A Query Language for Retrieving Information from Hierarchic Text Structures*

I. A. MACLEOD
Department of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6

Descriptive markup languages provide a mechanism for specifying the structure of a document. The basic premise of the work described here is that structure is an important characteristic of a document and is something more than a layout specification. For this reason, it appears important that retrieval tools should be developed which can take advantage of structural knowledge. In this paper, a query language is described which provides such a capability. The underlying implementation strategy is also discussed.

Received September 1989, revised July 1990

I. INTRODUCTION

There has been a rapid growth of interest recently in applications involving structured text. This stems from relatively diverse areas such as electronic publishing, office systems and linguistics.¹ Traditionally, the area in which structured text has had the most relevance has been in the display and publishing of documents. Two important developments here are the ODA² and SGML³ standards. The former is largely concerned with the development of standards to permit the interchange of structured office documents, while the latter provides a framework for developing descriptive markup standards. In essence, what these standards do, particularly in the case of SGML, is to allow us to talk about the logical components of the document, such as paragraphs, quotations, citations and so on, and to show the relationships among these components within the document. This has obvious application in document display. Knowledge that a particular section of text is a quotation, for example, would enable a formatter to display it in some distinctive manner such as italicised, indented or quoted.

The knowledge implicit in descriptive markup can be employed for applications other than display. It can be seen as a source of knowledge about a document.⁴⁻⁷ A major premise of the work described in this paper is that this knowledge can be important in document retrieval. Conventional text retrieval systems, as exemplified by the recently proposed command language standard,⁸ permit retrieval based on content. Simple contextual retrieval is possible where the context can be specified by defining a document to be a sequence of fields each having a unique name. Search-term matches can be restricted to occurrences within a particular field. The logical structures supported by descriptive markup are much more sophisticated than this. In SGML, for example, the document description is provided through a meta-language which is essentially a variant of BNF. Thus the complexity of a document structure has the potential to rival that of a program in a typical programming language. Implicit in this structure is knowledge about the documents. An example might be that a document contains a table with certain words in the associated table heading. Conventional retrieval systems cannot handle this additional information.

What is required, in addition to conventional retrieval, is the ability to perform complex contextual searches plus the ability to retrieve documents, or parts of documents, according to certain characteristics of their structure. It is the objective of the query language described in this paper to be able to provide these capabilities.

While there has been relatively little work in the area of structured document retrieval, there are a number of important on-going projects. The MULTOS project is intended as a multi-media application for document management in a distributed office system. It has been developed as an ODA application. It supports both context and structured searches. However, as described in Ref. 9, the contextual search capability is not fully general and only a rudimentary structure-based search is provided.

The OED work at the University of Waterloo¹⁰ has a more general retrieval capability. This is expressed using PAT¹¹ a command language front end. While this provides a more general retrieval capability than is found in conventional systems, flexibility is limited by the characteristics typical of command languages.

The Massive Memory Machine project at Princeton¹² has as one of its components a document database system which includes a document manipulation language (DML). DML is not intended as a user interface but rather is a relatively low-level language into which higher-level operations are intended to be mapped. The document model of DML is based on the representation of documents as sets of triples. Each triple represents a type, a key and a data item. Types, which may be user specified, define various storage structures. The key names an object within the document and is somewhat analogous to an SGML element. The data item associates a particular value with the key. An example triple is:

(string, 'title', 'This is the Title')

A complete document is a set of such triples. The advantage of this approach is that it needs only a very simple query language, which can be used to implement systems with more complex semantics.

The work described here is part of the Maestro project, which is concerned with the development of techniques for managing structured text. The three
systems briefly described above also deal with structured text, although, particularly in the case of MULTOS and DML, they have other goals. Of these systems, only MULTOS provides a query language interface, and this is a somewhat restrictive one. (DML is technically a query language but, as is apparent from its low-level nature, is intended to be used as an applications program interface within a host language.) In this paper a language is presented which is designed to provide a general high-level query language capability.

In addition to content, another hierarchical context for documents is that provided by their storage. An example is the ISO DFR proposal on document filing and retrieval, which suggests a hierarchic organisation for document storage modelling a typical view of office filing systems. Conventional retrieval systems for the most part regard documents as linear structures organised into independent collections. Experimental retrieval systems have recognised the importance of hierarchies in automatic classification (for example, see Refs 14 and 15), though in this context these are treated more as search aids than as storage management tools. An interesting aspect provided by the DFR proposal is that its view of the organisation of text meshes well with the SGML view of the structure of documents. Both are essentially hierarchies. This leads to the interesting possibility that the tools required to manage and process document collections may have similar characteristics to the tools required to perform document retrieval. This is one of the aspects being investigated in the work described in this paper.

