Implementing SMART for Minicomputers Via Relational Processing with Abstract Data Types

Edward A. Fox
Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

Designed during the 1960's as a research tool for the field of information retrieval, the SMART system has been operating on an IBM 370 since 1974. SMART is now being enhanced, redesigned, and programmed under the UNIX operating system [28] on a DEC VAX 11/780. The techniques used should allow real-time operation on smaller minicomputers in the PDP 11 family. The implementation provides for a combination of database and information retrieval operations which make it applicable to office automation, personal information system management, and research studies.

The SMART vector space model, which treats information requests and stored information records as vectors in an n-space (of terms), is integrated into the relational database model using the concepts of abstract data types (ADTs). Domains of relations are allowed to be any ADT; an extended relational algebra is described with operators that manipulate many complex data structures. After illustrating the application of these concepts to typical SMART tasks, a prototype implementation is outlined. Also included is a discussion of techniques to be employed in a more efficient version.

1. Introduction

Database retrieval systems employing the hierarchical, network, or relational models typically process fixed-format records designed to represent important attributes of entities in the real world. Document retrieval systems, on the other hand, manipulate representations of stored bibliographical items such as journal articles and books.

This study was supported in part by the National Science Foundation under grant IST 80-17589.

What is needed is a synthesis of ideas that will result in a conceptually simple generalization allowing both types of entities to be represented [36]. Such a system is discussed in this report. Specifically, the relational database model is extended to allow domains of relations to be ADTs rather than the simple types originally proposed in [6]. This and other extensions are described in section 4 and illustrated in terms of a prototype redesign of the SMART system in section 5. Issues of language design, data organization, and applications of such a system are covered in section 6.

Before delving into proposed extensions, some background is needed. Systems for combining document and database retrieval are described in section 3. Familiarity with relational database concepts as given in references such as [6], [4], [9], [24] is assumed; however, no knowledge of document retrieval is expected. Some introductory information about SMART is included in section 2; for a broader overview, the interested reader is referred to [37]. More detailed accounts are given in [32], [33], and [35].

2. The SMART System

In the late 1950's, automatic text analysis began to be seriously studied. Spurred on by interest in machine translation, text processing became an important research area [2:p.113-116].

In 1961, the SMART (System for the Mechanical Analysis and Retrieval of Text) project began, with the object of designing a fully automatic document retrieval system. An automatic indexing component constructs stored representations of documents. A search component selects documents that seem most similar to a user's query. An evaluation component allows rating of the results using relevance judgements obtained from the user population, facilitating comparisons between various indexing and search schemes [37:1-2].

At the heart of the system is the concept of a vector space. Let us assume a collection C of N documents employs T terms (each specified by an integer that identifies a word, stem, phrase, or thesaurus category). If the weight (e.g. an integer measuring frequency of occurrence) of term j in document D_i is d_{ij}, the collection can be represented as:

$$C = \{ D_i \} = \{ (d_{i1}, \ldots, d_{iT}) \} \quad (1)$$

A typical collection includes many terms and many documents. Since most weights are zero, the collection C can be represented by a very sparse matrix.

Each document vector D can then be more compactly represented by considering only the n terms that are present (each called c_k) and their weights (w_k), as in:

$$D = (\langle c_1, w_1 \rangle, \ldots, \langle c_n, w_n \rangle) \quad (2)$$

If the queries are indexed in the same fashion as the documents, a correlation function such as the cosine of the angle between the vectors can be used. Thus, we compute the similarity of a particular document D_i to a given query Q_j by:

$$COS(D_i, Q_j) = \frac{\sum_{k=1}^{T} (d_{ik} \times q_{jk})}{\sqrt{\sum_{k=1}^{T} (d_{ik})^2 \times \sum_{k=1}^{T} (q_{jk})^2}} \quad (3)$$

By generating a similarity coefficient between a given query and each of the various documents in a collection, it becomes possible to produce a ranked list of documents, in decreasing order of the query-document similarity [35].

2.1. Evolution of the Batch Version

By 1964, SMART was in operation on Harvard University's IBM 7094 batch computer system. In the late 1960's, operations were transferred to an IBM 360/65. By 1974, the present IBM 370/168 batch version was completed at Cornell University.

2.2. Design and Effort toward On-Line Versions

SMART was originally intended for operation in a time-sharing environment [37:p.3]. In 1964 a preliminary design suited for such operation was suggested [29], and implementation work began for a version to run under M.I.T.'s CTSS [3]. In 1966, M.E. Lesk concluded that fully automatic documentation centers were feasible with current technology [21]. In 1970, a detailed design for a IBM 360 based conversational system was prepared [44]; later a partial implementation under the 370 TSO was undertaken.

Much of the more recent research by members of the SMART group deals with issues relating to on-line use. Clustered file organizations, where "similar" vectors are stored nearby, can be effective for real-time identification of documents relevant to a query [35:323-385]. An automatic classification procedure constructs a tree structure, in which leaves are document vectors and nodes are centroid vectors; each centroid is the vector average of the nodes immediately below. Given such a tree, a modified depth-first search finds documents most similar to a query. Searching proceeds top-down from the root; at each stage the child of the current node with highest query-centroid similarity is selected.

Feedback is also employed in automatic query improvement methods as part of an iterative search process [37]. After the system retrieves a small number of documents, the user specifies which are relevant. Based on that information, a modified query is computed that generally performs better than the initial one. The process can be repeated a number of times so that many relevant documents can be located with a minimum of user effort.

Since 1980, efforts have been made to devise a time-sharing minicomputer version of SMART on computers of the Cornell Computer Science Department. A few key components of SMART were reprogrammed to allow experimentation on a PDP 11/60. Many projects are now underway using the first of two planned VAX 11/780 computers.

3. Combining Document and Database Retrieval

As was suggested in the Introduction, methods for combining document and database retrieval are interesting and useful. Before the discussion of my particular approach, two operational systems and some proposed relational systems that do include both document and database operations are briefly described.

