A Model for Integrated Information Systems

Ian A. Macleod

Department of Computing & Information Science
Queen's University, Kingston, Ontario K7L 3N6.

ABSTRACT

The application of the traditional data base models in the important areas of document retrieval and office information systems has not yet yielded great evidence of success. Here we present an alternative model based on array theory. This model appears to be better suited toward other types of information system,while at the same time, it is still applicable to conventional data base operations. An outline of the model is presented, a description of a suitable query language is given and some implementation issues are discussed.

1. INTRODUCTION

Traditional data base management applications have been reasonably well served by the major data models. However, it is not obvious that these models are appropriate for other information system applications. Two such important applications are document (or information) retrieval and office information systems, hereafter referred to as DRS and OIS, respectively. The work described here has evolved out of the author's experience in developing document retrieval systems within a relational context, [5,6], and, more recently, in modelling an electronic filing system, [1].

Of the three traditional models, the relational has the most attraction as a possible general basis for information systems because of its dynamic flexibility and powerful query languages. We will look at some of the shortcomings of the relational model in the context of DRS and OIS and propose an alternative model, which, as shall be shown, is not totally incompatible with the relational model.

The first problem that arises with the relational model is that caused by normalisation. This has already been noted in other work, [10]. Objects in DRS and in OIS do not naturally lend themselves to normalisation. A form often contains lists of information. So, normally, do documents. For example, a typical object in a DRS might be a document consisting of a title, a number of authors and a set of index terms. In 1NF this could be represented by the relation

```
+-----+-------+-----+
|Title|Authors|Terms|
+-----+-------+-----+
```

This relation is not optimally normalised. In 3NF, it is represented by the following three relations

```
+--+-----+
|Id|Title|
+--+-----+

+--+------+
|Id|Author|
+--+------+

+--+----+
|Id|Term|
+--+----+
```

The problem now is that what was once a logical unit of information is dispersed

across several relations and a new "artificial" attribute, the "Id", has been introduced. Some efforts have been made to reduce this problem. SQL provides "views" which are basically higher level objects derived from the join of basic relations, [2]. Also, Codd's extended relational model provides "E-relations" and "P-relations" where the former are again higher level objects, constructed from the join of the basic P-relations, [3]. These are both somewhat artificial approaches, since there are circumstances where the user must be aware of the disjoint nature of the underlying information.

The alternative approach of keeping the document in 1NF is equally unsatisfactory. In addition to the usual data management problems, the document is now scattered over a number of tuples. From the point of view of a worker in document retrieval, the best solution is to maintain the document, as a logical unit, in its original un-normalised form. (At the often neglected level of data display, it will be noted in passing, that such a representation largely obviates the need for any complex mechanism for displaying documents, as the original "natural" representation is never lost.)

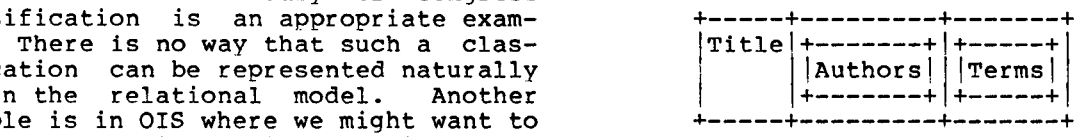
A second problem is that of hierarchy. The "pure" relational model has no such concept yet it is a very important one in many applications. In document retrieval, for example, an important way of classifying documents is hierarchical. The American Library of Congress classification is an appropriate example. There is no way that such a classification can be represented naturally within the relational model. Another example is in OIS where we might want to represent the hierarchical filing systems typically found in offices.

A third problem is that of type. Records are normally classed by type into files, mainly because this is a satisfactory approach to many applications and is the way things have always been done. Programming languages have evolved features to handle this approach to data organisation. However it is a somewhat artificial approach in some applications. 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. (Even libraries have difficulties with this problem, though it deals more with difficulties in managing physical storage and in cataloguing.) A more obvious 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 file folders contain is a number of physically quite different objects, related by their content rather than their structure.