2. OVERVIEW

The goal of the Maestro project is to provide an integrated environment in which the user can input, link and retrieve objects consisting of structured text. The query language described here provides the underlying retrieval capability. This is a syntactically simple but quite powerful tool. It is not however intended for the casual user. A mouse- and window-based front end provides the user interface, and a range of retrieval operations can be specified using menu selection. These specifications are translated into the full query language, which is also available for the knowledgeable user. The front end is being built using a programming environment tool. Our view of text as hierarchical meshes well with syntactic structures of programming languages, and many of the paradigms of such a tool are directly applicable to text.

The query language has two components, one to provide a definitional capability and the other a querying facility. The main objective of the definitional component is the provision of a document storage facility, which can bring together many of the structural concepts of the DFR proposal and the structures implied by the SGML standard. The major consequence of this objective is that the entire database is viewed as a hierarchy, while allowing the possibility of non-hierarchical linkages between nodes. The objective of the query language is to provide the capability to retrieve any node in the hierarchy based on its content and context.

Document input is performed using an SGML compiler. The output from this compiler is a pre-order traversal of the document tree. A compact representation of the tree structure is extracted from this output. In the prototype implementation, this is stored using the file structures of an existing conventional full-text retrieval system. More details of the implementation are provided elsewhere.

In the remainder of this paper, the definitional component of the language is described. The query language syntax and semantics are shown. The language syntax is derived from an earlier proposal for a conceptual model for document management. It is influenced by both SQL, the well-known relational query language, and Nial, a high-level language specifically designed for manipulating hierarchic structures. An overview is presented of the underlying techniques used to implement the language.

3. OBJECT TYPES

There is no consistent terminology for naming the various objects associated with text and textual databases. The terms used here are mostly derived from ISO documents, though these themselves use conflicting terminology. A database is a set of archives. (The DFR proposal calls these ‘document stores’, which is what an archive is.) An archive contains a group which contains either documents, references or other groups of any combination of these. A reference is a pointer to an object, presumably one not already contained within the group. This definition of a group is recursive, so that arbitrarily deep hierarchies can be specified. These objects may have attributes associated with them. These are used to contain information which describes the object but is not, strictly speaking, part of the object. An example might be an attribute containing the date on which the object was created. A document also contains text. The structure of this text is described by a hierarchy of elements. An element is a component of the document. It may consist simply of text, as in the case of a title, or it may contain other elements, as in the case of an abstract containing paragraphs. Further, elements may be lists, as in a list of chapters. Figure 1 illustrates the overall structure of an archive. In the remainder of this section the language facilities of Maestro used to describe these objects are defined and an example set is given. In the metanotation used to describe the syntax, square brackets enclose optional items and braces enclose items which may appear zero or more times.

An archive is defined as follows.

archive: ArchiveName [AttributeDefinitionList]

This is basically the way in which the root node of a hierarchy is defined. Archives can have attributes associated with them. An attribute definition list is of the form:

    with AttributeDefinition [.AttributeDefinition]

When other groups of the archive are defined, they must be associated with an already defined group.

    group: GroupName [AttributeDefinitionList]

References are handled by defining reference groups. This differs from the DFR proposal where individual references are considered to be independent objects. The use of groups allows this type of object to be readily identified and examined, and this seems to be a desirable
Figure 1. Overall database structure.

capability. One important use, for example, might be to allow groups of exceptions to be defined. In many applications – the *Oxford English Dictionary* is an example – the 'documents' will mostly, but not always, conform to a particular type definition so that techniques must be available for exception handling. Note that the characteristics of the text of documents are independent of their position in a group hierarchy. In contrast to the usual object-oriented philosophy, these characteristics are not inherited, nor would it make sense for them to be.

**reference**: `ReferenceGroupName [AttributeDefinitionList]`

Where a reference group name has attributes, these are appended to any document referenced from the group. This allows additional information, such as a note describing the reason for the reference, to be included as if it were one of the original attributes of the referenced document. Attributes are defined as:

```
AttributeName AttributeType
```

In the present prototype, only two types are provided, `text` and `reference`. This is a considerably less general capability than is presented in the DFR proposal. It is deliberately kept simple here as it is felt that the attributes have the potential to interact with some sort of DBMS facility, and that this interface area between text and DBMS is one that would benefit from considerably more investigation. These attributes are not inherited in the current model, and this aspect is also something that needs further consideration. At the present time no facilities are provided for handling the SGML type of attribute, which allows information to be attached to an element within the text of a document.

Documents are defined independently of groups and a separate mechanism is used to indicate what document types are associated with a group.

**document**: `DocumentName is TypeName`  
`[AttributeDefinitionList]`

A type is defined as follows:

**doctype**: `TypeName [' RuleList']`

The `TypeName` only appears in this pair of statements. It has no meaning outside this context and is simply a convenient mechanism for allowing the same type definition to be shared among classes. The notation used in the rule list is modelled on a basic subset of SGML. A separate mechanism is used to provide some control over the types of document that may be stored within a particular group.