3.1. STAIRS

STAIRS/VS operates on IBM 370s in batch and on-line modes. For each word of text in the database, an inverted file entry contains pointers to all document occurrences of that word. Document texts are also stored and can be examined using a BROWSE command [15].

Documents of interest can be identified by Boolean SELECT queries using the inverted index, or by standard database SEARCH queries dealing with separate formatted fields. Using the SELECT command, a portion of the collection can be isolated with a request like 'PUBLISHER=MCGRAW-HILL,ADDISON-WESLEY AND YEAR > 1975'.

3.2. FIRST

Robert Dattola designed FIRST at Xerox Corporation on a Sigma 9 computer. It employs a vector space model (based on SMART ideas) for the document retrieval portion and uses a CODASYL DBMS for bibliographic information. The two components are integrated and queries containing both types of information can be processed. Optimal retrieval paths are automatically selected; thus, handling of broad bibliographic queries is by clustered file search. Specific bibliographic requests are satisfied by navigating through the network data organization. Sequential ranking of the selected documents follows, using a cosine similarity function [10].

3.3. Proposals for Use of Relational Systems

Relational systems handle database retrieval tasks with operations on a single data structure called a relation. Codd originally proposed the relational algebra and the relational calculus to express a desired data manipulation of one or more relations non-procedurally [6]. The use of the relational model may be more easily adapted to document retrieval tasks than the network model used in FIRST.

Ian Macleod considers the use of the SEQUEL relational language for handling both the bibliographic database and content term aspects of an information retrieval system called Mistral/11 [24]. Robert Crawford describes further how the relational model can be employed for document retrieval purposes [8]. However, he feels that for output processing, "factoring" is needed to go beyond the normalization rules and allow printing a repeating group that is stored as a number of tuples. Crawford also uncovers a number of problems such as the handling of weighted retrieval, where a special function must be provided to enable ranking of documents against a query. An appropriate solution is needed, such as a suitably general extension to the relational model.

4. Extending the Relational Model

At present, appropriate techniques for handling combined document and database retrieval tasks are not available. Consider the task of indexing an abstract. Though an abstract can be treated as one of the domains of a relation [8], it is hard to do anything further with it. Ideally, one would like to transform a document relation DOC_REL (doc#, abstract) directly into an equivalent relation such as ABS_REL(doc#,word#, word), where "word#" gives the position of "word" in the "abstract".

Second, consider the problem of weighted retrieval. The importance of a term (term#) in a document (doc#) can be measured by a real-valued weight (dwt), as can the importance of a term in a query (qwt). How does one apply a similarity function to relation R (doc#,term#,qwt,dwt) to get S (doc#,sim) where "sim" is the cosine of the query and document weight vectors?

Third, examine the difficulties that relational models have with other types of data. One would like long strings like abstracts to be printed in a reasonable fashion. Further, to allow an array processor to access a specialized form of document vectors, one needs to arrange the data storage as a form of un-normalized relation, so the array processor can easily manipulate large blocks of the vector space [38]. Other special data structures should likewise be integrated into a relational file system so that separate access methods are not required; examples are: graphics lists for display processors, parse trees for stored programs segments, and tries for efficient dictionary searching.

Finally, consider the problems connected with programming a depth-first clustered tree search [33:224-242]. Tree handling primitives are the natural tools to employ; they must be inefficiently simulated when using relational data structures.

These issues are discussed in the following subsections, and a prototype design of SMART along the ideas of section 4.2 is given in section 5.

4.1. Minicomputers and ADTs

To be truly effective, minicomputer operating and database systems should have elegant and simple designs, like that of UNIX [28]. There is no room for a proliferation of software tools or a plethora of file and data handling methods. Much as the relational model has streamlined thought on database manipulation, the use of ADTs has suggested that program development can proceed in an orderly, modular fashion. Thus it seems wise to apply concepts of ADTs to minicomputer design projects.

ADTs are encapsulations of operations and data types. Thus, we think of arrays, queues, and trees as abstractions for handling convenient organizations of data. In programming languages that handle ADTs, a module would exist for each such type, and only the operations specified in that module could be applied to elements of that type [30, 14, 22, 12, 13, 1].

Many relational database systems have been developed (e.g. those described in [42] and [5]). Though there are semantic problems with any record oriented model [17], suggested solutions have been proposed for the relational model [41, 7]. Based on formal definitions as in [23], several programming languages have been developed that manipulate relations as a basic construct [39, 40]. Of particular interest, however, are languages such as PLAIN [43] and RIGEL [31], which handle ADTs, have relations as a data type, and are implemented on minicomputers.

4.2. Relations and ADTs

Ledgard and Taylor point out that database models and programming language handling of ADTs are both attempts to handle data abstraction [20]. Paolini discusses how ADTs can be applied to some issues of database research [27]. In this study, ADTs and relations are combined in a general fashion to provide a powerful tool for programming and database manipulation.

First, an instantiation of an ADT (called an ADTI or "abstract data type instance") may be either temporary or permanent. When the CREATE command is used to bind a name to an initially empty instance of a completely specified ADT, it is necessary to indicate whether that ADTI should be given permanent storage. The default policy for relations is temporary allocation. Thus, applicative expressions, where one operation is applied to the results of another as in "SELECT(PROJECT ...)", can be easily handled. An implementation on typical computers can use a backing store for permanent and for large temporary ADTI's. Some ADTIs could be implemented as "views" in a more sophisticated implementation.

Just as record types can be defined in many languages to include references to other record types, relations should be able to have any abstract data type for each domain. Thus, if we define an abstract data type D_VEC for document vectors, a corresponding ADTI might take the form of a compressed vector like that of equation (2) in section 2. A document collection would then be compactly represented as a relation COLLDVEC(DOC#, D_VEC).

