

Blank lines are ignored; anything on a line following a '#' is a comment.

For the most part, the specification lines are context free - the setting of one parameter will not affect the value of the another parameter. There are exceptions to this: to give values for parameters which depend on the concept type, the pseudo-parameter_field *current_ctype* is used. The value given for the last invocation of *current_ctype* determines the ctype for a parameter specification. Similarly, *current_section* is used to give the section name currently valid.

Three specification files being used at the moment: one for indexing, one for retrieval, and one for feedback. Each supplies a large number of parameter values, ranging from document collection name to the similarity function to be used when comparing a specific type of information within two vectors.

It is impractical to ask the person running a program to specify all of these values each time the program is run. Under the method used, only the parameters which take on non-standard values need to be specified. The SMART system contains a list of reasonable default values for all the parameters. These are used if no other value supplants them. Every program takes an argument of a specification file which contains values for those parameters with values different from the default. This is normally a file associated with each collection, containing information about how this collection differs from the norm. As a third means of giving parameter values, every program takes an optional list of <parameter_field> <parameter_value> pairs which gives information about how this particular run differs from the normal program run on this collection.

For database administrators and experimenters who don't remember all of the possible settings, a program (**display_menu**) is used to help form specification files. It consists of an editor with interactive help available to decide which parameters need to have non-default values specified, and what values are sensible to use. (Note: this program is not available outside of Cornell since it uses a non-public domain editor (emacs) as its base. Other people should refer to the help files distributed with the system in order to obtain the needed information (in the directory "smart/lib")).

8. SMART Relational File Objects

The next two sections go a bit more deeply into the implementation of the file and data structures that SMART uses. The casual reader may want to skim or skip these sections. A detailed understanding of them is not required for the rest of the paper.

A SMART relational object is stored as one or more UNIX files on disk. These objects contain information that needs to stored between invocations of programs. One example of a long-lived, but ever-changing object would be the dictionary of concepts for a collection. A short-lived example would be a query representative, which exists only while a user is running a query. A structure needs to be put on these objects; otherwise, anybody trying to understand what the system does will have an impossible task.

The general object in the current SMART implementation is an "ordered" relation, not necessarily in first normal form. These objects can only be accessed through routines which have a separate instantiation for each type of relation. All of the implementation features peculiar to a given type of relation are hidden from the user. This is obviously a standard approach; the only trick is to do it without excess copying of the huge amounts of data needed in information retrieval. Most implementation hiding techniques involve copying of data.

A previous SMART implementation was based on the relational database system INGRES. This provided a uniform method for accessing data in almost any manner desired. Information retrieval access, though, tends to be very simple but involve massive amounts of data. A standard relational database like INGRES will copy the data to be returned several times before the user sees it. This is unacceptable for information retrieval purposes. Instead, non-standard relational objects are being used. The type of access to the data is very limited compared with INGRES, but it is much faster.

A major improvement in speed comes with the ability to have non-normalized relations. This allows the system to return pointers to other relations or collections of data as "first class" attributes of the relation. This is important in two ways. First, a large space savings results from not having a fully normalized relation - duplicate attribute values do not have to be stored or returned to the user. Second, the pointer returned can be a pointer to the *implementation's* data space itself. This allows an implementation to read large amounts of data at a time into its private buffers and return pointers to it without ever copying the data. This produces large savings if the data stored on disk can be stored in a form immediately usable by a user's program. Note that this is only feasible if the user's programs are trustworthy, something a standard database system can never assume.

Each relation can be regarded as a collection of tuples, ordered by some attribute(s); a potentially infinite number of tuples can be fit between any 2 tuples. At any time when dealing with a relation, the concept of position within the relation is important (thus another difference between the SMART concept of a relational object and a true relation). This position can be at the beginning of a tuple, or at the end of the tuple and thus at the beginning of the "space" between this tuple and the next tuple.

Operations on relations include : create, open, seek, read, write, close. Each relational operation is invoked (currently) by calling the procedure <operation>_<relation_type>. For example, to create a relation of type *dict*, the procedure **create_dict** is called with the appropriate arguments. Each of these operations is described below. One additional operator can be called on a relation: **get_rel_header**. This will return information about the relation, such as the number of entries, what type of relation it is, etc.

8.1. The Relational Operators

8.1.1. Get-rel_header (relation_name)

Given a relation name, information about the relation is returned as a pointer to a tuple of type REL_HEADER This tuple contains the following attributes:

num_entries : number of tuples in relation

relation_type: integer code for the type of relation this is.

relation_subtype: integer code for a subtype of relation. (This isn't used at present.)

max_primary_value : limit to a value of the attribute on which the relation is sorted. This is a soft "limit", it can be exceeded by future insertions of tuples, but at any one time, is greater than or equal to the maximum value of that attribute attained by any tuple in the relation.

max_secondary_value: Same as above, but applies to a secondary attribute of the relation.

other implementation dependent attributes which are not relevant to the user of the relation also appear in this tuple.

