THE MACAIMS DATA MANAGEMENT SYSTEM

Robert C. Goldstein
and
Alois J. Strnad

MAC Technical Memorandum 24

April 1971

(Paper presented at 1970 ACM SICFIDET Workshop
on Data Description and Access, Rice University
Houston, Texas, November 15-16, 1970)

Massachusetts Institute of Technology

PROJECT MAC

545 Main Street Cambridge 02139

THE MACAIMS DATA MANAGEMENT SYSTEM

Robert C. Goldstein
and
Alois J. Strnad

MAC Technical Memorandum 24

April 1971

(Paper presented at 1970 ACM SIGFIDET Workshop
on Data Description and Access, Rice University
Houston, Texas, November 19-16, 1970)

Massachusetts Institute of Technology

PROJECT MAC

545 Main Street Cambridge 02139

The MacAIMS Data Management System

by

Robert C. Goldstein and Alois J. Strnad

I. Introduction

MacAIMS (MAC Advanced Interactive Management System) is a relatively small research project that was initiated in the summer of 1968 to investigate the feasibility of using some of the then existing computer facilities at M.I.T. to aid in the management of Project MAC. Several interesting and useful interactive programs were developed and are currently in use. However, the primary, and not very suprising, result of our early work was the discovery that no existing system offered either the desired highly human-oriented interactive style or the capability of providing very rapid, yet carefully controlled, access to a large data base. Thus, early in 1970, we found ourselves faced with the prospect of developing entirely new information handling tools if we wanted to extend the capabilities of our management system beyond the level that had already been achieved.

The work reported herein was performed at Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, and was supported by the Office of Naval Research under Contract N00014-69-A-0276-0002.

The MacAIMS Data Management System

Multics

At just about the same time that we were deciding to go ahead with this effort, a major new computing facility became available at M.I.T. This new facility, the Multiplexed Information and Computing Service (Multics), was the result of more than five years of work by the Computer Systems Research Group at Project MAC in cooperation with the Bell Telephone Laboratories and the General Electric Company, and is generally conceded to represent a major advance in the area of multi-programming, multi-processing computer systems. It is pertinent to say a few words here about Multics since many of the things we ultimately decided to do would be impractical in a less powerful environment.

Memory Organization

Perhaps the most important aspect of Multics for our present purposes is its memory organization. Nearly all large computers employ a hierarchy of memories, ranging from core through drums, disks, tapes, etc. Each level in the hierarchy is characterized by having larger capacity and slower access than the preceeding ones. In conventional systems, the applications programmer must individually manage his own use of the hierarchy, explicitly moving programs and data from device to device. In Multics, this

approach is replaced by a single level, but two-dimensional, memory organization. It is as if each user had available to him a large number of identical core memories. These memories are referred to in Multics as segments, and a user process may have up to 256,000 segments, each containing, in the current implementation, as many as 64,000 36-bit words. The total size of this address space is therefore 2^{34} words, and a user program may address any bit in that entire range as easily as if it were all implemented in core. Multics also employs a paging strategy for managing the physical memory hierarchy. This means, for example, that it is feasible to provide lengthy blocks of program or data for special situations, secure in the knowledge that except when those situations exist, these blocks will cost us nothing in either core space or swapping time.

Access Control

Another important aspect of Multics from the point of view of the information system designer is its facility for access control. Again, a two-dimensional approach is taken, covering first access to segments, and then access within a segment. Each segment has an associated access control list which identifies all the users authorized to use that segment and also specifies the mode of access for each of them. The four access modes are

read, write, execute, and append. Within a segment access is controlled by a system of concentric protection rings which can be thought of as a generalization of the "supervisor mode, user mode" facility of most modern computer systems. Any attempt to access a segment from a not sufficiently privileged (i.e. outer) ring will cause a trap. A routine provided by the "owner" of the segment will then be called to decide whether or not to permit the specific access being attempted.

With its very large directly-addressable virtual memory and its flexible access control capabilities, Multics provides important new tools for handling large data bases. Let us now turn to an examination of the way in which these capabilities are actually used in the MacAIMS Data Management (MADAM) System. It should be pointed out that all programming for the MADAM system is in PL/1, the primary language on Multics and the language in which the Multics system itself is written.