**member**: `DocumentName of GroupName`  
`{, GroupName}`

The following sequence of definitions is used to set up a simple archive where the archive contains three groups.

- **archive**: `Papers with CreationDate text`  
- **group**: `DatabasePapers of Papers with`  
  `DateOfLastPaper text`  
- **group**: `TextPapers of Papers with DateOfLastPaper text`  
- **group**: `MyPapers of Papers with DateOfLastPaper text`

This defines a simple two-level hierarchy. Now suppose it is necessary to allow the `DatabasePapers` and `TextPapers` groups to contain references as well as documents. This could be accomplished using the following definitions.

- **reference**: `XRef of DatabasePapers with ToGroup text`  
- **reference**: `XRef of TextPapers with ToGroup text`

Here the `ToGroup` attribute is provided, so that the name of the group containing the referenced document can be explicitly recorded if desired. Further suppose that only three types of document are to be stored and that these are defined as follows:

- **document**: `MyPaperType is MyType with Date text`  
- **document**: `CompJType is PaperType with Date text`  
- **document**: `CacmType is PaperType with Date text`

The association between document types and groups is specified by the following statements.

- **member**: `MyPaperType of MyPapers`  
- **member**: `CompJType of MyPapers, DatabasePapers, TextPapers`  
- **member**: `CacmType of MyPapers, DatabasePapers, TextPapers`

The structure of the documents is defined using an SGML-like notation. For example:

**doctype**: `PaperType`  
`[< * (Front, Body, Back) >  
< Front (Title, Author+, Location, Abstract) >  
< Abstract (Paragraph +) >  
< Paragraph (Sentence +) >  
< Body (Section +) >  
< Section (SectionHeading, (Paragraph + | (Paragraph*, SubSection +))) >]`
A QUERY LANGUAGE FOR RETRIEVING INFORMATION FROM HIERARCHIC TEXT STRUCTURES

The metalanguage used to describe the structure of the document is based on the usual metasymbols associated with programming language definitions.

doctype: MyType

{<Title, Abstract, Section*, Citation+>}
{<Paragraph+>}
{<Sentence+>}
{<Section (SectionHeading, Paragraph+)>}
{<Subsection (SectionHeading, Paragraph+)>}

The name of the first element of a definition is the DocumentType defined in the document definition. Since several different definitions may use the same structure definition, the symbol "*" appears as the first element of a structure definition.

To the underlying storage system, unspecified elements — in this case Title, Author, Location, SectionHeading, Citation and Sentence — are treated as consisting of a list of words. Figure 2 illustrates part of a possible hierarchic structure created as a result of the above examples.

![Diagram of database structure]

Figure 2. Example database structure.

4. SEARCH CAPABILITIES

The objective of the query language is the provision of sufficient capability to allow any node in a document hierarchy to be retrieved. One or more conditions may be specified. There are a number of contexts in which conditions may be applied. They can be applied to the node being retrieved, to an ancestor of that node, to a descendant of the node, or to a descendant of an ancestor node.

The result of a query is a set of references. These are pointers into the hierarchy. These can be named and subsequently queried. Access to the actual text and attribute values associated with a reference is achieved through the navigational features of the interface and is not described here.

The syntax of a selection is as follows:

ReferenceResultName gets ReferenceExpression
{union | intersection | difference
ReferenceExpression}

The ReferenceExpression specifies what is to be retrieved and any conditions it must satisfy. Since the storage organisation is also hierarchical, any object can be retrieved — archives, groups, documents or particular elements. This means that the same language that is used to retrieve documents from the database can also be used to navigate through it. Attributes of type reference can also be retrieved. (Non-reference attributes are accessible via the interface.) The only exception is that reference groups and reference sets are not considered as first-class objects. Rather they are collections of objects, and selecting such a group as a retrieval target causes their contents to be retrieved.

A ReferenceExpression includes the name of the object being retrieved and, optionally, a context specified by the names of some or all of the objects along the path from the retrieved object to the root of the hierarchy. Retrieval will normally take place within a single archive. A reference expression takes the following form:

ObjectSelection {of ObjectSelection} [from FinalSelection]

An ObjectSelection is an ObjectSpecification optionally followed by an ObjectConditionList. These are discussed in later sections. The simplest form of an object specification is just the name of the object. The FinalSelection specifies the source from which formation is to be retrieved. It can be the name of a group, a reference set name or a reference set name qualified by an element name. If it is omitted, the default is the last recognised group. It may be qualified by conditions, in which case a conditional retrieval operation takes place, as described below.

4.1 Simple selection and navigation

The simplest type of selection is unconditional. The classes of object present in the hierarchy are archives, groups, documents and document elements. Any of these can be selected, as is illustrated in the following set of examples:

Papercopy gets DatabasePapers from Papers

The result is a single reference to the group DatabasePapers.

Contents gets XRef

Since XRef is a reference group, the result is the set of references contained in XRef. Note that there are two XRef's. The default rule causes the one contained in DatabasePapers to be used.