Since relations are useful for many purposes, general conversion operators (e.g. operators 4 and 5 of section 5.1) can be provided for transforming data between the relational form and the form of other ADTs. The operators can refer to functions pre-defined in a library or added later by users; only a "to-relation" and a "from-relation" function would be needed for each high level ADT.

The above generalizations have many further implications. Relations can be embedded in other relations. Trees or networks of relations could then be manipulated without the specialized system mechanisms of [7]. With proper ADTs for texts, one could easily go from collections of abstracts, titles, and bibliographic items to vectors, relations, dictionaries, thesauri, and other constructs in a single integrated system, as illustrated by the examples of section 5.

One may object that such a scheme sounds interesting, but could never be efficiently implemented. Hence, after the utility of these proposals is demonstrated in the next sections, issues of implementation will be addressed in section 5.4.

4.3. Applications of the Extended Model

Numerous applications are possible for the generalized relational facility being proposed. Two will be given here; a more detailed example appears in sections 5.2 and 5.3.

First, consider a manufacturing facility that constructs special entities, called assemblies, from parts. Parts are sometimes made from other parts. For engineering purposes, an "explosion" or part tree is desirable, and should be graphically portrayed. Hence one might have an attribute of an assembly called PART_TREE, of ADT TREE. Though for most manipulation purposes the relation ASSEMBLIES (ASSEMBLY#,PART#,SUBPART#) would be adequate, conversion routines could change to and from ASSEMBLIES and EXPLOSION (ASSEMBLY#, PART_TREE).

Consider next a tracking station receiving data from a spacecraft's TV camera. The information might be stored in a relation PIC_ELEMENTS (PIC#,XCOOR,YCOOR,COLOR,INTENSITY). However, an image enhancement device or display system might want direct reference to a formatted matrix P_MAT for each given picture, so conversion would be made to PIC_MATRIX (PIC#,P_MAT).

5. Prototype Design of SMART

The wisest tactic to use in converting the SMART system to another computer is to have a general tool of the kind introduced in sections 4.2 and 4.3 above. Section 5.1 defines the operators needed, 5.2 gives a sample schema appropriate for an enhanced SMART system, and 5.3 describes how some simple functions of SMART might be coded using the defined operators. Finally, section 5.4 mentions some of the issues involved in an actual implementation.

Throughout part 5, reference will be made to a library application in which users might want to find articles about information science occurring in several different collections. The following example and explanation describes a simplified version of the more detailed schema specified in section 5.2.

databases

DATABASES			
coll#	subjects	collc	
1	lattice group	cit-1	dv-1
...	...	...	...

cit-1

CITATION		
doc_id	title	year
%10001	Groups and Group Theory	1978
...	...	...

dv-1

DOC_VEC	
doc_id	d_vec
%10001	<group,2>, <theory,1>
...	...

Assume that an instantiation "databases" of relation DATABASES (coll#,subjects,collc) contains the entire store of data. "subjects" is of a type for listing character strings which mention all the subjects covered in a particular collection. "collc" is a record of two relations containing all the information in the collection. One such relation, CITATION (doc_id, title, year, ...), would have details for bibliographic identification of documents with internal identifier "doc_id"; a possible instantiation "cit-1" is shown above. Another relation, DOC_VECS (doc_id, d_vec), would describe documents by storing indexing information in document vector form under the attribute "d_vec". The sample instantiation "dv-1" shows a document vector where the actual word stem is given instead of the concept number that normally designates it.

Using this structure, a user could submit one request to select each collection such that "information science" is in "subjects". The same query could then automatically search those collections to find documents about particular topics published in a certain range of years.

5.1. Operators

In describing the various operators, a two stage approach is adopted. Some general characteristics pertaining to all operators are given. Then, the role of each class of operators is given and the particular operators are defined.

Parameters are of a particular type, or have their types passed by other parameters. No attempt is made to strictly define each ADT, since an axiomatic approach should be employed and that is beyond the scope of this report. Some general functions are introduced, and the ADT "relation",

along with operators that deal with it, is described in particular.

Almost all the operators given here yield a relation as a result. They can thus be used in an applicative fashion. Though other constructs would be needed to describe a full programming language using these operators, the following gives the high-level syntax of the subset being described.

<PROGRAM> ::= { <STATEMENT> }$^{+}$.

<STATEMENT> ::= { <EXP> **|** }* <EXP> ;
where the **|** indicates that the value of the previous expression is "piped" into the following expression, in lieu of its first argument (usually, REL1). The output of such a "filter" can be piped into a subsequent expression, etc. The value of the statement is the value of its last expression.

<EXP> ::= <OP_EXP> [GIVING <BOUND_NAME>]
where the optional clause allows binding of a name (previously specified by a CREATE operation) to the result of <OP_EXP>.

<OP_EXP> ::= <any_operator_form_below>

Consider now the definitions of the various types of operators. As is usual, braces followed by the superscript * or + designate a number of occurrences. Bold face braces indicate that a decision must be made as to one of the possibilities that are separated by "|"; note that "**|**" is distinct from the "**|**" used above. Brackets denote optionality.

REL1 and REL2 usually designate the name of a relation. Except in cases like operators 1-2 where such use would be illogical, RELn could instead be replaced by the form "(<EXP>)".

ATTR designates an attribute name. FUN will be used for a function, VAR for a variable, and VAL for a value. XFILE is the type of an external name referring to an I/O device or backing store file; defaults exist for input and output.

The first set of operators deals with definitions and interactions relating to the language and system environment.

(1) CREATE <REL1>
WITH ({ (<TYPE>:<ATTR>) }$^{+}$)
[SURR <ATTR>] [PERM]
creates an initially empty relation. The SURR clause indicates that the relation models entities that are defined even if key values change. Thus, a surrogate identifier would be useful for each individual in a "person" relation if for some reason the employee number key field had to be re-numbered. The system generates an internal identifier of type SURROGATE whenever a tuple is added. The PERM clause forces the relation to be held in permanent rather than temporary storage.