II. Overview

In our early experiments in the MacAIMS project, we made use of a variety of data structures, including linked lists, trees, and networks with fairly complex interconnections. This diversity came about as a result of our attempt to store each data base in the manner most efficient for it. If nothing else, this certainly gave us a lot of experience handling different types of data structures. As a result, we have concluded that conventional approaches are incapable of providing the necessary capability, flexibility, and efficiency. We have therefore decided to base the MADAM system on a relational, or set theoretic, approach to data base organization. Since this paper is concerned primarily with implementation, it would not be appropriate to go into any detail on the theoretical basis for our system. It also would be unnecessary since several excellent papers on this subject have recently appeared (1,2). In essence, we take the view that all of the information we might wish to store in our data base consists of data elements and relations among them. Furthermore these data elements and relations fall naturally into sets. For example, we might have a set of persons names, and a set of telephone numbers. We might then also have a set of relations associating telephone numbers with people. One obvious advantage of this approach compared to more rigid network

oriented systems is the ease with which exceptional conditions can be handled. For example, the cases of a person who has no telephone number or of one who has several require no special treatment of any kind. A further advantage is that the substantial body of theory concerning sets and operations on them can all be brought to bear on the data management problem.

Access Control

A relational approach to data management also has some attractive implications for the problem of access control. This arises from the fact that the data elements can be stored separately from the relations among them. It would therefore be possible, for example, to give a particular user access to the name set and the salary set without allowing him to associate specific salaries with individuals. In other words, we can protect that particular relation while still allowing the user access to both names and salaries in other relations, for example, a <name,telephone> relation and a <salary,account> relation. As a second example, a user could be given access to all sorts of relations about people for some research purpose, while perfectly preserving individual anonymity by denying access to the name set or any other data element set that would permit identifying the individuals involved.

The MacAIMS Data Management System

Reference Numbers

The overall structure of the MADAM system is presented in Figure 1. It will be noted that the primary division in the system is between those procedure and data segments concerned with data elements and those concerned with relations. The following two sections of this paper will take up these portions of the system in more detail. However, before getting into that, some discussion of reference numbers is in order since they play such an important role in our implementation. Whenever a new data element enters the system, it is immediately assigned a reference number which is used for all subsequent operations on that element. This makes an important contribution to our goal of storage and operational efficiency. In the first place, all reference numbers are of known, fixed length. This is in sharp contrast to the data elements themselves which may vary from one bit to many characters. In our particular implementation, reference numbers are all 36-bit quantities since this is the word length of the computer on which the system is implemented. Incidentally, this does not imply that the total number of items and relations in the system is restricted to 2^{36}. A reference number is only unique within a set of either data elements or relations. Furthermore, if in some exceedingly large system, that limit should be approached, a simple change in the

declaration statement could be used to arbitrarily increase the length of a reference number. The effect of this on performance would depend on the particular hardware involved. In our case, reference numbers could be extended to 72 bits with very little loss in speed. Aside from the efficiency achieved by using fixed length quantities throughout most of the system, we probably also save storage space since the average length of a data element is almost certainly in excess of 36 bits.

The use of reference numbers also materially aids our access control efforts. Since relation sets are expressed exclusively in terms of reference numbers, we can easily give access to a relation while withholding the identity of the actual data items involved.

An obvious apparent liability of our use of reference numbers is the element of indirection that they introduce. We have just seen how that can be used to advantage in controlling access to the data base, but at first glance it might appear that they also introduce substantial overhead whenever an operation on a data element must be performed. In fact, this is not necessarily so. If proper attention is given to the algorithm for assigning reference numbers, the data element itself need never be referred to except on input or output. For example,

probably the most common operations performed on a data base are comparison and ranking. That is, we want to know whether two data elements are equal or if not, which one is greater. If the reference numbers are assigned using a method which preserves order, these operations can be performed directly on the reference numbers themselves. Not only is it unnecessary to refer to the actual data element, but the comparison itself can probably be performed much more quickly on the single-word long reference number than on the variable length data element. Again, of course, our access control capability has been enhanced by our not having to dig out the actual data element.