8.1.2. Create_<type> (relation_name, relational information)

Create a relation of the given type. The input relational information is a tuple of type REL_HEADER, which contains information which may be used to decide the type and magnitude of the data structure that will be used to contain the relation.

8.1.3. Open_<type> (relation_name, mode)

Open the (possibly) previously created relation "relation_name". Mode describes the operations possible to perform. Mode is an integer which is the logical **or** of some subset of

SRDONLY
SWRONLY
SRDWR
SCREATE
SLOCK
SEXLOCK
SBACKUP
SINCORE

SRDONLY, SWRONLY, SRDWR indicate whether the relation will be read only, written only, or both read and modified (respectively). SCREATE indicates that the relation does not exist, and **create_<type>** should be called to create it. SLOCK, SEXLOCK indicate whether it is desirable to lock the relation so that other invocations of this or other programs are not able to change it while it is in use here. SBACKUP indicates whether a backup copy of the relation should be made in "<relation_name>.bak" before committing any writes. SINCORE is an advisory flag telling whether the relation will be accessed in such a way as to make it worth keeping it entirely in-core. Some relations and access methods require the relation to be in-core; in this case, SINCORE will automatically be set for the user.

After the call to open, the relational cursor is positioned at the beginning of the first tuple of the relation. A small integer index is returned to be used for all future operations with this relation. If an error is detected, the integer UNDEF is returned.

8.1.4. Seek_<type> (relation_index, relation_tuple)

Change the position of the cursor according the information in "relation_tuple" which is a tuple of type <**type**>.

If "relation_tuple" is NULL, then the cursor is positioned at the beginning of the relation again and 1 is returned.

If "relation_tuple" contains enough information to theoretically completely identify a tuple in the relation, then a 1 is returned if that tuple is present, and the cursor is positioned at the beginning of that entry. If the tuple is not present, then the cursor is positioned at the beginning of a between tuple gap in the relation, such that the tuple could be inserted in that gap while still preserving the ordering on the relation. 0 is returned in this case.

If there is enough information to identify a consecutive group of tuples, (for example, if the relation is sorted by <field_1, field_2> and <field_1> is given), then the cursor is positioned at the beginning of this group of tuples and 1 is returned.

If there is not enough information to identify any tuple, then UNDEF is returned. The cursor is not positioned at any legal position in the relation. In the example above, if <field_2> is specified but <field_1> is not, then UNDEF is returned.

8.1.5. Read_<type> (relation_index, &relation_tuple)

Read a tuple of the appropriate <type> into the space pointed to by "relation_tuple". The first tuple in the relation following the current cursor position is read. If there are no more tuples, 0 is returned, else 1 is returned. If an error is detected, (for instance, the cursor is not positioned legally), then UNDEF is returned. Depending on the relation type and the mode of access, values returned by one read may or may not be guaranteed to remain fixed across the next read. In particular, if relations (like *vector*) involving large amounts of data are not opened with SINCORE, then the input buffers will eventually be copied, and pointers contained within "<relation_tuple>" may suddenly point to invalid data. See the discussions of the implementation of the individual relation types for further information.

8.1.6. Write_<type> (relation_index, relation_tuple)

Add the tuple "relation_tuple" to the relation designated by "relation_index". The tuple is added at the first empty position in the relation following the current cursor position. There is no concept of overwriting a tuple with another tuple;

however, **write_<type>** may be used to delete the tuple at the current cursor position if the primary attribute value of "relation_tuple" is UNDEF (or NULL if the primary attribute value is a pointer to another type of data object).

8.1.7. Close_<type> (relation_index)

Relinquish access to this relation. It is only after executing the close that an outside process or user is guaranteed access to any changes that may have been made to the relation while it was open.

8.2. Tid as a distinguished attribute

There is one distinguished attribute that is treated differently from the others when performing the above operations: *tid*. *Tid*, standing for "tuple_id", if included as an attribute in a relation, has an integer value assigned by the implementation which uniquely identifies the tuple within the relation. Furthermore, *tid* is always the primary attribute of the relation. It can be thought of as the implementation dependent position of the tuple.

Since the implementation assigns the value of *tid*, it necessarily cannot be given by the user during a **write_<type>**. The user may supply any value for *tid* at the beginning of the write. The implementation will replace this value within "relation_tuple" by the actual *tid* assigned the tuple. If the write fails (returns UNDEF), then the value for *tid* is set to UNDEF. Note that the user can even assign the value UNDEF to *tid* without any ambiguity with the operation of deleting a tuple. The tuple will only be deleted when the cursor points to it, and the user supplies UNDEF as the principal attribute value (in this case, *tid*). Since the cursor points to a tuple, the user must have just done a seek to that tuple, and therefore must have supplied the correct *tid* to the seek.