In some ways current data models 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.

2. THE ARRAY MODEL

The obvious solution to the problem of document representation is to retain the document in its original form. We could, for example, represent our previous illustration of a document as follows:



In this notation, the inner boxes denote non-atomic, un-normalised objects. An alternative notation is:

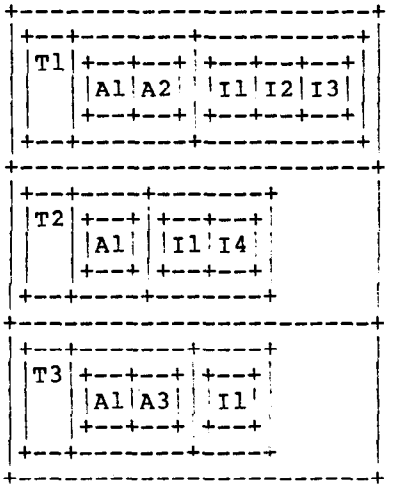
(Title (Authors) (Terms))

The implication is that the document is now a hierarchic object. What is needed then, is a suitable model for the representation of such objects. The obvious candidate is the hierarchic model. However, what is proposed is not an organisation where all information is hierarchically connected but rather an environment where the basic data objects are a collection of, possibly, independent hierarchies. The model is more closely akin to the relational model. The principle difference is that instead of tables, (or relations), there are

hierarchies. In other respects the two models are similar. In particular, there is the property of closure so that the results of a retrieval operation would itself be a hierarchy.

Rather than develop a totally new model, the proposal is based upon the array theoretic model first suggested by More, (for example, see [8]). This model has evolved out of attempts to generalise APL. The basic data element in array theory is an array whose elements may themselves be arrays or data of arbitrary type. A language, known as Nial, based on the theory has been designed and implemented on a variety of machines, [4]. In the context of generalised information systems, this array based approach is of interest because it appears to be a more realistic view of data.

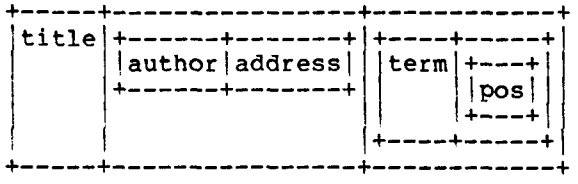
Suppose there exists an array of the type described earlier, containing titles, authors and index terms. The following diagram illustrates an array containing three such titles together with associated authors and index values.



The data is represented as a vector where each element of the vector is itself an array, in this case an array of three elements. Each of these elements corresponds to one of the attributes of the data, so that "Authors", for example, refers to the second array element. This too happens to be an array of author names.

We noted earlier that there are a number of problems with the relational model: normalisation, hierarchy and type. We will now examine each of these problems in the context of the array model.

(i) Normalisation: Since we now allow non-atomic attributes, this problem no longer exists. Array elements can be arbitrarily complex objects. For example, a document consisting of a title, authors and their addresses, index terms and their positions is representable as follows:



If this is how the user views the object, then it is preferable to represent it in a similar way rather than decompose it into smaller objects at the cost of losing the original structure. In the relational model, the equivalent information would require a number of relations and it would be non-trivial to reconstruct the original "document". Furthermore, any external representation of the object in a "natural" form will require a separate process outside the context of the model. It is not possible to integrate retrieval and display in a sensible way. The lack of integration will degrade interaction if a user's subsequent behaviour is partially dependent on the display of intermediate results, as is often the case in document retrieval.

(ii) Hierarchy: Hierarchy is not a problem within the array model. Arbitrarily complex hierarchies are permitted, but at the same time, hierarchies are not required. It is perfectly feasible, and for many applications quite natural, to represent data by "flat files". Hierarchy, as we shall see, introduces no additional complexities into the retrieval langauge. It is possible to ascend and descend trees in quite natural ways.

(iii) Type: Array theory has little in the way of type restrictions. In particular, it is not required that all the elements of an array be of the same type. They need not even be of the same structure. Furthermore, there is nothing in the theory that prevents indirection. Array elements can be array names and the contents of these array names can be accessed. In the data base context, this provides a "clean" method for representing references to objects of

different types.