(2) ENTERIN <REL1> FROM **{**
TUPLE { (<TYPE>:<VAL>) }$^{+}$ |
INPUT [<XFILE>] **}**
enters either a single tuple or an external file of tuples.

(3) OUTPUT <REL1> TO [<XFILE>]
[OUT_TRANS { (<FUN>:<ATTR>) }$^{+}$]
[HEADER { (<HDR_DESCR> <ATTR>) }$^{+}$]
outputs the entire contents of REL1 to the external file, where the standard transformations specified in the definition of each attribute's ADT is automatically employed unless overridden by explicit mention. Functionals like "floatf" could be applied to parameters specifying output width and number of fractional places; thus "floatf(6,2)" could be applied to a number like 12.3456 to yield " 12.34". Special operators could print entire document vectors spread over several lines. The HEADER option overrides the default heading content and size for an attribute.

Other such operators would also be needed but are omitted for the sake of brevity.

The second set of operators deal with ADT conversions. They are very general and presume that the actual transformations will be done by functions. Those functions would employ various relational operators and various operators on other data types.

(4) COMBINE <REL1>
CHANGING ({ <ATTR> }$^{+}$)
INTO <ATT_TYPE>:<ATT_NAME>
USING <FUN>
groups REL1 by the attributes not given in the list (as in Codd's PATT [7]) and combines tuples in each group so a single ADTI records the data of the chosen attributes of tuples in the group. That ADTI is named ATT_NAME in the result relation and is of type ATT_TYPE; the result relation has the remaining uncombined attributes of REL1 as well.
For example, in the expression
COMBINE relx CHANGING (term_id, t_freq)
INTO SF_VECTOR:d_vec
USING rel_to_sfloat_vec
a function "rel_to_sfloat_vec" could convert a group of tuples all having the same document number (doc_id) but having different concept (surrogate term_id) and weight (floating t_freq) values into a single tuple consisting of "doc_id" and "d_vec" (a document vector consisting of surrogate-floating pairs).

(5) EXPAND <REL1> YIELDING
({ <TYPE>:<ATTR> }$^{+}$)
FROM [<TYPE>:] <ADTI_ATTR>
USING <FUN>

has the opposite effect of a COMBINE operator: expanding a single tuple containing the given ADTI to a group of tuples with the same values on the other attributes of the original relation. A special function sfloat_vec_to_rel could convert back a relation with attributes doc_id and d_vec to one where d_vec is replaced by term_id and t_freq.

These two operators are the key to having relations with non-simple domains.

The third set of operators covers the relational algebra:

(6) SELECT <REL1> WHERE
<BOOLEAN-VALUED-EXP-USING-ATTR-NAMES>

(7) PROJECT <REL1> OVER ({ <ATTR> }$^{+}$)

(8) JOIN <REL1> WITH <REL2>
[WHERE <BOOLEAN-VALUED-JEXP>]
has JEXP denoting a join expression relating an attribute A1 of REL1 and an attribute A2 of REL2; A1, A2 are over a common domain. The WHERE clause can be omitted to obtain the cross-product of REL1 and REL2.

(9) E_JOIN <REL1> WITH <REL2> OVER <ATTR>
[WHERE <ATTR1>=<ATTR2>]

(10) DIVIDE <REL1> BY <REL2>

(11) SET_OP <REL1> { UNION | DIFF | INTERSECT }
<REL2>

These algebraic operations follow standard terminology.

The fourth set of operators facilitates manipulation of domain and tuple entries:

(12) WIDEN <REL1> ADDING
({ <TYPE>:<ATTR> "<-" <VAL-EXP> }$^{+}$)

(13) UPDATE <REL1> CHANGING
({ <ATTR> "<-" <VAL-EXP> }$^{+}$)

These first two operate on a tuple at a time, adding a new attribute or updating an existing one with the value of the expression. That expression is in reality an arbitrary function over constants, variables, and the other attributes of the tuple.

(14) SUMMARIZE <REL1> COMPUTING
({ <ATTR> [INITIAL <VAL-EXP>
UPDATING <VAL-EXP>] }$^{+}$) yields a relation made up of a single tuple with the attributes listed. If the INITIAL ... UPDATING clause is absent for an attribute, then that ATTR should have constant value over all tuples. When it is present, the clause prescribes an order-independent computation over the given domain, such as summing. Thus one could project from the "citation" relation the attribute "year", and summarize those values into an average by adding them together and then later dividing by the cardinality of the relation.

The final set of operators includes some that are handy for typical applications.

(15) SORT <REL1>
{ { ASC | DESC } ON <ATTR> }*

(16) UNIQ <REL1>
reduces the relation to have only distinct tuples (as in the formal definition of relations).

(17) NUMBER <REL1> ADDING <TYPE>:<ATTR>
adds an attribute that gives the ordering of the tuples, so that ATTR for the i-th tuple has the value of the i-th element of the domain of TYPE.

This set could easily be extended. Sorting and handling of uniqueness are of critical importance. The NUMBER command is simply suggested as an alternative to requiring special syntax such as using CURRENT, NEXT, PREVIOUS, FIRST, and LAST to deal with tuple orderings.

Various other operators could be added to these five sets. However, these seem to cover the important cases, and there is adequate flexibility to add special functions for other situations. A complete language description would include similar groups for other ADTs, but that is beyond the scope of this report. A general operator to apply functions operating on any ADTs will suffice to round out this version of the operator set.

(18) <FUN> ({ <ARGUMENT> }$^{+}$)
is appropriate for arguments of type relation and of other types.

The functions given in this section as operators simply provide syntactic sugar to facilitate use. Other, user defined functions, would be utilized by employing the function application operation.