9. Implementation of SMART Relational Objects

The SMART relational objects, whose access methods are described in the above section, are designed to ease the task of the application programmer. Since they present a uniform interface to the programmer, once the programmer has learned to access one object type, he or she can access any object type. The programmer can then concentrate on the information needed to solve the particular task at hand.

At least 12 types of file objects are currently used within SMART.

1. dictionary
2. vector
3. display
4. inverted_file
5. simple_inverted_file
6. graph
7. array
8. relevant_rank
9. top_rank
10. evaluation
11. pnorm_vector

12. linked_inverted_file

Each relational object is used to store a different type of information. (For the most part, the type of information can be guessed from the name of the relation.) Each has its own set of access routines (described above) and has a separate external form to be used for these routines. The implementation of a file object may have an additional internal form which will be invisible outside of the implementation.

Each implementation keeps a static array which contains information about each opened relation of the appropriate type. Note that since this array is static, only a fixed number of relations of any type can be open at once. Each opened relation has its own buffers and pointers; it is never the case that an operation on one opened relation will affect an operation or any data associated with a different opened relation.

The SMART implementation of relational objects has two logical levels. The user (application programmer) can access a set of functions for each object without worrying about the implementation or storage method used for that relational object. Underneath this user level, though, is a lower level of database access methods. This lower level roughly corresponds with the normal relational database access methods: hashed access, direct access, sorted access. The user level roughly corresponds to the notion of a relational specification schema giving the attributes and features of a set of relations. The application programmer invokes an access routine (eg. **read_vector**) to access a tuple of a given relational type. The access routine then invokes the database routine of the appropriate type to physically access the tuple on disk.

This separation of relational access routines from database access routines is more logical than physical in several cases. There are several relational objects with implementations which join the two levels. Even in these cases, it is helpful to think of distinct layers of code.

9.1. The Database Access Level

There are three database access methods used to implement most of the relational objects above. The *hash-dictionary* access method is specifically designed to meet the requirements of the dictionary relational object. The requirements and implementation are discussed in greater detail below in the section on the dictionary relational object.

The *sorted-fixed* access method corresponds very closely to the normal sorted relation of a relational database. The relational objects which use this method all have a fixed number of attributes, are kept sorted on one or two keys, and are generally fairly small (< 2000 tuples per relation). Each attribute is of a fixed size. This access method is generally used to store results of retrievals or experiments. These relations are normally written and read sequentially, so there is no need for immediate access to a random tuple.

The small size of these relations allows them to be kept entirely in main memory as a compact sequence of tuples. Positioning within the relation is done by binary-search of the keys. Insertion of a tuple in the middle of a relation is expensive, since every following tuple must be moved in order to make room for the new tuple. No disk accesses are involved other than the initial and final reading and writing of the entire relation.

The *direct-variable* access method allows some attributes to be of variable length. There must be one distinguished attribute which assigns a unique value to each tuple within the relation. The relation is kept sorted on this attribute. Each variable length attribute must be associated with another attribute in the relation whose value gives the length of each tuple's variable length attribute. These relations are the "work-horses" of the SMART system. Most of the vital collection relations (eg. document representatives, inverted files) use this access method. The variable length attributes are used to store lists of items, like the list of concepts associated with a document. This means that an entire document representative can be read with one call to the access method.

The direct-variable relations are stored as two UNIX files. The first file (the direct file) contains the fixed-length attributes and a pointer to the location of the start of the variable-length attributes. These variable-length attributes are found in a second file. The records within the direct file are all of fixed length. This allows the distinguished attribute (*tid*) value to be used as an offset into the direct file to immediately access the fixed-length attributes of a tuple. Since the lengths of the variable-length attributes are known once the fixed-length attributes are known, the location of each of the variable-length attributes can be directly determined.

If the application programmer has specified that the entire direct-access relation should be brought into memory (by using the flag SINCORE), then a tuple can be read by simply setting a few pointers. Most of the time, though, the relation will be too big to be brought into main memory. In this case, the fixed attribute is read into one internal buffer, and all of the variable length attributes into another buffer. Pointers are then assigned to address these buffers. Therefore, if a direct object is not SINCORE, the applications program must copy any tuple it wishes to save.

9.2. The Relational Object Level

The three database access methods above are used for all of the relational objects below (although indirectly in some cases). The application programmer will probably never have contact with the database access methods. The relational objects provided are expected to accomodate most information retrieval purposes.

9.2.1. The Dictionary Relational Object

The dictionary relation is used to map a token and a type of token to a unique concept number. Collection frequency information is also kept in the dictionary. The user views the dictionary relation as a sequence of tuples of the form:

```
typedef struct {
        char  *token;          /* pointer to the actual string */
        short ctype;           /* Classification type for this token */
        unsigned short freq;   /* Number documents in which concept occurs */
        long  con;             /* unique index for this token, ctype pair */
} DICT_ENTRY;
```

The relation is sorted by <con>, but can also be accessed given both <token>

and <ctype>. Note that the attribute <con> is really of type *tid*. The user will enter a new concept by seeking on <token>,<ctype>. The number <con> is assigned to the tuple by the system when the new concept is written.