It should be emphasised at this point that the array model is not an ad hoc solution to the problems of relational models. Array theory has a solid mathematical basis, [4]. At the same time however, it is necessary to demonstrate that this representation is in fact beneficial to the user. In the remainder of this paper we will examine the suitability of the model, what type of query language is appropriate, (the Nial language mentioned earlier is analagous to a data sublanguage rather than to a query language), and some implementation considerations.

3. ARRAY OPERATIONS

3.1 Array Description

In principle, arrays are dynamic objects and can be created at any time. In practice, in a multi-user application, some control will be needed. Arraya are specified by a statement of the form:

ARRAY name IS (structure)

Here structure is a parenthesised list structure defining the attributes of the array. Each parenthesised sub-list can be preceded by a name. Each attribute can be followed by a list of data descriptors. The named sub-arrays are called "twigs". This mechanism permits the same sub-array to be common to a number of arrays, if this is felt to be desirable. It also allows each sub-array to be treated as an array. (By always referring to twigs, the entire structure relationship can be ignored so that the data base can be considered as being basically relational.) For our earlier example we might have:

```
ARRAY Documents IS
   (Titles;
   Author-names (Authors);
   Index-terms (Terms))
```

The purpose of the data descriptors is to provide additional specifications regarding sort order, uniqueness, optionality and so on. One interesting feature of array theory is its ability to handle missing data in a consistent way. This is done through "faults". A fault is specified by a data descriptor consisting of a string preceded by a "?". A more complete example of the previous declaration is:

```
ARRAY Documents IS
   (Titles REQ UP;
   Author-names (Authors ?Author)
   Index-terms (Terms UNIQUE ?Term))
```

Here UP, (alternative is DOWN), specifies the array to be sorted by ascending value of "Titles". A value is required (REQ). The value of "Authors" is optional but if it is omitted, the fault "?Author" will be stored. Faults are propagated across array operations. For example, a count of all authors will return the value "?Author" unless explicit provision is made to check for faults. In the specification for "Terms", the data descriptor UNIQUE implies that the same term can only occur once within each sub-array.

The same attribute-name can appear more than once in an array. In this case, to avoid ambiguity, it will be necessary to qualify the attribute by its twig name whenever it is used.

3.2 References

A problem with any data base is how to handle temporary results. This is a particular problem in document retrieval where Boolean search strategies are normally iterative and a large number of such temporary results are accumulated. It is also a problem in probabilistic searches where results are ranked in order of relevance to the user query. In this type of search environment, it is commonplace for a user to "retrieve" a very large number of documents yet only look at the first few.

This problem would be even worse in the array model where conceivably the entire data base could be retrieved in some applications. Consequently, an explicit mechanism is provided for distinguishing between copies of and references to the actual data. In the former case a new copy is made of the data which then becomes completely independent of the original piece of data. In the latter case a reference to the original data is retrieved. In a retrieval by reference, an update operation will affect all references to the same object. (This may or may not be a disadvantage depending on the context.) A value can be obtained from a reference at any time.

References have names. A reference may refer to an entire array, or it may refer to a sub-array. Reference names are declared by a statement of the form:

```
REFERENCE name IS
   sub-array OF array-list
```

The IS clause can be omitted, in which case the reference is to the entire array. The OF clause can also be omitted at the same time, in which case the reference can be to any array in the data base. A sub-array is any sub-tree of the array, and must be common to all the arrays if it is associated with more than one array.