5.2. Sample Schema

A schema corresponds to definitions of newly defined data types and an association of names with types. Instead of giving type definitions axiomatically as should be done, they are given here descriptively. Some are in terms of already defined types; the rest rely upon prose explanation of the operators that would be applicable. Some of the definitions are polymorphic; parameters are employed to indicate the types of certain elements that make up each such generalized composite type.

Basic types of data include SURROGATE (system generated identifier), INT (integer), FLOAT (real), CHAR (single character), STRING (string of characters), and BOOLEAN (TRUE or FALSE). ARRAYs, RECORDs (as in PASCAL), LISTs (where one can insert, read, and delete elements at any position), and RELATIONs would be predefined composite types.

Other composite types are built up from basic or composite types. Since all bound names are defined in CREATE operations, the language is amenable to static type checking, and the type of a bound name can be fully specified. Expressions are type checked in advance, but operators are polymorphic; thus the E_JOIN operator only

requires that REL1 and REL2 are relations such that their common ATTR has equality defined for it - other aspects of the type structures of the relations are immaterial.

The following ADTs are defined in terms of the other data types:

(1) STRING_LIST = LIST (STRING)
where strings can be added at the end of the list or looked up.

(2) TREE(POS,X) = tree with leaves of type X that can be manipulated by operators for tree creation, traversal, and modification. Typically, X will be a record containing a node type and an item number for some item of data. POS will only be operated on by routines of this ADT; it is a type for tree positions in the tree, and allows several processes to simultaneously be searching the tree since each can remember its own POS.

(3) TRIE = tree-like data structure whose branches are labelled with letters of the alphabet, so STRINGs can be quickly added and looked up.

(4) VECTOR(X,Y) = ARRAY (RECORD (X,Y))
where is_entry(X) indicates if a record is present, add_entry(X,Y) adds one, and get_entry(X) returns the desired item of type Y.

Our example will have entities for each collection, document, author, content term (either stem, phrase, or thesaurus category), and publication. The following data types are specialized for the application in mind. It is presumed that they were defined earlier by suitable CREATE statements; here is a summary showing the type structure and binding of names. To eliminate confusion, all bound names are given in lower case.

```
DATABASES:databases = RELATION (
    SURROGATE:coll_id, STRING:coll_name,
    STRING_LIST:subjects,
    DB_CONTENTS:db_contents )
DB_CONTENTS = RECORD ( ABSTRACTS:abstracts,
    AUTHORS:authors,
    CENTROID_VECS:centroid_vecs,
    CITATION:citation, CLUSTER:cluster,
    DICTIONARY:dictionary,DIRECTORY:directory,
    DOC_NOS:doc_nos, DOC_VECS:doc_vecs,
    SOURCE:source, STOP_WORDS:stop_words,
    THESAURUS:thesaurus)
Q_COLLS:q_colls = RELATION ( INT:q_coll_no,
    STRING:q_coll_author, Q_COLL:q_coll)
Q_COLL = RECORD ( QUERIES:queries,
    QUERY_VECS:query_vecs,
    RELEVANCE_JUDGEMENTS:relevance_judgements)
ABSTRACTS = RELATION ( SURROGATE:doc_id,
    STRING:abstract)
AUTHORS = RELATION ( SURROGATE:author_id,
    STRING:auth_name)
CENTROID_VECS = RELATION ( INT:centroid_no,
    VECTOR(SURROGATE:cv_id,
           FLOAT:cv_val   ):c_vec)
CITATION = RELATION ( SURROGATE:auth_id,
    SURROGATE:doc_id, SURROGATE:pub_id,
    STRING:title, INT:vol, INT:no,
    INT:start_page, INT:year)
CLUSTER = TREE(POS:node,
    RECORD(INT:cent_or_doc,
           INT:vec_no     ):c_val )
DICTIONARY = RELATION ( SURROGATE:term_id,
    STRING:stem)
DIRECTORY = RELATION ( SURROGATE:author_id,
    STRING:auth_name)
DOC_NOS  = RELATION ( SURROGATE:doc_id,
    INT:doc_no )
DOC_VECS = RELATION ( SURROGATE:doc_id,
    VECTOR(SURROGATE:dv_id,
           FLOAT:dv_val   ):d_vec  )
SOURCE = RELATION ( SURROGATE:pub_id,
    STRING:publication_name,
    STRING:publisher)
STOP_WORDS = TRIE
THESAURUS = RELATION ( SURROGATE:th_cat_id,
    SURROGATE:term_id,
    FLOAT:prob_of_term_in_cat)
QUERIES = RELATION ( INT:q_no, STRING:query )
QUERY_VECS = RELATION ( INT:q_no,
    VECTOR(SURROGATE:qv_id,
           FLOAT:qv_val   ):q_vec  )
RELEVANCE_JUDGMENTS = RELATION ( INT:q_no,
    INT:doc_no, BOOL:relevant? )
```

5.3. Definition of Search and Other Operations

Consider now various typical operations:

(1) Dictionary creation by getting all words from abstracts that are not in the stop word list, stemming them, sorting them to unique entries, and assigning surrogate term numbers:

```
EXPAND abstracts
   YIELDING (INT:word#, STRING:word)
   FROM abstract USING text_cracking
   GIVING doc_words  |
PROJECT OVER (word) |
SELECT WHERE NOT
   ( in_trie( word, stop_words ) ) |
UPDATE CHANGING (word <- stem(word) ) |
SORT ASC ON word |
UNIQ |
WIDEN ADDING
   ( SURROGATE:term_id <- next_surr() )
   GIVING dictionary
```