The basic implementation of the dictionary object is as a simple hash table. Collisions are handled by linking to the next available spot, following a chain of links if necessary. If there are no available spots within a fixed number of entries, an overflow hash table (same implementation) is used, and the current operation is recursively invoked upon it. Note that the hash function used does not take into account the recursive level (probably a mistake) and thus any undesirable behavior in the initial dictionary will be repeated in the overflow dictionary (very large numbers of tokens hashing to one value cause serious problems).

The internal implementation of an entry

```
typedef struct hash_entry {
        short secondary_hash;          /* entirely different hash_value
                                       /* used to avoid reference to string*/
        short ctype;                   /* ctype of this concept */
        unsigned short freq;           /* document frequency of concept */
        short collision_ptr;           /* offset of next word hashed here */
                                       /* If IN_OVF_TABLE, then need to go to
                                       /* (next) overflow hash table */
        long str_tab_off;              /* position of string (byte offset) */
} HASH_ENTRY;
```

The physical file contains 3 sections - a 20-byte REL_HEADER, a fixed size hash table, and an expanding string table. New token strings are added to the end of the string table, and the byte offset is placed in the internal entry.

Two methods are defined for accessing a dictionary entry: hashing on <token,ctype> or direct access through <con>. <con> is simply the dictionary entry index that <token,ctype> hashes into when the entry is originally placed in the dictionary. Thus, a quick direct access to the token and freq values exists given the values of <con>. This is used (possibly) during retrieval and feedback operations. There may be some similarity computations based upon the token (for example, experiments using fuzzy matching of dates), and the freq information is used extensively by feedback. Accessing via <token,ctype> is essential during the indexing process.

In practice, the dictionary implementation is very fast for read access. The collision pointer is guaranteed to point to within either the current disk page or the next disk page (except if it points to the overflow table). There don't seem to be any efficiency problems for read access until the first overflow hash table starts to get full. Write access, which is done only when adding a document to the collection normally, becomes significantly slower when the initial hash table becomes about half full. However, the parsing and other actions of the indexing process still take longer than the dictionary accesses. In addition, document indexing is not time critical; a user is not normally sitting there waiting for the indexing to finish.

Problems with the implementation: The size of the hash table is fixed (by the value of max_primary_entry in the relation header) when **create_dict** is called. The overflow dictionary works reasonably fast, but is space inefficient. Currently, if any write operations are to be done, the entire dictionary, must be brought in-core. There is no inherent reason for this; at some point in the future the implementation should be changed. A B-tree implementation would be nice, but that is difficult to program efficiently under UNIX. A problem with the current use of the dictionary by the SMART package is that every desired concept must be entered in the dictionary. For collections with unique identifiers for each document, this means that the number of terms in the dictionary will be greater than the number of documents in the collection. This fills the hash table fast!

9.2.2. The Vector Relational Object

The vector relation is used to store "flat" (no imposed structure) document or query representatives. Each tuple gives a list of concepts (with their types) and weights for one document. Thus the relation as a whole represents an entire document (or query) collection. The user view of a vector tuple:

```
typedef struct {
        long id_num;           /* unique number for this vector within
                               /* collection */
        long num_conwt;        /* no. of tuples in the vector */
        CON_WT *con_wtp;       /* pointer to concepts, weights for vector*/
        short  num_ctype;      /* number of ctypes for this vector */
        short  *ctype_len;     /* length of subvector for each ctype */
                               /* For i = 0..(num_ctypes-1),
                               /* ctype_len[i] is the number of concepts
                               /* in con_wtp for this document
                               /* with ctype i */
} VEC;
typedef struct {
        long con;              /* Actual concept number */
        float wt;              /* and its weight */
} CON_WT;
```

Note the pointer to the CON_WT relation. The relation is sorted by <id_num>, with the con_wt tuples being sorted by <ctype , con>

The vector relational object uses the *direct-variable* database access method discussed above. Since these routines bear the brunt of most of the data manipulation involved in indexing and retrieval, it is important that they be as fast as possible. The implementation performs reasonable actions with all possible access modes, both in and out of core. On a read_vector operation, the implementation will make sure that all the CON_WT tuples are in memory (if not, then they will be brought into memory) and the value of <con_wtp> is set to that location in memory. No copying of this portion of the vector is ever done.

The only real problem with the vector implementation is that <id_num> needs to be user assigned at an early stage. The SMART indexing process in general calls for the assigning of an id to a vector long before the document collection itself is opened. Therefore, holes in the document collection caused by deletion of documents cannot be filled during the indexing stage, but have to be compacted by some other means later.