This idea of referencing parts of an array also handles the problem, noted earlier, of retrieving multiple types. Reference lists may be generated as the result of a retrieval, but they can also be stored in other arrays. References can also be 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:

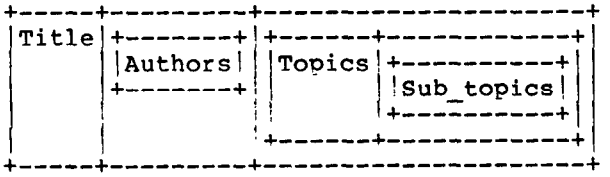
```
File IS (Name REQ UNIQUE;
   (Folder REQ UNIQUE UP;
   (Contents REF)))
```

This mechanism defines the overall structure of the array but allows references to sub-arrays of any type to be attached to it dynamically. Examples of this usage are shown later. 4. BASIC RETRIEVAL

In this section the capabilities of the proposed query language are informally outlined. In many ways, array theory is a generalisation of set theory, so that many of the relational language constructs carry over. We have tried, as far as possible, to model our language on SQL, [2], and, to a limited extent our proposed language can be regarded as being an extended version of SQL. The full query language, AQL, [7], is described elsewhere.

Initially as an illustration, the information stored in the array "Books", shown below will be used. Here there is a set of documents consisting of a title, authors, topics and sub-topics.

Books:



This structure can be specified in AQL by:

```
Books IS (Title[REQ] (Authors)
   (Topic [UNIQUE]
   (Sub_topic [UNIQUE ?Missing])))
```

4.1 Retrieval

The general form of the retrieval statement is:

```
SELECT [name IS] source
    FROM target
    WHERE conditions
```

The name is the name of the retrieved array. It can be omitted in which case it can be implicitly used as the target in the immediately following retrieval operation.

The source specifies the information to be retrieved. It can be a sub-array, a reference, the result of a function, or a combination of all three. For a sub-array this is a "retrieve-by-value" operation. A new copy is made of any data to be retrieved. Data descriptors may be specified in the source, otherwise the corresponding target descriptors are inherited. This field is optional. If omitted the entire array is copied.

The target is a list of one or more arrays from which retrieval is to take place. If omitted, the previous result is used as the target.

The conditions specify what conditions various attributes must satisfy before retrieval of an array element occurs. This field can be omitted in which case all the array elements are retrieved.

4.2 Simple Selection

Simple selection is analogous to projection in SQL. It allows sub-arrays to be retrieved. For example, to select all titles:

```
SELECT Title
   FROM Books
```

No filtering of duplicates takes place. However, array descriptors can be attached to the source. For example, to retrieve a list of individual authors:

```
SELECT Authors [UNIQUE]
   FROM Books
```

An array structure may be retrieved.

```
SELECT Title (Topics)
   FROM Books
```

This retrieves a portion of the original array. Parent-child relationships do not have to be followed exactly.

```
SELECT Title (Sub_topics [UNIQUE])
   FROM Books
```

This selects each title and its associated sub-topics. The data descriptor, UNIQUE, is included since the terms might not necessarily be unique.

Not only need parent-child relations be modified, they can be inverted. Such inversion is called reshaping. For example, the array of papers can be reshaped to give:

```
(Authors (Title))
```

There is no explicit reshape operator. It is only necessary to specify the reshaped array as in:

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

Duplicate parent values are automatically discarded during a reshape operation.

4.3 Conditional Selection

Retrieval may be based on parts of the array being required to satisfy certain conditions. The usual Boolean operators are allowed between multiple conditions so in general the conditions are written as:

```
condition [AND|OR|NOT condition]...
```

In its simplest form, a condition is a predicate written in a subset of the Nial language. A predicate can be applied to any node in the array hierarchy. For example:

```
SELECT Topics
   FROM Books
   WHERE  "War and Peace" IN Title
```

Here the array of "Title"s is searched for a particular value. When a particular Title is found, the array instance in which it occurs is also available. An array instance is a particular value of the upper level element(s) of the array together with all "descendants" of that value. In the "Books" array, each title will have associated with it a vector containing authors, a vector of topics, each element of which has attached to it a vector of sub-topics. In this particular example, assuming titles are unique, the vector of topics associated with this particular title is retrieved.