PaperGroups gets group from Papers
The result is a set of references to all the first-level groups in the Papers archive.

LowerGroups gets group from PaperGroups

The result is the set of references to the next level of groups.

XDocs gets document from LowerGroups

The result is a set of references to all the documents contained in this last set of groups.

AbstractList gets Abstract from XDocs

Here references to the abstracts of the last set of references are retrieved. Note that the full path of an element need not be specified. This allows elements from different document types to be selected within a single operation.

TitleList gets Title from AbstractList.document

In this case all the associated titles of the retrieved abstracts are retrieved. In this case it is necessary to specify an ancestor source, since Title is not a descendant of Abstract. In general, this form permits retrieval of elements from a different branch of the tree than the ones indicated by the original reference list.

The usual set operations, union, intersection and difference are allowed in reference expressions. The references in both sets must be to the same type of object, otherwise the result is the empty set. For example:

CrossReference gets XRefOf TextPapers union XRef of DatabasePapers

These examples mainly show how the query language can be used to ‘navigate’ through the database. More interesting retrieval operations can be specified using conditional retrieval, as described in the next section.

4.2 Simple object selection conditions

Conditional retrieval is specified in much the same way as in other well-known query languages. A major difference is that the user is not required to be totally familiar with the structure of the material being retrieved. As noted earlier, undefined elements in the document-type definition are considered to be composed of lists of words. Other elements defined in terms of these elements may also be viewed in this light. Thus conditions can be applied at any level of the hierarchy.

Any element of a ReferenceExpression may be followed by a set of conditions. These can appear in two forms, the simpler of which is structured as:

where (ObjectSelectionConditions)

A more complex form, required to handle lists, is discussed in the following section.

The ObjectSelectionConditions may involve a single operator or may be a parenthesised expression. The conditions may be applied to the current node or to a descendant node. At the present time only a limited number of operators are being implemented. These have either one or two operands; one of which is normally an ObjectSpecification. The most important operator is the in operator, which tests if its first operand is a member of the list represented by the second operand.

As previously noted, a condition may be applied in a number of different contexts. The following four examples show each of the different contexts.

(i) Condition at the node being retrieved:

PaperList gets document where (‘database’ in document)

This is an example of a simple condition. The in operator tests if ‘database’ is one of the list of words comprising the document. As is further discussed later, the present version of the system is implemented on top of a conventional text retrieval system and the usual stem and wildcard capabilities are supported, though not a general substring search capability.

(ii) Condition at a descendant:

SmithList gets document where (‘Smith’ in Author)

This is an example of a simple contextual condition. In this case the underlying semantics are somewhat more complex in that the operation must first check that ‘Author’ is in fact an element of the documents in the current group. In the example database, it is in fact not an element of MyPapers.

(iii) Condition at an ancestor:

SectionList gets Section of document where
(‘database’ in document)

Here an element other than a document is being retrieved. The condition applies to the document node rather than the element, so that what would be retrieved is every section of every document satisfying the condition.

(iv) Condition at a descendant of an ancestor:

SectionList gets Section of document where
(‘database’ in SectionHeading)

Here the condition is applied to an element of the document which is not necessarily a descendant of the element being retrieved. As in the previous example, all the sections of any document satisfying the condition would be returned.

Queries involving any number and any combination of these types of condition are possible. Also quite complex membership tests can be performed, and these correspond to the search capabilities fund in conventional text retrieval systems (and are in fact implemented directly on them in the current prototype). For example,

PaperList gets document where
(‘database | data . base & text’ in document)

This example retrieves documents containing either the word ‘database’ or the phrase ‘data base’ and the word ‘text’.

An example of a request involving several conditions is the following:

List gets document where (‘Smith’ in Author or ‘database’ in Title) and ‘text’ in Abstract)

Another example involving a combination of node conditions is the following:

SubSectionList gets SubSection where (‘database’) in SectionHeading) of Section where (‘text’ in SectionHeading)

Here, the first condition is applied to those subsections contained within sections satisfying the second condition.
Retrieval can also be carried out within reference set results. These are simply a set of pointers within the archive hierarchy. A subset of them can be retrieved:

SubList gets SubSectionList where ‘text’ in SectionHeading

Descendants can be retrieved:

SubList gets SectionHeading of SubSectionList

Ancestors can also be retrieved:

SubList gets SectionHeading of SubSectionList . Section

In other words, given a set of references, the query language facilities allow these to be used as a context for the retrieval of any object related to these references.

4.3 Contextual conditions

The general capability to perform element searching in hierarchies introduces complexities caused by the fact that elements may be lists and that they may also appear in more than one path (or context) in the hierarchy. For example, in the definition of the contents of ‘PaperType’, the element ‘SectionHeading’ appears in two different contexts, one as the heading of a section and the other as a heading of a subsection. Language facilities are required to enable the user to distinguish between these two contexts.