We should also mention two special reference numbers which turn out to be very useful. One is used to indicate a null, or absent, field. This may arise as a result of certain of the set operations. The other special reference number is represented graphically by an asterisk and acts something like a wild card. That is, it will match anything in its particular field. For example, if I had the relation set <name,country of citizenship>, I could locate all of the Australians by intersecting that relation set with the single element one <*,Australia>.

The MacAIMS Data Management System

Standard Forms

A further interesting property of this scheme is that we have the opportunity to do some preprocessing at the time we assign reference numbers. A standard form is defined for each type of data element and this is the only form in which data is stored in the system. The procedures discussed in the following section of this paper have the responsibility of processing an input item into a standard form prior to the assignment of a reference number. Thus, any two input items which result in the same standard form will have the same reference number assigned and there will be no ambiguity at a later time as to whether or not they are equivalent. One example of the value of this approach is the multitude of different ways in which people write dates. Another, which is a bit trickier, concerns the equivalence of upper and lower case letters in a character string. In some cases this distinction might be important. In others, it should clearly be ignored.

This section has attempted to provide a general introduction to the MADAM system. The rest of the paper will go into greater detail on the representation and manipulation of data elements and relations.

III. Data Elements

A data element, in our terminology, is any basic piece of data that is stored within the system. For example, the name "John" might be a data element; so might the name "Smith". Alternatively, if it were appropriate, "John Smith" could be one. Data elements are represented throughout the system by 36-bit reference numbers. This approach enables storing each data element only once within the MADAM system and also substantially improves the operational efficiency by providing a consistent representation for all data elements.

Data Element Module

A user of MADAM must be protected from having to bother about the detailed data base maintenance functions which must be performed but which are of no direct concern to him. He should not even have to be aware of how the data is represented and organized since, even in an operational system, efficiency considerations may require changes from time to time. The Data Element Module (DEM) provides the direct interface between the user and the part of MADAM that performs these functions.

Each Data Element Module is designed for a particular data type. In other words, each data type, such as person's name,

date, salary, occupation, deparment, field name, etc., may have its own module. The number and types of DEM's depend on the particular application. Each Data Element Module is one procedure segment. As will be seen later, it may have its own associated data base which is a data segment. A DEM has to perform all of the functions which a user needs for storage and manipulation of data elements. We have defined the following set of entry points:

read - read the input stream and produce the standard form. For example, the input to the DEM for dates might be " 6, Jan., 1962.". The standard form returned by this entry is "19620106". A data element can be stored within the MADAM system in the standard form only.

write - this entry will write the given standard form on the output stream in natural, human - oriented form. Taking the data from the previous example, the output would be "January 6, 1962".

get_reference number - the reference number corresponding to the given standard form will be returned.

get_data_element - this entry will return the standard form of the data element corresponding to the given reference number.

insert - the given standard form will be made known to the

specified Data Element Module and a reference number assigned.

delete - the specified data element is logically deleted by setting a status bit.

These entries in a DEM will be used by a "system writer" - a programmer who uses MADAM to set up a particular application. They never appear to the ordinary user of an applied system. Each DEM does not necessarily include detailed coding for all these functions. In many cases, a Data Strategy Module (a procedure segment) may be invoked as an intermediary.

Data Strategy Module

Most data elements fall into one of two basic classes: character strings and integers. (In some applications, a third class, real numbers, might have to be added. Since our initial application is in the managament area, we do not have the problem of scaling numerical data and choose to work exclusively with integers for precision.) The distinction between character string data elements and integer ones is that for integers we can use the data element itself for the reference number.

For character string data elements, the reference number is a 36-bit binary number assigned by the DSM. The leftmost "one" bit

is used as a flag to indicate the start of significance. The actual mechanism for associating a reference number with a particular character string is represented in figure 2. It will be seen that a binary tree is used to store the standard forms. This scheme meets the conditions suggested for reference number assigment algorithms given in section II.

The reference number for an integer data element is the integer itself. The mechanism for storing these data elements in a binary tree is represented in figure 3.