(2) Indexing documents using "doc_words" and "dictionary" from (1), where frequency is found by adding ones and vectors are formed by "rel_to_sfloat_vec":

```
PROJECT doc_words OVER (doc_id, word) |
SELECT WHERE
   NOT( in_trie( word, stop_words ) ) |
UPDATE CHANGING (word <- stem(word) ) |
WIDEN ADDING (FLOAT:freq <- 1.0 ) |
COMBINE CHANGING (freq) INTO FLOAT:t_freq
   USING + |
E_JOIN WITH dictionary OVER stem |
PROJECT OVER (doc_id, term_id, t_freq) |
COMBINE CHANGING (term_id, t_freq) INTO
   VECTOR(SURROGATE,FLOAT):d_vec
   USING rel_to_sfloat_vec
   GIVING doc_vecs
```

(3) Indexing queries by word stems: as above, but starting with "queries" and resulting in "query_vecs"

```
(4) Ranking of all docs for each query by
  computing the cross-product of doc and
  query relations and then finding the cos
  correlation of the vectors:
   E_JOIN doc_vecs WITH doc_nos OVER doc_id |
   PROJECT OVER (doc_no, d_vec)
      GIVING dno_vecs |
   JOIN WITH query_vecs |
   WIDEN ADDING
      (FLOAT:corr <- cos(d_vec, q_vec) ) |
   PROJECT OVER (q_no, doc_no, corr) |
   SORT DESC ON corr, ASC ON q_no,
      ASC ON doc_no   GIVING ranking

(5) Selecting only the 10 top ranked docs.
  for above queries by making the group of
  results for a query into a relation that
  can be ranked and selected from, and later
  converting back to the original relation's
  format.  Note use of COMPILE, COMPOSE.:
   COMBINE ranking
      CHANGING (doc_no, corr) INTO
      RELATION(INT:rank,INT:doc_no,
               FLOAT:corr     ):top_doc_list
      USING COMPILE("SELECT WHERE RANK <= 10"
                    COMPOSE
                    "NUMBER ADDING INT:RANK"
                    COMPOSE   group_to_rel)
   EXPAND YIELDING (INT:doc_no, FLOAT:corr)
      FROM RELATION(INT:rank,INT:doc_no,
                    FLOAT:corr):top_doc_list
      USING rel_to_group GIVING top_ranking

(6) Print document collection with standard
  headings:
   OUTPUT dno_vecs

(7) Print the average correlation for the
  results of (5):
   PROJECT top_ranking OVER (corr) |
   WIDEN ADDING ( INT:count <- 1 ) |
   SUMMARIZE COMPUTING
      (count INITIAL 0  UPDATING count,
       corr INITIAL 0.0 UPDATING corr ) |
   UPDATE CHANGING ( corr <- corr/count,
                     count <- count/10 ) |
   OUTPUT HEADER ("NO. OF QUERIES" count,
        "AV. CORR. OF TOP 10 DOCS" corr )

(8) Construction of feedback queries by
  positive feedback, so new query = (old
  query) + (av. of rel. docs. of rank <= 10):
   E_JOIN top_ranking
      WITH relevance_judgements
      OVER (q_no, doc_no) |
   PROJECT OVER ( q_no, doc_no ) |
   E_JOIN WITH doc_nos OVER (doc_no) |
   PROJECT OVER ( q_no, doc_id ) |
   E_JOIN WITH doc_vecs OVER (doc_id) |
   PROJECT OVER ( q_no, doc_vec ) |
   COMBINE CHANGING (doc_vec)
      INTO (VECTOR(SURROGATE,FLOAT):av_vec
      USING vec_average |
   E_JOIN WITH query_vecs OVER q_no |
   UPDATE CHANGING
      ( av_vec <- vec_sum( av_vec, q_vec ) )

(9) Construction of feedback queries with
    negative feedback: like (8) above, but use
    set DIFF to get non-relevants of rank <=10.

(10) Centroid search via predefined function:
     cent_search ( query_vecs, cluster,
        centroid_vecs, doc_vecs, doc_no )
```

5.4. Implementation

Parts of SMART have been implemented using the C language [18] and features of the UNIX operating system [28]. UNIX has been used elsewhere as the basis on which to build a programmer's workbench [16]; it is especially well suited for program development. The command language of UNIX lets one invoke commands via C-like syntax, and enables "piping" the results of one program (or command file) into another in applicative fashion. Further, the C-shell command language can be invoked from inside a C program. The result is that many of the 18 operators of section 5.1 have been implemented in part using combinations of existing programs and procedures. A side effect is that all programs are small and require little main memory - important characteristics for minicomputer based systems. Let us consider the current plans for completing a prototype implementation of SMART.

UNIX text files are used to store all relations. Only string data types are stored in those files. Special characters are used, however, as prefixes to strings of numbers to indicate that they are surrogates. Following [7], there is an entity relation to connect relation-surrogates with actual file names. Other entity relations are present for other data types. Basic data types are coded into character strings, as are some composite types. Special data types that are manipulated by procedures are stored in internal form in a separate file for such items; a surrogate will serve as key to the hashing function that locates such entries. Routines already exist to manipulate term vectors and Boolean query strings; using surrogates to identify them will enable those data types to be elements of other types.

As long as only character strings are present in relations, a number of programs can be easily adapted to manipulate them. Operators 2-3, 6-7, 9, and 11-18 can thus be easily implemented, if certain limitations are made on the functions and expressions allowed. After a mechanism is programmed to create and keep descriptions of relations and special data types, operator 1 can be coded and some of the above can be revised to better meet their specifications. The rest of the operators will require more development, but (inefficient) first versions are not especially difficult to devise.

Before widespread use is made of the system, type checking logic must be added, so operations will not accidentally be applied to data of the wrong type. Several relations such as the following will be accessed by the compiler to see what is permissible:

```
TYPES ( TYPE_NAME, NEEDS_SURROGATE?,
   NEEDS_FILE?, USES_INT_FORM? )
TYPE_OPS ( TYPE_NAME, OP_NAME, INFIX_USE?,
   FUNCTION_TO_USE )
FUNCTIONS ( FUN_NAME, CAN_OPTIMIZE_THROUGH?,
   NO_OF_ARGS )
FUN_ARGS  ( FUN_NAME, ARG_NO, ARG_TYPE )
CONVERSIONS ( TYPE_1, TYPE_2,
   CONV_FUN_FOR_1_TO_2 )
```

One of the departmental VAX systems will be employed for further research and experimentation, while a production version of SMART can be made available on the second VAX computer. The thoroughly tested operational version can serve as a retrieval system for users. Collections of documents on computer and information science could be made available and the members of the Computer Science Department would be able to access those files for academic and research purposes.