9.2.3. The Display Relational Object

The display relational object stores information needed for the display of documents after retrieval. The UNIX file name and location within that file for each section of a document are kept within a display relation tuple. The user's image of a display tuple is

```
typedef struct {
        long  id_num;          /* Id of doc */
        short num_sections;    /* Number of sections in this did */
        long  *begin;          /* beginning of sections 0..numsections-1
                               /* (bytes from start of file - section 0
                               /* is the start of the document */
        long  *end;            /* end of sections 0..numsections -1 */
        char  *file_name;      /* File to find text of doc in */
        char  *title;          /* Title of doc (up to TITLE_LEN chars */
} DISPLAY;
```

The *direct-variable* access routines are used again. All of the comments pertaining to the vector relation apply here also. One optimization could be made for the display implementation which has not yet been made. Very often documents are stored sequentially in one UNIX file. Thus the filenames of all the tuples of a relation could actually be stored in one collection.

It is not clear that the title of a document needs to be stored in the display tuple since it can be obtained easily from the physical document. A lot of space can be used by storing the titles an extra time. However, an earlier implementation suggested that it is just too slow to read the titles from the documents. This is a classic time-space tradeoff; perhaps the display routines should allow for either choice.

9.2.4. The Inverted File Relational Object

The main purpose for the inverted file relations is to provide an alternative access method to a collection of documents with concepts. Instead of storing the collection as a sequential list of vectors, each vector having a list of concepts associated with it; the collection is stored as a sequential list of concepts, each concept having a list of documents associated with it. Hence, an inverted file relation is an inverse of a vector relation. An inverted file tuple looks like

```
typedef struct {
        long  key_num;         /* key to access this inverted list with*/
        short num_list;        /* Number of elements in this list */
        long  *list;           /* pointer to list elements  */
        float *list_weights;   /* Pointer to weights for corresponding */
                               /* elements */
} INV;
```

In its most common use, the key_num attribute of an inverted relation is a concept number, the list attribute is a set of documents ids (in increasing order) in which that concept occurs, and the list_weights attribute is a set of weights for those occurrences. List_weight[i] is the weight of concept key_num in document list[i]. Note that other uses of an inverted file relation are also possible.

The *direct-variable* database access method is used in the implementation. Unfortunately, one problem of this method mentioned earlier becomes much more serious here. Garbage collection of the variable-length UNIX file is not automatically done. If a new document is added to a collection, then the inverted lists for all of the terms occurring in the document need to be re-written. These new inverted lists are written at the end of the UNIX file and the space taken up by the old versions is never re-claimed. Currently, this means that the program **copy_inv** needs to be run fairly often to copy the inverted lists (and avoid copying wasted space).

9.2.5. The Simple Inverted File Relational Object

The simple inverted file relations are just a tool for the application programmer. There is no particular application for this relational object. The object is basically the same as the regular inverted file object above except that no space is reserved for weights.

```
typedef struct {
      long  node_num;          /* key to access this inverted list with*/
      short num_list;          /* Number of elements in this list */
      long  *list;             /* pointer to list elements  */
} SIMP_INV;
```

9.2.6. The Graph Relational Object

The graph relational object is also just a tool for the programmer. Graph relational objects can be used to represent a general graph with weighted edges. They are implemented using the *direct-variable* access routines.

```
typedef struct {
      long  node_num;          /* unique number for this node within
                               /* graph */
      long  info;              /* Information stored in node (often index */
                               /* into another relation or array) */
      short num_parents;       /* Number of other nodes which point to node*/
      short num_children;      /* Number of children of this node */
      long  *parents;          /* node_nums of parents of node. */
      long  *children;         /* node_nums of children of node. */
      float *parent_weight;    /* Weights of each of the links in parents */
      float *children_weight;        /* Weights of each of the links in children */
} GRAPH;
```

9.2.7. The Array Relational Object

Again, these are a general purpose tool for the application programmer. The array relation simply gives you access to an integer value, given an integer index.

```
typedef struct {
   long index;
   long info;
} ARRAY;
```

The array object just gives a uniform way of storing this type of information on disk.

9.2.8. The Top Ranks Relational Object

A top ranked relation (TR for short) stores the ranks of the top documents seen for a query. This is the output form of a retrieval operation (and the input form for the display of documents). A tuple appears as

```
typedef struct {
        long  qid;      /* query id */
        long  did;      /* document id */
        char  rank;     /* Rank of this document */
        char  rel;      /* whether doc judged relevant(1) or not(0) */
        char  action;   /* what action a user has taken with doc */
        char  iter;     /* Number of feedback runs for this query */
        float sim;      /* similarity of did to qid */
} TR;
```

The tuples are sorted by <qid,did> within the relation.

The TR relations are straightforward, without pointers and are typically quite small. They are implemented using the *sorted-fixed* database access method. One problem with the current implementation is that there is no provision for deleting or overwriting tuples. Nobody has yet come up with an application that needs this. but it is a gap in the completeness of our implementation of relational objects in general!