Because of different methods for storing and associating the reference numbers with character strings and integers and because the structures of the nodes are not exactly the same in the two cases, there are two Data Strategy Modules: DSM_string and DSM_integer. Both have the same functions and entry points as a Data Element Module except that the entry points "read" and "write" are omitted. Implementing these two DSM's means that most of the entry points in a typical DEM are nothing more than calls to the equivalent function in one of the DSM's. "Read" and "write" must be handled separately since they are heavily dependent on the "meaning" of the particular data element.

Nodes of a binary tree

We have explained a mechanism for storing a data element as a node in a binary tree and associating a reference number with it. Because the data element may not be the only information in the node, we must describe a node more precisely. As we mentioned earlier, MADAM is written in PL/1. A PL/1 declaration of a data structure is very descriptive and we will use it to indicate the structure of the nodes (figures 4a and 4b). Note that the data element is stored in standard form. As can be seen, the two nodes are very similar. In a node representing an integer the data length and reference number are omitted. The data element in this case is also the reference number.

Data Element Segment

A Data Element Segment is the data base for a particular Data Element Module. A DEM can have at most one DES.

As new data elements are inserted and space for their nodes allocated in the Data Element Segment, the binary tree may became unbalanced. That is, some of the branches of the tree may became much "longer" than others and therefore substantially reduce the efficiency of the MADAM system. The same effect may occur if

there are a large number of logically deleted nodes in the tree. (See the DEM entry point "delete"). Special programs run in a background mode handle this situation by physical deletion of the flagged nodes and by rebalancing the tree.

Figure 1 illustrates the connections among the various types of segments. As can be seen, each Data Element Module uses its "own" DES via a Data Strategy Module. It should be pointed out that a DEM does not have to have a DES. For example, a DEM for salary might have a built-in procedure for encoding the salaries, but the data elements might not have to be stored at all.

There is one DEM for field names. Since field names are no different from any other type of data element, they can be handled exactly the same way. In order to have the system be self-descriptive, the data base must include the identity of the Data Element Module that should be used to process instances of each field name. This connection between a field name and a DEM is stored just like any other relation (see Section IV.) Each DEM must therefore have an associated reference number which is assigned by the DEM for DEM's.

IV. Relations

We have just presented a very elaborate mechanism for keeping track of individual data items and assigning reference numbers to them. Let us now look at the process of representing and manipulating relations among data items. Throughout this discussion, whenever we refer to a data item, we really mean the reference number of a data item. Relations are represented exclusively in terms of reference numbers.

We all know intuitively what a relation is. It is, for example, "John is Paul's father". Technically, however, we should call this an n-tuple (in this specific case, a duple). The terms "relation", or "relation-set", we reserve for the whole set of duples relating fathers and sons.

Having adopted the notion of a set, we can now implement all of the basic set theoretic operations such as union, difference, projection, etc. We choose also to extend the usual mathematical definition of a set by introducing the concept of order among the n-tuples. This is, of course, unnecessary for any of the set theoretic primitives but makes the computer implementation of them much more efficient. It is also useful on input and output since people tend to prefer ordered information. This leads us to define two additional primitives: sort, and

get_successor_relation. Sort takes as input a relation set and a sort key. The key is in the form of a Relation Descriptor. The sorting operation consists of ordering the n-tuples numerically after first rearranging each one in accordance with the new Relation Desriptor (sort key). Get_successor_relation requires as input a relation set and an n-tuple and returns the n-tuple immediately following the given one in the appropriate collating sequence.

Data Structures

Most previous data management systems have been capable of supporting only a limited number of data structure types, usually in fact just one. It should be clear that any data structure type which is at all suitable for storing relations can be used to store any relation. However, depending on the nature of the relation and the type of access required to it, alternative data structures may vary tremendously in efficiency. Since we had no way of knowing as we designed this system how and what it would be used for, we felt obligated to at least provide a mechanism for implementing any type of data structure.

