Handling Multiple Data Bases in Document Retrieval

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

ABSTRACT
There is no such thing as a standard document. Bibliographic information comes in a wide variety of formats. Existing retrieval systems handle different document styles either by creating an artificial document type or by providing different and independent data bases. Neither approach seems satisfactory. In this paper we describe a data model which we feel is more appropriate for document representation and show it can handle the multiple document type problem quite naturally.

1. INTRODUCTION

Existing retrieval systems normally can handle only one type of document at a time. They handle different document types either by defining a "standard" document format and constraining individual documents to fit inside this format or by providing independent data bases. Related information held in separate data bases cannot be combined within the context of the retrieval system environment, not even when all that is wanted is common information such as a list of titles.

It would obviously be advantageous to be able to retrieve common information from documents which are otherwise dissimilar. For example, in libraries there are many different types of document: books, reports, maps, journals and so on. We normally go to a library to collect information, not a particular type of document. Another example is in "people" files. File folders in filing cabinets do not, in general, contain documents of the same type. Indeed, the content of a file may itself be a file. What they contain is a number of physically quite different objects. What relates them is their *content* rather than their *structure*.

In some ways current approaches to data organisation have evolved from a rather idealistic view of data. Traditional data processing techniques grew around the view that data could be organised into clean well structured files. Data was constrained to this form. Data models evolved to aid in the management of related files. What they reflect is a bias towards modelling of data suited for a computer rather than the real world data that exists in people's libraries and offices. The information here suffers from never having been computerised, or, at best, computerised in a variety of ad hoc ways, as for example, can be seen in the case of current document retrieval systems. The work described here is premised upon the belief that a data modelling approach to document retrieval is a good one. We give an abbreviated description of a data model which we feel is appropriate for document representation and show it can handle the multiple document type problem quite naturally.

2. DATA MODELS

There are generally assumed to be three "classic" data models, the hierarchic, network and relational models. However, this is slightly misleading. A model can be more appropriately defined as consisting of a logical data structure, or structures, and a set of operators to access and manipulate the data structure. In this light, it is perhaps more correct to refer to the three classic models as generic types of model. Specific instances of models are usually based on one of these types but there are often significant differences among instances of the same type. For example, the relational calculus and relational algebra models are quite different although they are based on the same underlying concept [3]. SQL, another relational system, differs from both of these and, indeed, there are several SQL variants [2]. There are even some models which are not based on one specific generic type. Daplex is a particular example [10].

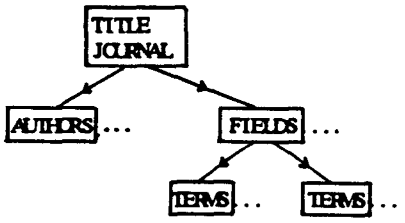
In the context of document retrieval it is interesting to briefly look at the more obvious relevant properties of the three generic models. The hierarchic model is based on the assumption that all data can be represented within a single hierarchy. The "pure" hierarchic model allows only one-to-many relationships, although variations permit more than one hierarchy and allow interconnections between different hierarchies. This model is obviously not totally inappropriate for document organisation. Libraries, for example, almost invariably classify their contents in a hierarchic fashion. The network model permits, in principle, many-to-many relationships which, as well as permitting hierarchies also allows data to be shared among hierarchies. (Since some of the variant hierarchic models permit a similar sharing, IMS is a classic example [5], it is apparent that the classification of a particular data model into one of the generic types is not always straightforward.) The network concept is also relevant to document collections. One useful application would be in permitting different logical organisations to co-exist, as they do in most real libraries, where the same document can appear in a variety of different types of catalogue.

In the relational model, the basic data type is a relation or table. In the "pure" model, a relation is basically a set of tuples where each tuple is equivalent to a data record. All tuples within a relation are of the same type. In most actual instances of relational implementations, sets are replaced by tables where duplicate entries and orderings are permitted. A fundamental difference between the relational view and the other two model types is that all relationships are established dynamically. That is, when related information appears in two or more different tables, it is brought together by linking the information through the values of the data attributes in the table. In the tree and hierarchic views, the relationships are static and are implemented by explicit pointers. In the relational model we tend to ***build*** new information from existing information while in the others we tend to ***navigate*** through the database using the preassigned pointers. Thus the concept of a ***schema*** is fundamental to the tree and hierarchic approaches. A schema is a mechanism for describing the structure of the database. As a consequence the basic query languages associated with relational models tend to be extremely powerful high level languages. The other two models have simple query languages which are almost procedural in nature and which are mainly concerned with moving through the data base. It is this dynamic aspect of the relational model which makes it attractive in a document management environment. At the same time it should be noted that the representation of documents and document organisations as tables is not necessarily an ideal approach.