As a further example, to retrieve every book on the topic of "computers":

```
SELECT Title
   FROM Books
   WHERE "computer" IN Topics
```

This example differs from the previous one in that there is not a single array of "Topic"s but rather one array for each title. Thus, the operation is iterative in the sense that it is applied to each array in turn. Going down one more level:

```
SELECT Title
   FROM Books
   WHERE "computer" IN Sub_topics
```

Here there is not just an array of arrays but rather an array of arrays of arrays. Because this particular subtopic value may be associated more than once with the same title, the same array instance may be retrieved several times. If this is undesirable, as it presumably will be in this case, a quantifier can be used to avoid unnecessary selections, (see "Quantifiers", below).

In general a predicate is a sequence of operators involving a single node. Various useful functions can be used in predicates. For example, to select titles by more than one author

```
SELECT Title
   FROM Paper
   WHERE TALLY Author > 2
```

Evaluation of a predicate is strictly left to right unless parenthesisation is used. The result of a function can also be retrieved. For example, to retrieve each title and a count of the number of authors:

```
SELECT Title, Count FROM Papers
   WHERE Count GETS TALLY Author
```

A further point to note here is that the result of a predicate is assumed to be true if it is not false, so that for this example, there is no need to add a dummy test to the predicate in order to ensure a true result.

4.4 Quantification

As we noted above, predicates are applied iteratively where there are a number of instances of the array. This is not always desirable as was shown in the example:

```
SELECT Title
   FROM Books
   WHERE "computer" IN Sub_topics
```

Here, since there are potentially a number of arrays of sub-topics associated with each title, the same title may be retrieved a number of times. To avoid this, the selection process can be quantified. There are three quantifiers, EACH, ANY and NOT_ANY. Quantifiers appear before an array operand and quantify the number of times individual instances of arrays are to be tested. EACH, the default, implies that each array is examined; ANY that testing proceeds until one array occurrence satisfies the condition; and NOT_ANY implies that testing continues as long as no array instances satisfy the condition. Further, both ANY and NOT_ANY, may be numerically qualified by succeeding them with an integer value which indicates how many array instances must satisfy the condition.

For example, the previous example can be modified so that only one title is retrieved, by writing:

```
SELECT Title
   FROM Books
   WHERE "computer" IN ANY Sub_topics
```

4.5 Qualification

Often it is desirable to access a value or array whose position is dependent on some previously accessed value. For example, to retrieve titles indexed by the topic "computer" and the sub-topic "information":

```
SELECT Title
   FROM books
   WHERE "computer" IN Topics
   AND "information" IN Sub_topics
```

However, this query does not guarantee that "information" is a sub-topic of the topic "computer". It only guarantees that there is some topic which has "information" as a sub-topic. If this is not what is wanted, it is necessary to restrict the second predicate to the sub-array satisfying the first predicate. This can be done using the "WITH" qualifier. In general, a condition takes the form:

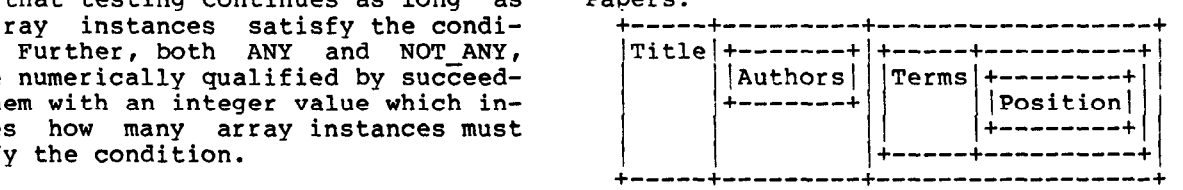
```
predicate [WITH (conditions)]
```

The conditions in the WITH clause are applied to the nodes satisfying the predicate, together with any descendants it might have. The above query would be correctly written as:

```
SELECT Title
   FROM Books
   WHERE "computer" IN Topic
   WITH ("information" IN Sub_topic)
```

Sometimes it is necessary to do more than simply establish a position inside an array. Rather the location must be remembered for subsequent use. For example, suppose there is a set of papers consisting of titles, authors, terms and the positions of the index terms, as illustrated below:

Papers:



Suppose the purpose of the query is to find the titles of all papers containing "information" and "retrieval" with "retrieval" occupying the next position after "information": In this case, it is necessary to remember which array element contained the term "information" so that its corresponding sub-array, "position", can be later accessed. This can be done as follows:

```
SELECT Title
   FROM Papers
   WHERE X IS "information" IN Terms
   AND "retrieval" IN Terms
   WITH (ANY X.Position+1 IN Position)
```

The "X IS" operation is basically a local assignment which identifies all the array elements satisfying the subsequent condition. The notation "X.Position", then refers to any "Position" arrays associated with these array elements.