This capability is provided by a ContextualCondition. This establishes that a named element or document type exists in the collection. It also establishes a context for succeeding conditions. The general form of a contextual condition is:

[not] having ObjectSpecification [where
(ObjectSelectionConditions)]
[ContextualCondition (Connective
ContextualCondition)]

The following examples illustrate the major permutations.

(i) A single contextual condition

A gets document not having Citation

This example retrieves all instances of papers containing no citations. The following example retrieves all papers of a particular class type.

A gets document having CompJType

(ii) A contextual condition qualified by object selection conditions

A gets document having SubSection where ‘database’ in SectionHeading

Here the having restricts the subsequent condition to ‘SectionHeading’ of ‘SubSection’. The condition would fail for papers where the only headings containing ‘database’ were those of sections.

(iii) Multiple contextual conditions

A gets document having Section having SubSection

In general, this type of construct can be used to test for the presence of a particular path.

(iv) Multiple contextual conditions qualified by object selection conditions

A gets document having Section where ‘database’ in SectionHeading having SubSection where
‘database’ in Paragraph

This construct allows the context of the first condition to be further constrained before the second contextual condition is applied.

Contextual conditions can be freely combined with other syntactic constructs as is illustrated by the following examples.

A gets document having SubSection where
‘database’ in SectionHeading or ‘text’ in SubSection

In this example two conditions are specified. Both are applied within the context of the same SubSection. For the conditions to be applied independently, two separate contextual conditions would be required as in:

A gets document having SubSection where
‘database’ in SectionHeading and having SubSection where
‘text’ in SubSection

A ContextualCondition can be applied to any element in an ObjectSelection, as is illustrated by the following example to retrieve, from all the sections with ‘retrieval’ in the header, all the subsections that contain a paragraph containing the word ‘database’.

SLList gets SubSection having Paragraph where
‘database’ in Paragraph of Section where
‘retrieval’ in SectionHeading

These examples show that a very powerful yet compact query language can be designed to provide general retrieval facilities within a text hierarchy.

4.4 Object specification

An element may, and in many cases will, be a list. In addition to the context question introduced above, this introduces issues dealing with position and quantification. These appear in conjunction with ObjectSpecification. The general form of this construct is:

Object | Locator Object |
Object @(Position)|
Quantifier Object

As can be seen from the previous discussion, an ObjectSpecification can appear in any of three contexts. One is in a ReferenceSelection, another is in a ContextualCondition and the last is in an OperatorExpression. Examples of the various ObjectSpecification constructs in each of these contexts are shown below.

There are four types of Locator. These are first, second, third and last. These are used to specify particular instances of elements. Some examples are:

A gets first Section of Papers where ‘database’ in SectionHeading

This query retrieves the first section of any papers having ‘database’ in any section heading (not necessarily the first).
A **gets** Section of Papers where (‘database’ in first SectionHeading)

This example retrieves every section of each paper where ‘database’ appears in the first SectionHeading of the paper.

A **gets** Section of Papers having first SubSection where (‘database’ in SectionHeading)

Finally, this example retrieves every section of each paper where ‘database’ appears in the SectionHeading of the first SubSection of the paper.

A more general way of specifying particular elements is possible using the ‘@’ notation. Here the Position is a list of one or more integer values. An absurdly complex query is the following:

A **gets** Section@(1,2,3) having SubSection @4 (where ‘database’ in Sentence @5) of Papers @1..100

This retrieves any of the first three sections of the first 100 instances of Paper, where the section contains a fourth subsection having a fifth sentence containing the term ‘database’. This example most usefully illustrates that the power of this construction is sufficient to resolve any reasonable query of this nature.

Quantifiers are used to determine the number of members of a list of elements that should participate in a search. There are three quantifiers, any, notany and all.

The first two of these may be optionally qualified with an integer to specify an exact quantity. Examples are the following:

A **gets** document where (any 2 ‘information, retrieval, database’ in Abstract)

Here the documents are retrieved if any two of the specified list of terms appear in the abstract. This form of quantification is basically an alternative way of expressing a more complex boolean operation.

A **gets** document having all Section having any 2 Subsection where (‘text’ in SectionHeading)

This example represents a somewhat unlikely query but illustrates the use of quantifiers in the context of the having clause. Here documents will be retrieved only if every section contains at least two subsections with the word ‘text’ in each of their headings.

A **gets** any 3 document where (‘text’ in SectionHeading)

This last example retrieves the first three papers found which satisfy the query.

The syntax is sufficiently powerful to express complex queries based purely on document structure.

A **gets** document having all 3 Section not having any 5 SubSection and having all SubSection having any 5 Sentence

This query retrieves all papers having three sections where each section at most has four sections each containing at least five sentences.

The above examples illustrate the main syntactic features of the language. A more complete syntax definition is provided in the Appendix. Additional functionality, particularly a display capability, is provided within the underlying implementation. These features are accessed from within the user interface and are discussed elsewhere.