Our basic approach is similar to that discussed in the preceeding section for data elements. That is, there are two types of segments. One type contains the actual relational data

in whatever form the user feels appropriate. The other type of segment contains the procedures that operate on this data. As in the preceeding situation, there are alternative sets of procedures. However, whereas one Data Element Module was required for each type of data element known to the system, a Relational Strategy Module (RSM), so called, is required only for each type of data structure implemented, regardless of the contents of the relation. Thus, while a typical operational system would have a large number of Data Element Modules, it would probably have not more than two or three RSM's. This is fortunate since the latter is a significantly more complicated piece of programming. Each Relational Strategy Module knows how to perform the primitive operations on its type of data structure. The operation of the system is therefore completely transparent to the particular data structure used, assuming only that the appropriate RSM is available.

The Canonical Form

There is one further point which ought to be made concerning data structures. Most of the primitive operations that we have defined require two structures as inputs and produce a third as the output. A potential difficulty arises when it becomes necessary to operate on two sets of relations that happen

to be stored in different types of data structures. It is clearly not feasible to write a Relational Strategy Module that can operate on every other possible type of structure in addition to the one it is specialized for. Our approach to this problem has been to define one type of structure as the "canonical form". Every RSM must be capable of accepting a canonical form as either of its inputs and of producing the canonical form as output. Thus, to operate on two data structures of different types, one of the relation sets would first be converted to canonical form using its particular Relational Strategy Module and then the RSM for the second data structure would be used to perform the actual operation using the canonical form as one of its inputs.

In order to make the job of programming a Relational Strategy Module a reasonable task, we have attempted to define a canonical form that would represent a simplest common denominator. The structure we have decided upon can be thought of most easily as a two-dimensional array where each row is one n-tuple and each column corresponds to a particular field. The canonical form for the relation <father,son> is illustrated in figure 5. It requires exactly two columns, one for fathers and one for sons. In order to identify the columns, we add a zeroth row to our array known as the relation descriptor. It looks

exactly like any other row, except that its entries, rather than being data elements, are field names. These, like the data elements themselves, are referred to only by reference number in the relation data segments. While it is convenient to think of our canonical form as a two-dimensional array, it is actually implemented as a linked list of n-tuples to facilitate insertions, deletions, and sorting. It is likely that when we come to a detailed consideration of inter-computer transmission of relation sets, we will transform the linked list back into an ordinary array in order to eliminate all use of pointers which tend to exhibit a particularly virulent form of machine dependency.

Organizing Relational Data

An important aspect of any data mangement system is remembering just what information is in the system. We have already seen how defining a data element called "field name" allows us to keep track of all the different kinds of data elements in the system without having to create any additional mechanism. It would clearly be nice to do something very similar for relational data. Upon close examination, in fact, the requirement appears somewhat more stringent. For example, if a question comes up which involves the <father,son> relation, we

would like to be able to find out not only whether that specific relation exists in the system, but if not, whether the required relation set is a subset of any existing one, e.g. <father,mother,son>. This question can also be turned around. If we are actually looking for a complicated relation that does not already exist, we would like our indexing scheme to tell us which existing relation sets contain at least some of the required information and can therefore be used to build the desired one.

By direct analogy with the scheme used for data elements, we keep track of all the relation sets known to the system by creating another relation set whose component n-tuples are the relation descriptors of all the known relation sets along with the information required to locate each one. Pursuing the analogy further, there is nothing special about this relation set. It has a relation descriptor whose components might be, for example, "field_1", "field_2", etc. Furthermore this relation descriptor would appear as an n-tuple in the body of the relation set along with all the other relation descriptors. To find out whether some specific relation existed, for example, our <father,son> case, one would merely have to construct the desired relation descriptor, take it as a one element set, and intersect it with the set of all relation descriptors. If one were clever

about using the "wild card" mentioned in section II, one pass could not only identify any occurences of this specific relation set, but also any in which the desired one was a subset. It should be clear how this process can be turned around to select relation sets that are likely candidates for combination when the desired one is more complicated than any in the basic system.

V. Acknowledgements

We would like to thank Professor James D. Bruce, and the Messrs. Burton J. Smith, Andrew I. Fillat and Leslie A. Kraning, all of the M.I.T. Electrical Engineering Department, who first called to our attention the possibilities inherent in a relational approach to data base management. We also wish to acknowledge the contribution of all the other members of the MacAIMS team, particularly Mr. Douglas M. Wells, who was too busy implementing the system to help write the paper. Mrs. Eileen Moore did all of the typing with her usual speed and accuracy.