The major reason for having an TR relation at all is size. The experimental results of a retrieval need to be kept around long after the experiment is done. Unless attention is paid to keeping the results in a compact form, the size of the results quickly becomes larger than the size of the collection!

9.2.9. The Relevant Ranks Relational Object

A relevant ranked relation (RR for short) stores information needed for the evaluation of an experimental run. The ranks and similarity values for every relevant document seen are kept in the RR tuples. Again, the *sorted-fixed* access method is used. The user's (and system's) view of a tuple:

```
typedef struct {
        long  qid;      /* query id */
        long  did;      /* document id */
        long  rank;     /* rank of document */
        float sim;      /* similarity of did to qid */
} RR;
```

The relation is sorted by <qid,did>.

9.2.10. The Evaluation Relational Object

The eval file objects store the results of an evaluation of a retrieval run. SMART calculates a large number of different evaluation measures, mostly for historical reasons. The measures given below fall into only 4 classes, where if one measure shows improvement, the other measures in the class show improvement.

```
#define NUM_RP_PTS  21
```

```
#define NUM_CUTOFF 2
#define CUTOFF_VALUES {10, 30}
#define NUM_BETA 3
#define BETA_VALUES {0.5, 1.0, 2.0}
typedef struct {
        long  qid;             /* query id */
        float recall_precis[NUM_RP_PTS];/* Recall precision at .05 increments */
        float av_recall_precis;       /* average at 3 intermediate points */
        float norm_recall;     /* Normalized precision figure */
        float norm_precis;     /* Normalized recall figure */
        float rank_recall;     /* Rank recall value */
        float log_precis;      /* Log precision value */
        float recall_cut[NUM_CUTOFF]; /* Recall value at each of the */
                               /* doc cutoff levels (10 and 30 documents) */
        float precis_cut[NUM_CUTOFF]; /* Precision value at each doc cutoff */
        float e[NUM_CUTOFF][NUM_BETA]; /* van Rijsbergen's E measure.
                               /* calculated at each doc cutoff level for
                               /* three values of the relative importance of
                               /* recall versus precision (0.5, 1.0, 2.0) */
} EVAL;
```

9.2.11. The Pnorm Vector Relational Object

The pnorm objects are a special purpose kind of query vector. Along with the normal vector's list of attributes and weights, the pnorm vector includes a tree structure for each query. The tree structure allows boolean operators and boolean operator weights to be given in a query. Since the pnorm vectors are mainly of interest to experimenters, they are not discussed further here.

9.2.12. The Linked Inverted File Relational Object

The linked inverted file objects are commonly used to associate a list of documents (and weights) with a concept. They provide the same functionality as the regular inverted file relations, but constitute an entirely different implementation and access method. To the user, it appears that the entire relation is sorted by <con,id_num> and is composed of tuples of the form

```
typedef struct {
        long  con;          /* concept */
        long  id_num;       /* Document id that concept occurs in */
        float weight;       /* Weight of concept in this document */
} LINV_ENTRY;
```

Two logical files are defined in the implementation of the linked inverted file object. The first is a direct access file with concept number as the key. The entry here contains the first element of a linked list of document ids for this concept. The rest of linked list is in the second file. The linked list is sorted by did.

Note that the linked list approach allows easy insertion and updating, but at a substantial cost in space. This implementation shares with the vector and display implementations the nice property that garbage collection is very simple. Copying a relation by reading and writing a tuple at a time will eliminate any wasted space in the variable record file. In addition, immediately after the inverted relation is copied, all the linked list elements will actually point to the next physical location in the file. Thus, only one disk access is needed to read the entire linked list. This implies that the inverted relation should be copied right after the collection is initially formed for maximum performance.

There was a lot of debate (at least within the implementer's mind) about the linked list implementation as opposed to having an inverted tuple having a pointer to an array of all of the id_nums for a given concept. The linked list approach is much simpler to use for the user, but the performance penalty paid is substantial. Not only is extra space taken up by the links, but adding a large number of tuples at a time can be costly. Following a linked list in order to insert a new tuple can lead to a large number of page accesses. The linked inverted list was SMART's original inverted list implementation, but is no longer being used by for any application within SMART.

10. Concurrency, Consistency, and Protection

Any system that expects more than one user to access the system at once must address the issues of concurrency and consistency. Care must be taken that no user's action is affected by another user changing something in the system. Fortunately, the constraints imposed by an information retrieval system are less strict than those needed in a normal relational database. Information retrieval is primarily a read-only activity: documents are added or deleted rarely and (conceptually at least) never modified. Serializability of the read and write requests (as required by databases) is much less important since writing a record does not significantly interfere with reading a database. The information retrieval system's failure to return to the user a document which is in the process of being written is not fatal.