5. DOCUMENT CREATION AND DELETION

Documents cannot be input directly using the query language. This is because all text for a particular document type must satisfy the constraints imposed by the related type definition. This conformity is achieved by processing all document text through an SGML compiler. Text input is specified by a statement of the form:

*DocumentName* \{of *GroupName*\} **gets** ‘*FileName*’

This operation gets the text from the specified file and processes it through an SGML compiler. The processed output is stored in a temporary location.

For a reference group, the equivalent statement is:

*ReferenceGroupName* \{of *GroupName*\} **gets**

*ReferenceSetId*

Attributes of documents can be input directly, and this capability provides a very simple yet powerful mechanism for creating complex document structures because of the capability of storing references as attributes of documents.

*AttributeName* \{of *StructureNames*\} **gets**

*AttributeValue*

This construct is used to supply attribute data to a new document. An attribute value can be a literal value or the result of another retrieval operation. The data is stored in a temporary location.

The operation of adding the text and attributes to the database is carried out by the operator **new**, whose single operand is the name of the document type. The result of the operand is a reference to the newly created document.

For example, suppose links to and from citations and their cited documents were required. The following sequence of operations shows an example of finding a cited document and linking it to the document citing it.

First, a reference group is set up:

**reference**: CitationLink of Papers with **BackLink**

Citing **gets** Citation@1 from document where (Title = ‘Text and Database’)

This operation finds the first citation in a document with a particular title. Next, the cited document is itself obtained.

CitedDocument **gets** document where (Title = Title of Citing)

The reference to the cited document is next stored.

CitationLink **gets** CitedDocument

The attribute value containing the back link is then set.
A QUERY LANGUAGE FOR RETRIEVING INFORMATION FROM HIERARCHIC TEXT STRUCTURES

BackLink from CitationLink gets Citing
Finally, a new instance of CitationLink is created.

new CitationLink

Obviously, this particular example is somewhat artificial. Finding and storing references in this way would be unpleasantly tedious. However, what it does show is that the capability to perform these operations can be expressed in a straightforward manner. A more natural interface based on mouse and windowing operations is being layered on top of these types of operation.

Once such a group has been created, this structure enables all available Papers which cite particular documents to be retrieved. For example:

Cittings gets BackLink from CitationLink
where (CitationLink in document of MyPapers)

Here all the back links are found for any of the documents cited which are in group MyPapers. This example has been a very simple one, but it illustrates all that is needed to build complex structures and to perform retrieval within them. Obviously for retrieval from potentially very complex documents such as hyperdocuments where there will be both cross links and hierarchical links among the various components, there will be a need for tools to assist in navigation through the object. Again, such tools are seen as being higher-level aids provided within the interface rather than as some additional query language feature.

The operator remove is used to delete documents. The result of the remove operation is simply the references to any documents deleted. Physical deletion does not take place until exit is made from the system.

6. IMPLEMENTATION CONSIDERATIONS

The semantics of the retrieval operations are expressed in terms of an underlying algebra. The operators are described briefly here. The result of every operation is a list of references. The implementation view of a reference is as a path through the tree to a particular element. Thus the conceptual structure is:

GroupName Documentld Element [Index] [Element [Index]] ...

Here the Index is the position of the element if it is part of a list of elements. For example, the heading of the third subsection of the second section of database paper 1000 would be logically represented as:

DatabasePaper | 1000 | Section | 2 | Subsection | 3 | Heading |

The actual physical representation is somewhat different but effectively contains this information.

6.1 Selection operations

These operations select references on the basis of the structure and/or content of documents.

attributeSelect (TypeList, AttributeName, AttributeValue, Relation)
pathSelect (TypeList, Path)
contextSelect (TypeList, Term, Path)

These three operations select references on the basis of an attribute value, the existence of a particular path and the existence of a term in a particular path, respectively.

6.2 Restriction operations

These operations extract subsets from reference lists based on certain characteristics of the references

ancestor (ReferenceList, Element)

This operation finds the references to the element which is an ancestor of the reference list passed as a parameter.

pathChoose (ReferenceList, IndexList)

This operation extracts particular references containing the elements whose position is specified by the index list.

6.3 Set equivalent operations

These operations are equivalent to the usual set operations of conventional systems.

pathIntersect (ReferenceList, ReferenceList, AncestorPath)
pathDifference (ReferenceList, ReferenceList, AncestorPath)
pathUnion (ReferenceList, ReferenceList, AncestorPath)

The above three operations perform intersection, difference and union operations based on the existence of a common ancestor.