6. Future Plans

Many interesting problems will remain after the initial implementation is complete. They are classifiable into the following areas.

6.1. Language Design

An axiomatic definition of all the operators employed is certainly of use, so that each system-defined ADT can be properly verified. Since C does not even have all the basic types being considered, effort is needed to define a suitably general language that can be compiled into C in an efficient manner. Of course, thought will be given at that time to see if other higher level language available under UNIX could be more easily adapted to the desired purposes.

6.2. Data Organization

In System R, considerable care is taken to compile data manipulation statements into efficient access methods [5]. A reasonable implementation of SMART would have to progress beyond utilizing only character string files for relations. Special routines would be needed to handle relations and to allow rapid access through (invisible) index files, especially where frequent joins are needed. In this regard, we plan to first examine other relational systems [31, 43] implemented under UNIX, to see if they can be properly generalized.

In any case, with the new operators included, interesting questions arise on how optimization can best be carried out. Further, there is need to explore how clustering in relations can be performed and how clustered orderings can be maintained in a dynamic environment. Finally, issues of locking, backup, recovery, and distributed processing are of interest when data types are viewed through this new perspective. Schemas and ADTs are interrelated, and so these problems must be analyzed on a low level as well as on the usual high level.

6.3. Application in Office Information Systems

As shown in [34], automatic information retrieval methods work about as well as systems where documents are carefully indexed by hand. Einar Nodtvedt demonstrated that letters can be analyzed and indexed by computer through methods similar to those used for documents [26]; reasonable retrieval performance results. At Cornell, one collection of electronic mail has been obtained. Typical user questions deal with combinations of content issues and database issues, as can be seen from the following edited versions of lists submitted to me:

CATEGORIES

Urgent Items	DEC Full Time Employees
Items Due Today	DEC Part Time Employees
DEC Rented Software	DEC purchased Software
DEC Hardware Budget	General Expense Budget
DEC Software	Incoming Correspondence
IBM Software	Outgoing Correspondence

QUESTIONS

This letter: from whom, when sent, how urgent, how many characters?
Person P: what mail (sent/received), what was sent (since a given date), no. of letters (sent/received)?
Date: no. and/or list of mail received (answered/unanswered/total) since then, between then and another data?
List of topics: which were mentioned during dates, which authors discussed, which letters were about (any/all) of them?
Greatest no.: letters (to/from)?
How long: since received/sent to P, since discussed topic T?

These and many other questions can be handled by the combined database and document retrieval system being described. Automatic storage and retrieval of electronic mail and machine readable correspondence seem to be ideal and practical applications of the methods being discussed here. They are an important component of any office automation system.

7. Conclusions

Faced with the need to convert the SMART information retrieval system to a VAX or PDP 11 minicomputer, expanding the concepts of ADTs to the database realm provides a general solution. An integrated language for handling ADTs must include a set of operations for type "relation" and for transforming between relations and other types. Eighteen such operators are defined and solutions for typical retrieval tasks are given using those operators to manipulate a sample schema. The command expressions employed provide a handy user-oriented version of important functions dealing with relations. A few simple commands allow rapid conversion from the uniform view characteristic of relational languages to the specialized data structures that are so much a part of other approaches to abstraction.

Issues of implementation, language design, and data organization are discussed in relation to this proposal. A prototype system is under development and a more efficient version should provide a versatile database and document retrieval facility for minicomputers. Perhaps such a flexible tool will be part of the office of the future.

REFERENCES

(1) Boehm, H., A. Demers, and J. Donahue. 1980. An Informal Description of Russell (Technical Report 80-430.) Ithaca, New York: Cornell University Department of Computer Science.

(2) Borko, Harold and Charles L. Bernier. 1978. Indexing Concepts and Methods. New York: Academic Press.

(3) Cane, Mark. 1964. Adapting SMART to the M.I.T. Compatible Time-Sharing System. Information Storage and Retrieval. (Scientific Report No. ISR-8.) Ithaca, New York: Cornell University Department of Computer Science.

(4) Chamberlin, Donald D. 1976. Relational Data-Base Management Systems. ACM Computing Surveys. 8:1.43-66.

(5) Chamberlin, D.D., M.M. Astrahan, W.F. King, R.A. Lorie, J.W. Mehl, T.G. Price, M. Schkolnick, P. Griffiths Selinger, D.R. Slutz, B.W. Wade, and R.A. Yost. 1981. Support for Repetitive Transactions and Ad Hoc Queries in System R. ACM Transactions on Database Systems. 6:1.70-94.

(6) Codd, E.F. 1970. A Relational Model of Data for Large Shared Data Banks. Communication of the ACM. 13:6.377-387.

(7) Codd, E.F. 1979. Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems. 4:4.397-434.

(8) Crawford, Robert G. 1981. The Relational Model in Information Retrieval. Journal of the American Society for Information Science. 32:1.51-64.

(9) Date, C.J. 1977. An Introduction to Database Systems. Second Edition. Reading, Massachusetts: Addison-Wesley Publishing Company.

(10) Dattola, Robert T. 1979. FIRST: Flexible Information Retrieval System for Text. Journal of the American Society for Information Science. 30:1.9-14.

(11) Hall, Patrick A.V. and Geoff R. Dowling. 1980. Approximate String Matching. ACM Computing Surveys. 12:4.381-402.