Write-write conflicts, on the other hand, must be guarded against. The system cannot allow the possibility of two different programs modifying the same relational object at the same time, and each then writing out its own version of the complete relation. Only the actions of the second program to finish will necessarily be found in the resultant relational object. An entire relation needs to be locked at once, since some access methods operate entirely in core.

The SMART system could have implemented concurrency checking at two distinct levels in the code. One possibility was to implement access control at relational object access level. The routines which actually open an object could also lock the relation, preventing another program from opening that relational object for writing. While this is easy to program, it still leaves open the question of what is to be done if an object cannot be opened due to another program having a lock on that object. The information that a failure occurred because of lock conflicts must be passed back to a level of the program that knows what to do. This is normally the level of the program that deals with the user.

Since the top-level routines must already know what to do in the case of concurrent access, it was decided to let the top levels be responsible for the locking as well. This way, only one level of programming needs to know anything about concurrency control. The top level programs are responsible for obtaining a write-lock for the entire database if any writing is to be done in the course of the program invocation. This write-lock excludes other programs from obtaining a write-lock, but programs which only read data can execute freely.

Currently, the top level programs lock a database for writing by creating a subdirectory "lock/WRITE" in the indexed collection directory. This method of locking was chosen because the top level programs are UNIX shell scripts and there aren't that many atomic operations at the shell script level that can be used for locking. It is quite expensive to create directories, but it has to be done so rarely that the cost is not too important.

10.1. Consistency

There are three levels of consistency that SMART needs to worry about. The first arises from the need of multiple users to have a consistent view of the database. As was stated above, SMART does not require perfect consistency at this level. It is quite possible that two users will have a different idea about which documents are in the database, even though they are accessing the database at the same time. However, the views will only differ in documents being added or deleted at that moment. These documents will be a very small percentage of documents in the collection and future modifications to the database will not be based on these documents either appearing or not appearing. It is this last difference from a normal database that allows flexibility in an information retrieval system's treatment of concurrency.

The second level of consistency to be looked at is that of the collection level of the information retrieval system. There are some constraints that one would like to hold across the relational objects of a collection. For example, if **retrieve** says that a particular document is the best match for the user's query, then it would be nice if that document were in the display relational object so it could be shown to the user. SMART makes no guarantees about this kind of consistency at any one particular point in time. SMART programs perform reasonable actions if inconsistencies between relations are discovered. It is guaranteed that the relations will be consistent if there are no write operations currently running or pending. The top level programs are responsible for rolling back any changes caused by a write transaction that aborts. The current implementation is shadowing relations: any program that changes a relation must make a copy of the original relation. The top level program will either destroy these shadows (if the transaction "commits") or restore them as the valid relation (if the transaction "aborts").

The last level of consistency covers the within relation consistency. If a write and read transaction are allowed to be going on simultaneously, it must be the case that the write transaction will not interfere with the read transaction's access of individual tuples. The implementation of the access procedures for the various relational objects must assure this. For instance, a procedure which writes a tuple can never be allowed to over-write any valid data.

As far as is known, there are only a couple of small windows of vulnerability left in SMART (in the dictionary routines, and in the collection update routines). However, since there is no scheme of locking or protection at this level, there is no good way of proving that the system ensures consistency. Judging from experience there are probably other vulnerable points. Little attention was paid to concurrency issues in the overall design of the current implementation of SMART. Enough was done to make SMART usable, but the approach was much more haphazard than it should be ideally. A model of the concurrency constraints required by an information retrieval system needs to be developed.

10.2. Protection

At the present, SMART assumes that the only sort of protection needed to assure proper access to a database can be accomplished by proper modes on the UNIX files involved. This is not a good assumption, and work needs to be done in this area. Currently, a malicious user could virtually destroy any database to which they can add a document. The destruction would be detected and could be recovered from fully, but the chore of doing so would take a long time.

11. Relational Databases and Information Retrieval

A large number of people in recent years have suggested putting an information retrieval system on top of one of the existing commercial relational database systems. There are many advantages to be gained from this, including uniform mechanism for accessing data, concurrency control, and protection features. The disadvantages are the overhead in speed and space. The previous version of SMART (Fox's) was based on such a system; the current version is not.

Fox's original implementation was entirely based on INGRES, a freely available relational system written at Berkeley. (There is a commercial version of INGRES, supported by Relational Technology, which is 2 to 3 times faster than the version we used). Fox quickly changed to a hybrid implementation, where the actual large collections of documents were actually accessed by UNIX files instead of through INGRES. All of the other information used by the system remained in INGRES.

The relational system was very efficient for experimental design. It was extremely flexible and it allowed easy, uniform viewing and manipulation of the data input and results.

Unfortunately, these easily designed experiments still needed to be run. They were tremendously slow. Operations which ideally should take about 5 seconds, were taking from 3 minutes to 45 minutes. One experimental run on a medium size database (77 queries and 12000 documents) could take several *days* to complete. The slowness of the system hampered efforts of the experimenters to perform as many experiments as they would like to.