Query language statements are translated into a sequence of these operations. Taking an earlier example:

SList gets SubSection having first Paragraph where ('database' in Paragraph) of Section where ('retrieval' in SectionHeading')

This would be processed to yield the following sequence of operations:

A = contextSelect (Papers, 'retrieval', 'Section/SectionHeading')
B = contextSelect (Papers, 'database', 'SubSection/Paragraph')
C = pathChoose (C, '1')
SList = pathIntersect (B, A, 'Section')

In the prototype implementation, the operations are implemented on top of an existing full text retrieval system, Ful/Text, as is more fully described elsewhere. This approach allows the construction of an effective test bed at relatively little cost in the sense that all the features of conventional retrieval systems are made available. Thus a full inverted index is available and wildcards and stemming are handled. A further advantage unique to Ful/Text is that it supports zones. Zone numbers can be assigned to portions of a document and searches can be restricted to particular zones. Overlapping and non-contiguous zones are permitted, so that it is possible to map SGML elements into zones. The major problem is performance for certain types of operation. The underlying conventional operators do not retrieve paths. They treat the text as linear and retrieve character offsets. In most cases these must be converted into paths, and this can involve some significant overhead. In the above sequence, the second context-select operation will almost certainly yield a large number of references which will

THE COMPUTER JOURNAL, VOL. 34, NO. 3, 1991 261
not end up in the final result, and the cost of generating the full paths for these may be non-trivial.

7. SUMMARY

The query language is implemented as a prototype. The implication of this is not that the system works only with small databases but rather that some of the generality implied in the preceding description is lost because of restrictions in the underlying file structures. In particular, all documents in the same group must have the same attributes (but not the same document type) and the number of sub-groups in a group is restricted to four. No attempt is currently made to enforce referential integrity.

From a practical point of view, there are a number of important retrieval tools which cannot be expressed in the basic language. Examples are weighted searches and sorted results. These are conceptually simple operations which can, however, have lots of variations. In weighted retrieval, there are various weighting algorithms based on word distributions. Further, locations of terms may affect their importance, results can be sorted, there may be cutoff thresholds, and so on. Further, a different weighting algorithm might be appropriate for retrieving sections, say, as opposed to full documents. It is extremely difficult to express all these various possibilities using a predefined syntax, and this is why they are omitted from the basic language.

Stonebraker has shown how extending database language through procedures adds little to the complexity of a relational query language, but provides an extremely powerful framework for specialised operations. A similar approach is being taken in Maestro. What is being provided is a limited language extensibility. This will permit applications to be written as external procedures. These can be invoked from the query language using a keyword notation for parameter passing. This enables specialised statements to be constructed in a fairly straightforward manner. For example:

```
A gets WeightedList of Mypapers about 'text and database'
```

Procedures are invoked by 'retrieving' them. Here

‘WeightedList’ is a procedure. It is invoked in the context of MyPapers, which becomes a parameter to the external procedure. The user-defined keyword ‘about’ precedes the single explicit parameter.

This approach allows extensions to be incorporated into the language using a relatively readable syntax. At the same time, the compactness of the original language is preserved.

As previously noted, the query language is not regarded as suitable for the casual user. Present efforts are concentrated on adapting a programming environment tool to provide a user-friendly front end and also an environment for document display and browsing. The design has been completed of a sophisticated mouse- and window-based hypertext interface. One of the interesting features of this design is that all operations such as link traversal, document elision and path creation are expressed in terms of query language operations. This shows that the language is sufficiently powerful to express the semantics of hypertext operations and allows the interface design to be carried out independently of the physical database structures.

The present implementation is not wholly satisfactory. The use of an existing conventional text retrieval system provided a convenient basis on which to implement and experiment with the language but, depending on the data actually retrieved, certain operations, such as the set equivalent operations, are potentially slow.

Another problem is the lack of an integrated high-level programming language for handling non-procedural sequences with conditions and loops. The earlier example of creating citation links is somewhat artificial without such a capability.

A further area of concern is the future of SGML. While the work described here is not based specifically upon SGML, it does presuppose the existence of documents marked up in such a way as to reflect their hierarchical structure. Unless SGML or some similar system becomes a general tool, the cost of assembling and processing large databases of structured documents may be prohibitively expensive. However, as Barron has indicated, SGML does seem to have some sort of future, though whether as a niche product or as a universal one remains to be seen.

REFERENCES


APPENDIX – THE LANGUAGE SYNTAX DEFINITIONS

archive: ArchiveName [AttributeDefinitionList]

GroupDefinition
  group: GroupName of ArchiveName
       [AttributeDefinitionList]

ReferenceGroupDefinition
  reference: ReferenceGroupName of GroupName
             [AttributeDefinitionList]

DocumentDefinition
  document: DocumentName is TypeName
            [AttributeDefinitionList]

ClassAssociation
  member: DocumentName of GroupName (,GroupName)
         AttributeDefinitionList
         with AttributeDefinition (,AttributeDefinition)

AttributeDefinition
  AttributeName AttributeType

ContentDefinition
  docType: TypeName ? "('<AssociateSymbol ComponentList>
             Rule '+')' "'

ComponentName (MetaStructure)
  MetaItem (,MetaStructure)