VI. References

1. Codd, E.F. "A Relational Model for Large Shared Data Banks", Comm. ACM 13,6 (June 1970), 377-387.

2. Fillat, A. I., and Kraning, L. A. Generalized Organization of Large Data-Bases; A Set-Theoretic Approach to Relations, M.I.T. Electrical Engineering Dept. Masters Thesis, June 1970

The entire MacAIMS System is discussed in exhaustive detail in a series of MacAIMS Multics Memos. A complete list is available from the authors.

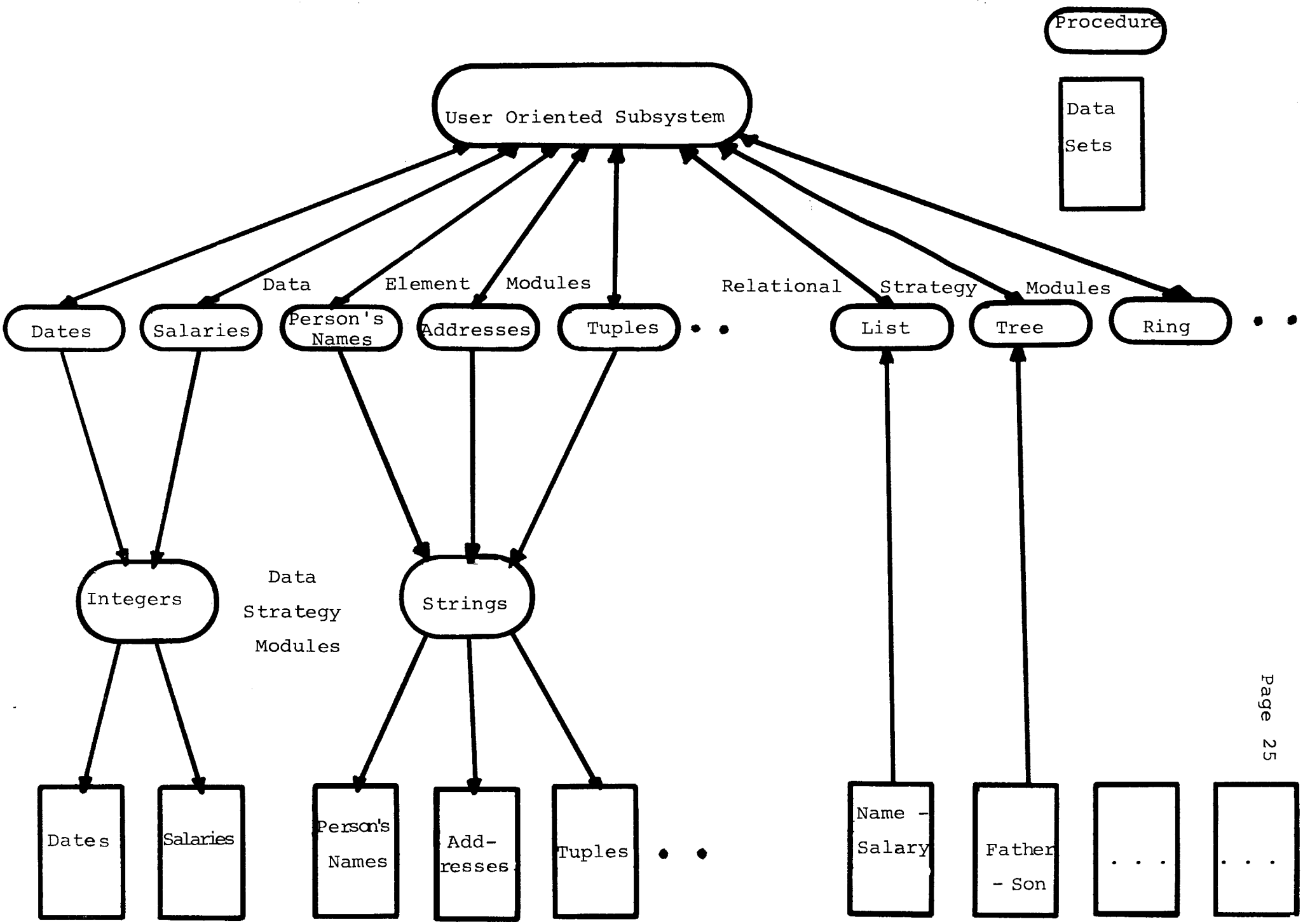
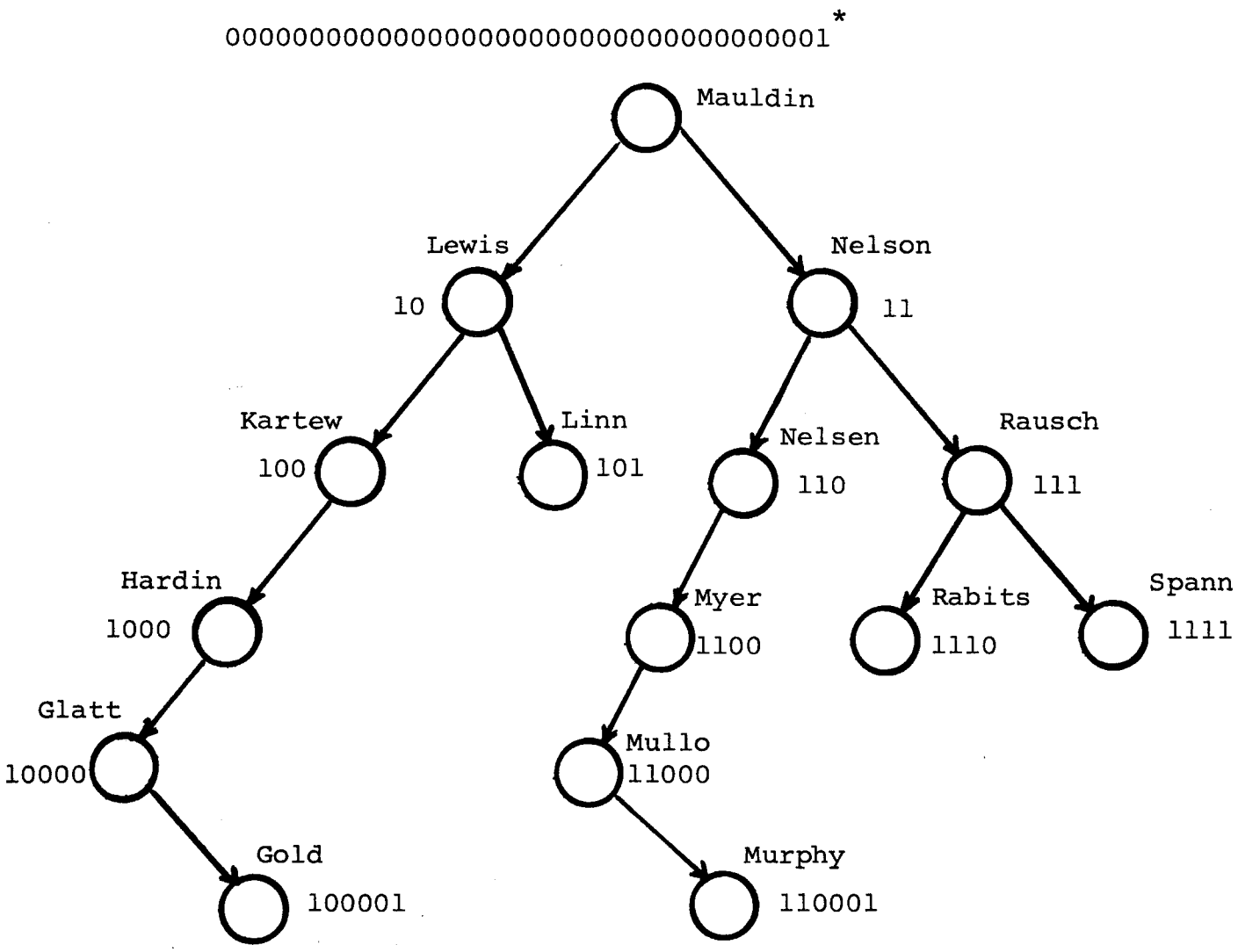


Figure 1 - System Overview

Binary Tree for Character Strings



*In succeeding reference numbers, all bits to the left of the flag have been omitted for clarity. First left-most "one" bit is used as a flag to indicate the start of significance.