3. THE ARRAY MODEL

Most work involving the application of data models to document retrieval has centred on the relational model. This is obviously because of the inherent attractiveness of the associated query languages and the conceptual simplicity of the model. Unlike hierarchic and network environments, users do not have to be aware of any underlying schematic description of the database before they use it. At the same time there has been an increasing awareness that there are a number of problems associated with the model and this has prompted a great deal of research into variations of the relational concept and not just in the area of document retrieval [4], [9], [11], [6]. The work described here is an example.

The array model was first suggested some years ago as a generalisation of APL [8]. We have adapted some of More's ideas application in a data base management environment. Details of the model are provided elsewhere [6,7], so we will describe only the major features here, mainly through examples. In this model, the basic information structure is a non-homogeneous array where each element of the array may be a basic data type such as an integer or a string or it may itself be an array. Thus an array is a hierarchical object. (We use the terminology "array model" to distinguish the model from the classic hierarchic model and also to reflect the fact that we borrow many concepts from More's array theory.) For example a paper consisting of a title, a list of authors, a list of fields, each of which had associated with it a list of index terms, would be represented as:



In our model, a data structure of the type illustrated above is declared by a statement of the form:

Papers: ARRAY
(Title, Journal, (Authors), (Fields, (Terms)))

Here the hierarchic levels are specified by nesting of parentheses. The basic assumption in our model is that information is made of composite objects which may include lists of objects and may be arranged hierarchically. This seems to be a valid assumption for many of the objects found in bibliographic data bases. We call the object specified in this way an array type or, simply, an array. A particular set of data conforming to the structure of the array type and stored in it is called an *array instance*. The following is a potential array instance of type "Papers".

(data and information, sigir, (smith browne)
(title (data information)
abstract (database interactive retrieval))

In our examples, user defined terms begin with a single upper case letter, data is in lower case and words of the data definition and query language are all in upper case.

While the basic data structure is a hierarchical one, our model differs from the classic hierarchic model in a number of fundamental respects. Most importantly we permit independent hierarchies. That is, a particular data base will generally consist of a collection of arrays where there is no explicit linkage among the different array types. In this respect the model is analogous to the relational model. Also, and again there is an obvious analogy to the relational model, we permit new array types to be created dynamically. Thus, not surprisingly, there is a certain similarity between our proposed query language and that of SQL which is probably the best known relational query language.

4. THE QUERY LANGUAGE

Superficially, the basic select operation resembles the equivalent operation in the relational model as represented by SQL. For example, to retrieve all titles by Smith we would write:

```
SELECT Title
       FROM Papers
       WITH "smith" IN Authors
```

The general structure of queries is similar to this one. We specify which attributes we want, where they are coming from and what conditions must be satisfied by the array instances containing them. A number of operators can appear in conditions, but "IN" is probably the most useful and the only one we need in subsequent examples. Basically, it tests to see if its first operand is contained within the list specified by the second operand.

Conditions can be applied to any attribute of the array and the usual Boolean connectives can appear with multiple conditions as in:

```
SELECT Title
       FROM Papers
       WITH "smith" IN Authors
       AND "abstract" IN Fields
```

However, because we are retrieving from a hierarchy, questions of *context* arise. For example, it is not immediately obvious what the following query might mean:

```
SELECT Title
       FROM Papers
       WITH "abstract" IN Fields
       AND "database" IN Terms
```

Do we mean a paper containing an abstract where the abstract contains the term "database", or do we mean a paper with an abstract and also containing the term "database" though not necessarily in the abstract? In fact the query would be interpreted as meaning the latter. Conditions are applied independently of each other within a particular array instance unless specific provision is made to establish a context. Context is established by a "WITH" connective. For example:

```
SELECT Title
       FROM Papers
       WITH "abstract" IN Fields
       WITH "database" IN Terms
```

Here the second condition is applied in the context established by the preceding condition. That is, we would look for the term "database" in the array of terms associated with the "abstract" field.