5. RETRIEVAL ACROSS 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 the sub-array being retrieved must be common to all the arrays from which retrieval is taking place. If the retrieved array is not contained in the target array, nothing from that array will be retrieved. Any conditional test of a field not contained in one of the arrays is automatically considered to have failed. For example:

```
SELECT Title (Authors)
   FROM Book, Paper
```

In this type of retrieval, the identical structure must occur in all the target arrays.

More interestingly, information from differently structured arrays can be combined by creating a virtual array, (see below). For example, suppose we have an array containing the current addresses of authors of the form illustrated below.

```
          Location:
          +----+-------+
          |Name|Address|
          +----+-------+
```

We can now create a new array of the form:

```
(Name; Address; (Title))
```

Conceptually, this array is created by sequencing through both existing arrays in parallel, that is for each distinct value of "Name". The appropriate query is:

```
SELECT Name; Address; (Title)
   FROM Location, Papers
   WHERE EACH Location.Name
         IN Papers.Authors
```

This operation in analogous to the join of SQL, and like the join its efficiency will be very dependent on the physical storage organisation. However, for many applications it should be a relatively infrequent operation because there will be less need to decompose un-normalised data.

6. UPDATING

Updating operations permit items to be added to an array and to be removed from it. DISCARD and FILE operations are provided. An item can be detached from an array by discarding it. The general form of a discard is:

```
DISCARD name
   FROM array
   WHERE conditions
```

Here the name is any attribute of the array. The effect is to discard any attribute(s) satisfying the conditions together with any sub-arrays. For example,

```
DISCARD Title
   FROM Book
   WHERE "smith" IN Author
```

This would remove all the documents and associated sub-arrays where one of the authors is "smith".

Filing is the converse of discarding. The general form a file statement is:

```
FILE [subarray FROM] source
   IN array
   [AT condition]
   [IF condtions]
```

The source is either a single literal value, or is the name of an array containing the data to be filed. The subarray specifies which attributes are being updated, if a complete array instance is not being added. The "AT" condition is only needed where a subarray is being filed. It identifies the location of the update within the array structure. The "IF" conditions can be used to specify integrity constraints.

For example, suppose we want to create a file containing documents of various types with each file categorised by topic. First we specify the file:

```
Myfile IS
   (Filename; Topic; Contents (Ref))
```

Next we can create an instance of a file:

```
FILE "File/1"; "OIS"; "" IN Myfile
```

Next we might collect all documents about "OIS".

```
SELECT Mylist [REF]
   FROM Papers, Books
   WHERE TERM = "OIS"
```

We can then add this set to our file by:

```
FILE (Contents)
   FROM Mylist
   IN Myfile
   AT Topic = "OIS"
```

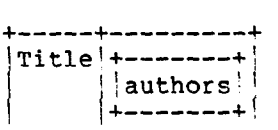
7. RELATED WORK

A great deal of work has recently been directed towards some form of integration. Most of this work has centred around the relational model. Macleod, [5], and more recently Stonebraker, [11], have suggested means whereby relational languages can be applied to document retrieval. However, neither address the basic problem of the suitability of the model. Tsichritsis, [12,13], has concentrated on the integration of messaging within a data base system. A more radical approach is that taken by Schek and Pistor, [10], who have proposed a significant modification to the original model by allowing elements of sets to be themselves sets. This idea largely disposes of the problems caused by normalisation. However there seems to be little value in continuing with the idea of the basic data element being a set since in practice ordering and position are of fundamental importance. The Schek model recognises position by providing template matching which, to be efficient, is based upon an implementation of fragment indexing. The array model permits a lower level view of position and, at the same time, does not require any special implementation structures. An array can be implemented by:

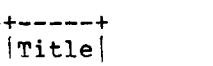
(i) A "picture" or schema defining the array

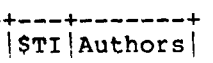
(ii) A set of inter-related tables where each table corresponds to a nested sub-array.