(12) Geschke, Charles M., James H. Morris Jr., and Edwin H. Satterwaite. 1977. Early Experience with Mesa. Communication of the ACM. 20:8.540-553.

(13) Gries, David and Narain Gehani. 1977. Some Ideas on Data Types in High-Level Languages. Communication of the ACM. 20:6.414-420.

(14) Guttag, John. 1977. Abstract Data Types and the Development of Data Structures. Communication of the ACM. 20:6.396-404.

(15) IBM Corporation. 1979. Storage and Information Retrieval System for OS/VS (STAIRS/VS), Program Reference Manual. (Publication SH12-5400.) White Plains, New York: IBM Corporation.

(16) Ivie, Evan L. 1977. The Programmer's Workbench - A Machine for Software Development. Communication of the ACM. 20:10.746-753.

(17) Kent, William. 1979. Limitations of Record-Based Information Models. ACM Transactions on Database Systems. 4:1.107-131.

(18) Kernighan, Brian W. and Dennis M. Ritchie. 1978. The C Programming Language. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

(19) Lacroix, Michel and Alain Pirotte. 1980. Associating Types with Domains of Relational Data Bases. Brodie, Michael L. and Stephen N. Zilles, eds. Proceedings of Workshop on Data Abstraction, Databases and Conceptual Modelling. Association for Computing Machinery, Inc.

(20) Ledgard, Henry F. and Robert W. Taylor. 1977. Two Views of Data Abstraction. Communication of the ACM. 20:6.382-384.

(21) Lesk, M. E. 1966. Design Considerations for Time Shared Automatic Documentation Centers. Information Storage and Retrieval. (Scientific Report No. ISR-11.) Ithaca, New York: Cornell University Department of Computer Science.

(22) Liskov, Barbara, Alan Snyder, Russell Atkinson, and Craig Schaffert. 1977. Abstraction Mechanisms in CLU. Communication of the ACM. 20:8.564-576.

(23) Lockemann, Peter C., Heihrich C. Mayr, Wolfgang H. Weil, and Wolfgang H. Wohlleber. 1979. Data Abstractions for Database Systems. ACM Transactions on Database Systems. 4:1.60-75.

(24) Macleod, I.A. 1979. SEQUEL as a Language for Document Retrieval. Journal of the American Society for Information Science. 30:5.243-249.

(25) Michaels, Ann S., Benjamin Mittman, and C. Robert Carlson. 1976. A Comparison of the Relational and CODASYL Approaches to Data-Base Management. ACM Computing Surveys. 8:1.125-151.

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

(27) Paolini, Paolo. 1980. Abstract Data Types and Data Bases. Brodie, Michael L. and Stephen N. Zilles, eds. Proceedings of Workshop on Data Abstraction, Databases and Conceptual Modelling. Association for Computing Machinery, Inc.

(28) Ritchie, D.M. and K. Thompson. 1974. The UNIX Time Sharing System. Communication of the ACM. 17:7.365-375.

(29) Rocchio, Joseph J. 1964. Possible Time-Sharing Organization for a SMART Retrieval System. Information Storage and Retrieval. (Scientific Report No. ISR-7.) Ithaca, New York: Cornell University Department of Computer Science.

(30) Rowe, Lawrence A. 1980. Data Abstraction from a Programming Language Viewpoint. Brodie, Michael L. and Stephen N. Zilles, eds. Proceedings of Workshop on Data Abstraction, Databases and Conceptual Modelling. Association for Computing Machinery, Inc.

(31) Rowe, Lawrence A. 1980. Issues in the Design of Database Programming Languages. Brodie, Michael L. and Stephen N. Zilles, eds. Proceedings of Workshop on Data Abstraction, Databases and Conceptual Modelling. Association for Computing Machinery, Inc.

(32) Salton, Gerard. 1968. Automatic Information Organization and Retrieval. New York: McGraw-Hill.

(33) Salton, Gerard, ed. 1971. The SMART Retrieval System: Experiments in Automatic Document Processing. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

(34) Salton, Gerard. 1972. A New Comparison Between Conventional Indexing (Medlars) and Text Processing (SMART). Journal of the American Society for Information Science. 23:2.75-84.

(35) Salton, Gerard. 1975. Dynamic Information and Library Processing. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

(36) Salton, Gerard. 1979. Suggestions for a Uniform Representation of Query and Record Content in Data Base and Document Retrieval. (Technical Report 79-363.) Ithaca, New York: Cornell University Department of Computer Science.

(37) Salton, Gerard. 1980. The SMART System 1961-1976: Experiments in Dynamic Document Processing. Encyclopedia of Library and Information Science. 28.1-36.

(38) Salton, G. and D. Bergmark. 1980. Parallel Computations in Information Retrieval. (Technical Report 80-439.) Ithaca, New York: Cornell University Department of Computer Science.

(39) Schmidt, Joachim W. 1977. Some High Level Language Constructs for Data of Type Relation. ACM Transactions on Database Systems. 2:3.247-261.

(40) Shopiro, Jonathan E. 1979. Theseus - A Programming Language for Relational Databases. ACM Transactions on Database Systems. 4:4.493-517.

(41) Smith, John Miles and Diane C.P. Smith. 1977. Database Abstractions: Aggregation. Communication of the ACM. 20:6.405-413.

(42) Todd, S.J.P. 1976. The Peterlee Relational Test Vehicle - a system overview. IBM Sys. J. 15:4.285-307.

(43) Wasseerman, Anthony I. 1980. The Extension of Data Abstraction to Database Management. Brodie, Michael L. and Stephen N. Zilles, eds. Proceedings of Workshop on Data Abstraction, Databases and Conceptual Modelling. Association for Computing Machinery, Inc.

(44) Williamson, D. and R. Williamson. 1970. A Prototype On-Line Document Retrieval System. Information Storage and Retrieval. (Scientific Report No. ISR-18.) Ithaca, New York: Cornell University Department of Computer Science.