Also the meaning of the following query is not obvious:

```
SELECT Title
       FROM Papers
       WITH "database" IN Terms
```

An array of terms is associated with each field instance. In a typical instance of "Papers" there will not be one list of terms, but rather a list of lists of terms. The question arises then as to what it means when an operand is a list of lists. Again we need to establish context and where there are multiple contexts we need to *quantify* which if any contexts are to satisfy the condition. The correct version of the above query is:

```
SELECT Title
       FROM Papers
       WITH ANY Fields
       WITH "database" IN Terms
```

In our query language operands, other than atomic values and simple lists, are augmented with quantifiers. The most important of these are ALL, the universal quantifier, and ANY, the existential quantifier. If no quantifier is explicitly provided, ANY is assumed. In this case the meaning of the example is obvious. Had we wanted to require the term "database" to appear twice we would have written "ANY 2", and if we had wanted it to appear in every list of terms we would have replaced "ANY" with "ALL".

Because of the frequent occurrence of this type of query, the first form is automatically interpreted as being equivalent to the second. Strictly speaking, the query should have been written as:

```
SELECT Title
       FROM Papers
       WITH Papers
       WITH ANY Fields
       WITH "database" IN Terms
```

However the "WITH Papers" is obviously redundant and would not normally be supplied except in the case where we are retrieving from more than one array as shown in later sections.

A more complex example illustrating context and quantification is the following where we want to find all titles by "smith" or "browne" containing the terms "model" and "data" in the field "keywords". This can expressed as:

```
MYDATA: SELECT Title
        FROM Papers
        WITH ANY <"smith", "browne"> IN Authors
        AND "keywords" IN Fields
        WITH ALL <"data","model"> IN Terms
```

The optional name preceding SELECT is the name given to the retrieved data. The result of a select operation is itself an array, so it too can participate in later selections.

5. RETRIEVAL FROM MORE THAN ONE ARRAY

Our examples so far have all shown retrieval from a single array. However, there is no intrinsic reason why more than one array may not be involved. The major restriction is that if a specific sub-array is being retrieved, it must be common to all the arrays from which retrieval is taking place. If the retrieved array is not wholly contained in the source array, null values will be retrieved for the missing attributes. Any conditional test of a field not contained in one of the arrays is automatically considered to have failed. For example, if we had an array "Books" defined as follows:

```
Books: ARRAY (Title (Authors) Publisher (Topics))
```

It would then be possible to retrieve all the common information in this array and the "Documents" array by writing:

```
SELECT Title (Authors)
        FROM Books, Papers
```

An attempt to select a non-existent attribute from an array will result in a null value being retrieved. For example:

```
SELECT Title Publisher Journal
        FROM Books, Papers
```

Here either Publisher or Journal will be null depending on from which array we are selecting. Another example is the following:

```
SELECT Title
        FROM Books,Papers
        (WITH Books
        WITH "smith" IN Authors)
        OR "browne" IN Authors
```

Here we are selecting any type of document with "browne" as one of the authors as well as any books with "smith" as an author. Note the use of parenthesisation here. Conditions are applied left to right without precedence unless parenthesisation is used. Also the first condition shows an example of an array name being required as a condition since we need to establish the array "Books" as the context in which the following condition is to be applied.

This type of retrieval from more than one array is obviously useful. However, a major limitation is that either common subsets of attributes must be retrieved or a "pseudo-document" containing possibly many null attribute values must be created. An alternative mechanism is that provided through the use of references.

6. REFERENCES

Another major difference between the array model and the classic relational model is the ability to handle *indirection* in the former. It is often desirable to be able refer to information, either in whole or in part, without maintaining a physical copy of the information. In the array model this ability is provided by *reference* mechanism. A reference is simply a type of pointer which identifies an array instance.

The simplest use of references is in a SELECT statement, where they can reduce the amount of data actually retrieved. For example, if we have:

```
SELECT Title (Authors)
        FROM Papers
```

This would cause a complete copy of the relevant information to be retrieved. On the other hand, if we write:

```
SELECT
        FROM Papers
```

Here only pointers to the data will be retrieved. Logically, there is no difference from the user's point of view between a copy and a reference. The only discernible effect will be that updates to the original affects references but not copies, which may or may not be a disadvantage depending on the context.