MetaSymbol
  ElementName [MetaSymbol] | ComponentList
  + | * | ? MetaStructure | ']' ComponentList
  Rule

OSTATMENTS

Retrieval/Operation
  ReferenceResultName gets ReferenceExpression

CreateOperation
  new DocumentName (of GroupName)

DeleteOperation
  remove ReferenceExpression

TextInput
  DocumentName (of GroupName) gets File

ReferenceAssignment
  ReferenceGroupName (of GroupName) gets
  ReferenceSetName

ReferenceSetName
  ReferenceResultName | ReferenceGroupName

AttributeInput
  AttributeName (of StructureNames) gets AttributeValue

StructureNames
  ArchiveName | GroupName | DocumentName
             | ReferenceGroupName

AttributeValue
  String | ReferenceSetName

SELECTION

ReferenceExpression
  ObjectSelection (of ObjectSelection) [from FinalSelection]
    {SetOperator ReferenceExpression}

ObjectSelection

ObjectSpecification [OptionalConditionList]

FinalSelection

FinalSpecification [OptionalConditionList]

SetOperator
  union | intersection | difference

OptionalConditionList
  where (ObjectSelectionConditions) | ContextualCondition

ObjectSelectionConditions

ObjectConditionExpression | Connective

Connective
  and | or

ObjectConditionExpression

ObjectCondition | (ObjectSelectionConditions)

ObjectCondition

Operation | ContextualCondition

ContextualCondition
  [not] having ObjectSpecification [where
   (ObjectSelectionConditions)]

ObjectSpecification

Object | Object @(Position) | Locator Object | Quantifier Object

Object

Collection | ElementName

Position

Integer PositionOptions

PositionOptions
  {Integer} | ... Integer

Locator

first | second | third | last

Quantifier

AnyQuant | NotAnyQuant | AllQuant

AnyQuant

any [Integer]

THE COMPUTER JOURNAL, VOL. 34, NO. 3, 1991  263
I. A. MACLEOD

Correspondence

Sir,
Professor Elliott has, I understand, written to you concerning the factual accuracy and scholarship of *Innocating for Failure* in connection with the history of the computer Pegasus; and I am content to leave to him the enlightenment of the author, Dr John Hendry.

As the only surviving senior member of the Pegasus design team, may I however supplement Professor Elliott’s observations with a note on the commanding position in computer architecture and construction won by Pegasus through the engineering leadership of Charles Owen and the logical insight of Christopher Strachey, a position abandoned and then lost through lack of judgement and badly handled investment issues.

There was no Pegasus prototype; the first machine was a handbuilt but otherwise fully engineered construct, and from the outset the objective was for production to follow validation of the pilot model.

Economy was achieved (on the lines of the Elliott 401) by designing a small range of standard plug-in packages covering every aspect of logic, immediate-access memory and drum-store electronics; reliability was ensured by accepting highly conservative limits on gate design, valve parameters, magnetic recording density and pulse-strobing tolerances. All mechanical issues (including those of drum bearing wear which had worried NRDC at both Elliotts and Ferranti) were resolved, unusually for the electronics industry of the 1950s, under the direction of a gifted mechanical engineer, the late Brian Maudsley.

The outstanding architectural achievement was the realisation of balance in design, eliminating ‘wait states’ through the working harmony of logical control, ‘mill’, 7 accumulators, immediate-access and drum-store systems, co-ordinated and addressed through the powerful and consistent Pegasus order-code – which continued to influence computer design for many years.

It is noteworthy – contrary to Hendry’s account – that the drum-store played its part in achieving design balance, reducing the need for controlling the physical layout of drum records with an interleaved address block and fast electronic cross-bar switching for both writing and reading – the latter feature a completely new departure using a germanium diode switch operating on the unamplified signal from the drum read-head. Reliability also benefited from the elimination of electromagnetic relays.

It is a matter of record that all of these features were working in the Pegasus pilot by April 1956. Developments begun in 1955 added card and tape peripherals leading to eventual sales of nearly 40 machines overall.

Regardless of these good beginnings, an evident loss of focus on the part of the Ferranti senior management coupled with NRDC’s short term financial preoccupations fostered an atmosphere in which by 1956 the burden of continuing investment was only acceptable at a level requiring a fundamental choice between the Mercury team in Manchester and the Pegasus team in London.

Few remember Mercury now, but except for the peripheral developments mentioned above the Pegasus team was largely disbanded, and staff were redirected to work on Ferranti contract and defence work or, in the case of some of the leading team members, regrouped under American auspices from September 1956. This was a blow to the infant British computing industry at a most crucial time, from which subsequent events have shown it never wholly recovered, exemplifying how inadequate investment ensures a nett and enduring loss. In comparison, other management failings disclosed by these events are insignificant.

Yours faithfully

IAN W. MERRY

264 THE COMPUTER JOURNAL, VOL. 34, NO. 3, 1991