For example, the array:



can be represented by the following two tables:





Each table has a surrogate whose value uniquely identifies each tuple of the table. Additionally, each sub-array element contains the surrogate value of its parent. Thus the surrogates serve as associative addresses for the purpose of representing an array structure.

An alternative aproach to integration is to conceal as many of the problems of the underlying model as possible by using a high level interface. Some of Zloof's work, for example see [14], can be considered in this light as well as some of the existing natural language interfaces. However these approaches are only partially satisfactory. A join in QBE in no more natural an operation than it is in SQL. An advantage of the array model is that it is a more natural representation of data and requires less manipulating at the interface level to be easily usable.

8. SUMMARY

Current data models are inadequate for handling information systems outside the traditional scope of data base management systems. In particular, none have been proposed as the basis of a hierarchical filing system and while the relational model has been considered as a basis for document retrieval systems, it certainly has non-trivial handicaps in this context.

In our view, the array model resolves most, if not all, of the fundamental problems encountered when applying the relational model to these other contexts. It contains many of the characteristics of network, hierarchical and relational models. While it may be tempting to classify the array model as an ad hoc collection of parts of these other models, it most certainly is not. The query language described here is, like most new languages, somewhat experimental, but the underlying model is strongly founded. If the array model has one defect, it is that it is expensive to implement in terms of storage requirements. A large number of associative links and indexes are needed for its efficient implementation. On the

other hand, the costs of such storage have been and are continuing to be significantly less from year to year.

The main motivation for this work has been to further efforts towards the integration of information systems. Given the rapidly expanding use of computing systems in information oriented applications, this is becoming a serious problem, as, for example, anyone with access to a network messaging system knows. There is a real need for flexible tools with which to organise and manage data. Traditional information systems such as DBMS and document retrieval systems tend to be narrowly focussed and lack flexibility. At the present time our model is being prototyped in Nial. The Nial language is an implementation of More's array theory. It has been implemented on a wide variety of machines including IBM PC's. It is also available under Berkeley 4.2 Unix. This prototype is also serving as a model for a realistic implementation using disk for array storage.

REFERENCES

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., "Extending the Relational model to Capture more Meaning", ACM Transactions on Database Systems, Volume 4, 1979.

4. Jenkins, M.A. "The Q'Nial Reference Manual, Tech. Report 82-123, Queen's University, Kingston, Ontario, 1982.

5. Macleod, I. A. "SEQUEL as a Language for Document Retrieval", Journal of the American Society for Information Science. Vol. 30, pp. 243-249, 1979.

6. Macleod, I. A. "A Data Base Management System for Document Retrieval Applications, Information Systems, Vol. 6, pp. 131-137, 1981.

7. Macleod, I.A. "AQL - A Query Language for the Array Model", Technical Report, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1983.

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

9. More, T. "Notes on the Diagrams, Logic and Operations of Array Theory", Tech. Report G320-2137, IBM Scientific Centre, Cambridge, Mass., 1981.

10. 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, pp.197-207, 1982.

11. Stonebraker, M. et al. "Document Processing in a Relational Database System", ACM Transactions on Office Information Systems, Volume 1, pp. 143-158, 1983.

12. Tsichritzis, D. and Christodoulakis, S. "Message Files", ACM Transactions on Office Information Systems, Volume 1, pp.88-98, 1983.

13. Tsichritzis, D. "Integrating Data Base and Message Systems", Proceedings of the Seventh International Conference on Very Large Data Bases, Cannes, pp.356-362, 1981.

14. Zloof, M.M. "Query-by-Example: A Data Base Language", IBM Systems Journal, Volume 16, 1977.