We can also write the select statement in the form:

```
SELECT REF attribute_name FROM etc.
```

This retrieves references to a node within an instance. All attribute values at or below this node are accessible. For example, we could have:

```
My_query: SELECT REF Field
        FROM Papers
        WITH "War and Peace" IN Title
```

What this effectively does, is make My_query the name of an array whose contents are all the fields, terms and positions of this title. We could now write:

```
SELECT Terms
        FROM My_query
        WITH "abstract" IN Field
```

This example is somewhat contrived. The main use of this facility is for navigation through a hierarchy such as a file organisation as shown in the next section.

Earlier we gave an example of an array declaration. It is often desirable to provide additional specifications for attributes regarding sort order, uniqueness, optionality and so on. This is done by placing *descriptors* in the array declaration.

Descriptors follow the name of the attribute to which they apply. A more complete example of the previous declaration is:

```
Papers: ARRAY
        (Title REQ UP; (Author INDEXED)
        (Fields DISTINCT (Terms UP DISTINCT)))
```

Here REQ means the attribute must have a value in each array instance; UP (or DOWN), means the list of attribute values inside an array instance are maintained in ascending (or descending) sorted order; DISTINCT means no duplicates are allowed inside each particular instance; INDEXED means a fast access technique is provided.

An array attribute can be a list of references. References are specified using the REF data descriptor. Only a reference to another array can be stored in such data. For example, a file containing documents of different types might be specified by the following array:

```
Topic_list: ARRAY
        (Topic DISTINCT
         (Name DISTINCT (Contents REF)))
```

Here the array consists of a list of topics. Under each topic is a list of names and associated with each of these is a list of references to other array instances of any type. Examples of the use of this array are given below.

7. FILE MANIPULATION

We will now show how the types of operation associated with typical office file organisations can be carried out within our model and how this is applicable to retrieving documents from multiple data bases.

Items are added and deleted to an array using a single command. Its syntax is:

```
FILE [REF] [COPY] source
        IN array_name
        AT target
        [WITH conditions]
```

If a "COPY" is specified, then the array instance is copied from the source otherwise it is moved from the source. If a "REF" is specified then a pointer is filed rather than the the actual array instance except that if the array instance is itself a reference, REF has no effect. The source may be an array name, a select statement or a literal. The "array_name" is the name of the array being updated and the target is a list of the attributes of this array which are to be modified. No structural information is required since this is implicitly supplied by the array name. The target must be compatible with the source. That is, the attributes being modified must have the same structural relationship in both the source and the target. The target may optionally have conditions applied to it.

We will now illustrate how the array model features as they have so far been described can be applied to managing information in a file organisation. Here we mean by file management, the type of operations we would typically perform in an office filing system.

Suppose we want to take our two arrays, Papers and Books and organise these by topic where, within each topic, works by the same author are grouped together. A possible array structure to handle this organisation is the example, "Topic_list", specified in the preceding section. Information can be added to this array by locating documents in the original arrays and filing references to them. For example, to move copies of all of the Books array, we would write:

```
FILE COPY
        SELECT Topics (Authors (REF Title))
        FROM Books
        IN Topic_list
```

Here the SELECT part of the statement extracts the individual topics and the authors from the array "Books". A reference to each "Title" associated with a particular author is also retrieved. Note that the information retrieved from "Books" is structured differently from the original. This *reshaping* operation is permitted in any SELECT statement. In reshapes, duplicate parent nodes are eliminated and their children are merged. The information is appended to the array "Topic_list".

Next we might want to add to this file, all papers containing at least one of "Topics" in its "Terms". First, we select each term and references to papers containing the term.

```
Temp: SELECT Terms (Authors ((REF Title))
        FROM Papers
```

Next we file each of these array instances in "Topic_list".

```
FILE Temp
        IN Topic_list
        AT Topic
        WITH Term IN Topic
```

Now to retrieve, say, the titles of all books about a particular topic, "database" for example, we would write:

```
SELECT Title
        FROM Topic_list
        WITH Books
        WITH "database" IN Topic
```

This query is again a case where we need to specify the array name as a condition since there are references to array instances of different types within the array of references "Topic_list".