There were several reasons for this painful slowness. For example, as was stated above, our particular relational system was 2-3 times slower than current commercial systems. Some of the data structures used in information retrieval programs were difficult to represent in a pure relational system (eg. trees). The major reason, though, was simply the number of "transactions" between the information retrieval programs and the relational database, and the overhead on each transaction. It was impossible to work totally within the relational system; data had to be exchanged between a conventional programming language program and the relational system. While INGRES offered facilities to do this easily, the cost was high.

It took much longer to set up such a transaction than it did to physically read the information from disk. Given the massive amounts of data used in information retrieval processes, this transaction overhead dominated the time needed by the overall retrieval operation.

The generality of the relational database systems is the major problem when using them to implement information retrieval systems. There is too much work to do that is not directly related to obtaining data from disks, which is the traditional constraint of information retrieval sytems. I would guess that a well-designed (from an information retrieval point of view) relational database system will still cost a factor of 5 in speed over a direct implementation (and I don't know of any such well-designed relational systems).

12. Conclusion

There is very little that is new about the current design of SMART. Instead, the standard information retrieval algorithms are implemented in an efficient and flexible manner. The core of the system is the set of low-level data access mechanisms that allow the rest of the system to look at stored information as sequences of tuples and to efficiently access individual tuples. The experimenter and database administrator are aided by a uniform approach to specifying parameter values. A rudimentary user interface exists that allows interactive help for many purposes. Concurrency issues in SMART are dealt with superficially, but in a manner that should be sufficient for most non-commercial uses of the system.

The resulting system turns out to be quite usable for both casual and experimental purposes. A casual user can submit a query and receive back the relevant documents within a couple of seconds. The experimenter can change parameters and even algorithms with minimal effort. For example, one recent investigation into term weighting schemes involved implementing several different term weighting methods. It took 1 day (about 25 hours) to implement, run, and evaluate the methods (a total of 119 experimental runs were made). This type of investigation would previously have taken a couple of weeks.

There are still a number of problems with SMART. The foremost of these is the user interface. There are clear improvements that can be made in the present interface; the need for other improvements will become obvious as the system is used by more people. Another area for improvement already discussed is that of concurrency. Both the user interface and concurrency problems stem from the gradual change of SMART from an entirely experimental system to one that can be actually used.

A number of the algorithms used in the implementation can be improved. In general, straight-forward algorithms were preferred. More complicated algorithms which are more efficient, especially space efficient, exist and should be implemented. The dictionary access procedures are a good example of this.

The number of applications for SMART will undoubtedly increase in the next couple of years. At this time at Cornell, it is being used for

1. Searching a collection of CACM abstracts

2. Providing a help facility for UNIX. There was a lot of documentation for UNIX on-line that was inaccessible because nobody could find it.

3. Accessing a user information database (interests and hobbies as well as factual information).

4. Accessing reference databases (easy, non-factual searches of standard databases of references)

5. Searching electronic mail files (eg. the old mail to system support staff)

6. Searching archives of electronic bulletin boards (USENET news)

The major changes in SMART for the next few years, though, will probably come from the addition of new methods of retrieval, information storage, and models of information retrieval. As experimental work is done, new algorithms will be implemented and added to the present core of the SMART package. At Cornell there are already a number of programs which could augment SMART (eg. clustering, probabilistic retrieval, phrasing). After they are "fine-tuned" a bit more, they will undoubtedly be added. There are still entire areas of information retrieval not covered by the current system. There is now hope for great improvements in the understanding of natural language in information retrieval contexts. Hopefully, the current system can serve as a stepping stone for further research for a number of years to come.

References

[1] J. Rocchio, "Possible Time-Sharing Orgaization for a SMART Retrieval System". Information Storage and Retrieval, Vol 7, Cornell University (1964).

[2] M. E. Lesk, "Design Considerations for Time Shared Automatic Documentation Centers". Information Storage and Retrieval, Vol 11, Cornell University (1966)

[2] M. E. Lesk, "Design Considerations for Time Shared Automatic Documentation Centers". Information Storage and Retrieval, Vol 11, Cornell University (1966)

[3] D. Williamson, R. Williamson, "A Prototype On-line Document Retrieval System" Information Storage and Retrieval, Vol 18, Cornell University (1970)

[4] D. Williamson, R. Williamson, M. E. Lesk, "The Cornell Implementation of the SMART System". In *The SMART Retrieval System*. edited by G. Salton. Prentice-Hall, Englewood Cliffs, N.J. (1971).

[5] SMART Staff, "User's Manual for the SMART Information Retrieval System". Technical Report 71-95, Revised April 1974. Cornell University (1974).

[6] E. Fox, "Some Considerations for Implementing the SMART Information Retrieval System under UNIX". Technical Report 83-560, Cornell University (1983).

[7] G. Salton, M. McGill, *Introduction to Modern Information Retrieval.* McGraw-Hill, New York. (1983).