We can also navigate through the file.

```
Dbase: SELECT REF Name
        FROM Topic_list
        WITH "database" IN Topics
```

This effectively makes "Dbase" a reference to the list of "Names" under this particular topic. If we have the following query:

```
SELECT Title
        FROM Dbase
        WITH "smith" IN Name
```

This will select all titles about "database" by Smith, irrespective of the document type. It is also possible to query lower levels of both document types, as for example:

```
SELECT Title
        FROM Dbase
        WITH "ACM" IN Publisher
        OR "IBM" IN Organisation
```

In the FILE statement, the source and target may be in the same array. For example:

```
FILE SELECT Name FROM Topic_list
        WITH "database" IN Topic
        WITH "smith" IN Name
        IN Topic_list
        AT Name
        WITH "operating system" IN Topic
```

What this statement does is move the instance of "Name" whose value is "smith" from the topic "database" to the topic "operating system".

8. SUMMARY

Hierarchies are fundamental to the handling of documents. They occur in at least two contexts. One is in the structure of the document itself. The other is in the file structures we create to store and retrieve documents in an office environment. Both of these structures can be created and manipulated in a straightforward fashion in the array model and linkages between the two can be accomplished quite naturally through the use of references. Since hierarchies are themselves created by the use of internal pointers, it is relatively straightforward to extend our model to permit the explicit use of these pointers and this is in fact all that references are.

The array model is in some ways a generalisation of the relational model to permit the handling of hierarchies. The model is a view of data and says little about the underlying file organisations. A variety of physical implementations are possible. It would be feasible to "layer" our query language on top of the relational model, or to use the types of file structures used to implement either the hierarchic or network models. We tend to compare our model with the SQL type of relational model because of our emphasis on a high level query language with the property of *closure*.

As the above examples have illustrated, it is possible to perform quite sophisticated operations in the array model using only a few conceptually simple constructs. Most previous work in this area has involved the use of the relational model which views the world as being essentially tabular. For example see, [1]. However it is obvious that this is not a natural mechanism for handling hierarchic structures. This has been recognised for some time and various proposals have been made to extend or modify the model, [4,9,11]. However, the array model seems to provide a more realistic view of data than that offered by the relational model, at least certainly in the context of handling multiple data types and hierarchic file organisations.

The efficiency of an implementation will largely depend on the underlying physical file structures. Since we plan to use the types of constructs current in existing implementations of other models, there should be no degradation of performance in comparison with these. However it is unlikely that an implementation of the array model will ever be as efficient in terms of basic performance as a special purpose document retrieval system of the type in current use. On the other hand we gain a great deal in terms of flexibility. The model is currently under implementation. The underlying file organisation has been implemented and the basic array representation has been built on top of this. In parallel, a prototype system is being developed on top of an existing relational system (INGRES), which will permit a rapid evaluation of the query language constructs.

BIBLIOGRAPHY

[1]. Barnard, D.T. and Macleod, I.A. "A Methodology for the Development of Office Information Systems", *Proceedings of the Canadian Information Processing Society*, pp. 127-134, 1982.

[2]. Chamberlin, D. and Boyce, R. "SEQUEL: A Structured English Query Language", *Proceedings of the 1974 ACM-SIGMOD Workshop on Data Description, Access and Control*. Ann Arbor, Michigan, (May 1974), pp. 249-264.

[3]. Codd, E.F. "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM*, Volume 13, pp.377-387, 1970.

[4]. Codd, E.F., "Extending the Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, Volume 4, 1979.

[5]. IBM Corporation, "Information Management System / Virtual Storage General Information Manual", IBM Form No. Gh20-1260.

[6]. Macleod, I.A. "A Model for Integrated Information Systems", *Proceedings of the 9th Conference on Very Large Data Bases*, pp.280-289, Florence, 1983.

[7]. Macleod, I.A. "AQL - A Query Language for the Array Model". Technical Report, Department of Computing & Information Science, Queen's University, Kingston, Ontario, 1984. (Copies available on request from the author).

[8]. More, T. "A Theory of Arrays with Applications to Databases" Tech. Report G320-2106, IBM Scientific Centre, Cambridge, Mass., 1975.

[9]. Schek, H.J. and Pistor, P. "Data Structures for an Integrated Data Base Management and Information Retrieval System", *Proceedings of the Eighth International Conference on Very Large Data Bases*. Mexico City, (September 1982), pp.197-207.

[10]. Shipman, D.W. "The Functional Data Model and the Data Langauge DAPLEX" *ACM Transactions on Database Systems*, Volume 6, pp.140-173, 1981.

[11]. Zaniolo, C. "The Database Language GEM" *ACM SIGMOD Conference Proceedings*, pp.207-218, 1983.