Figure 2

Binary Tree for Integers

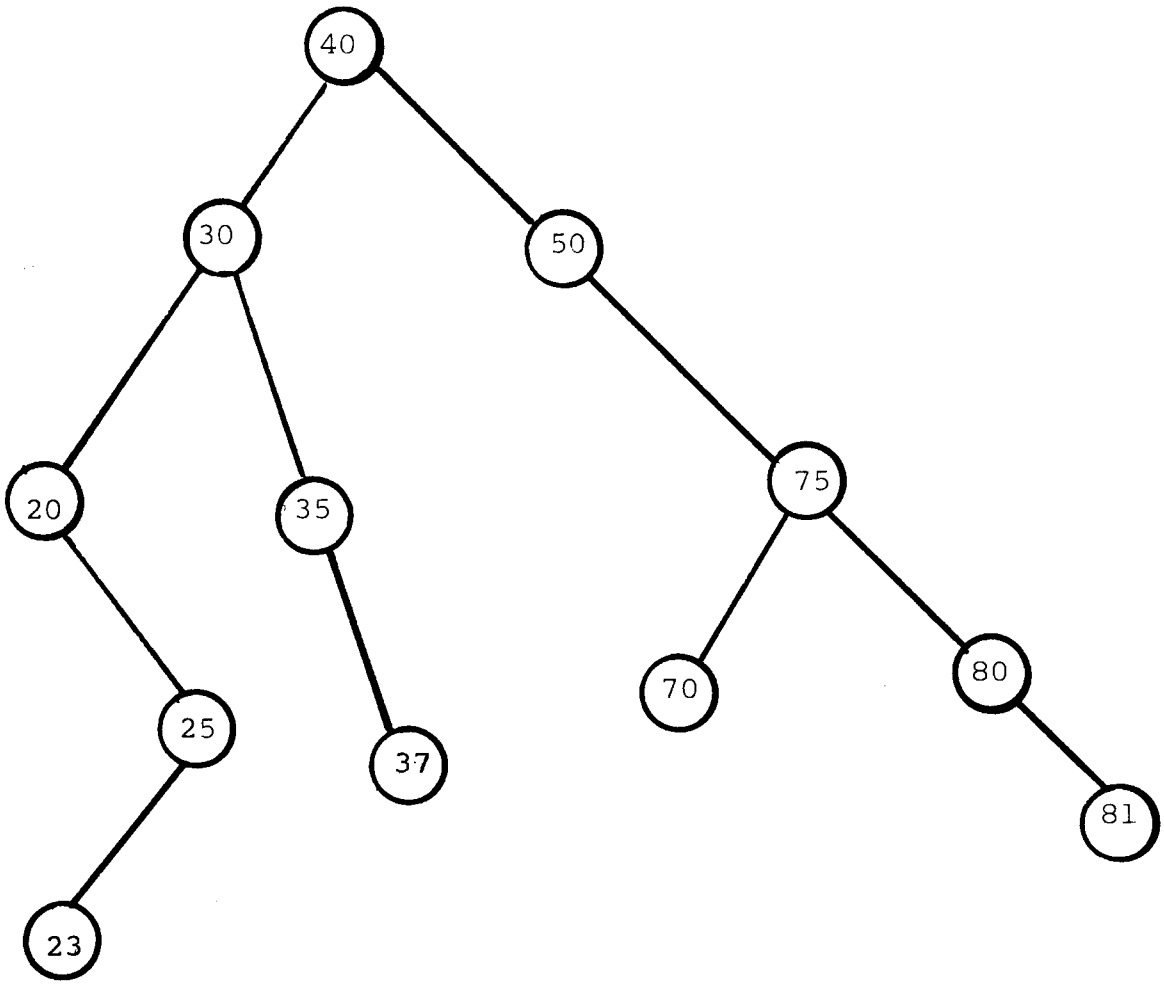


Figure 3

Structure of the node for a character string.

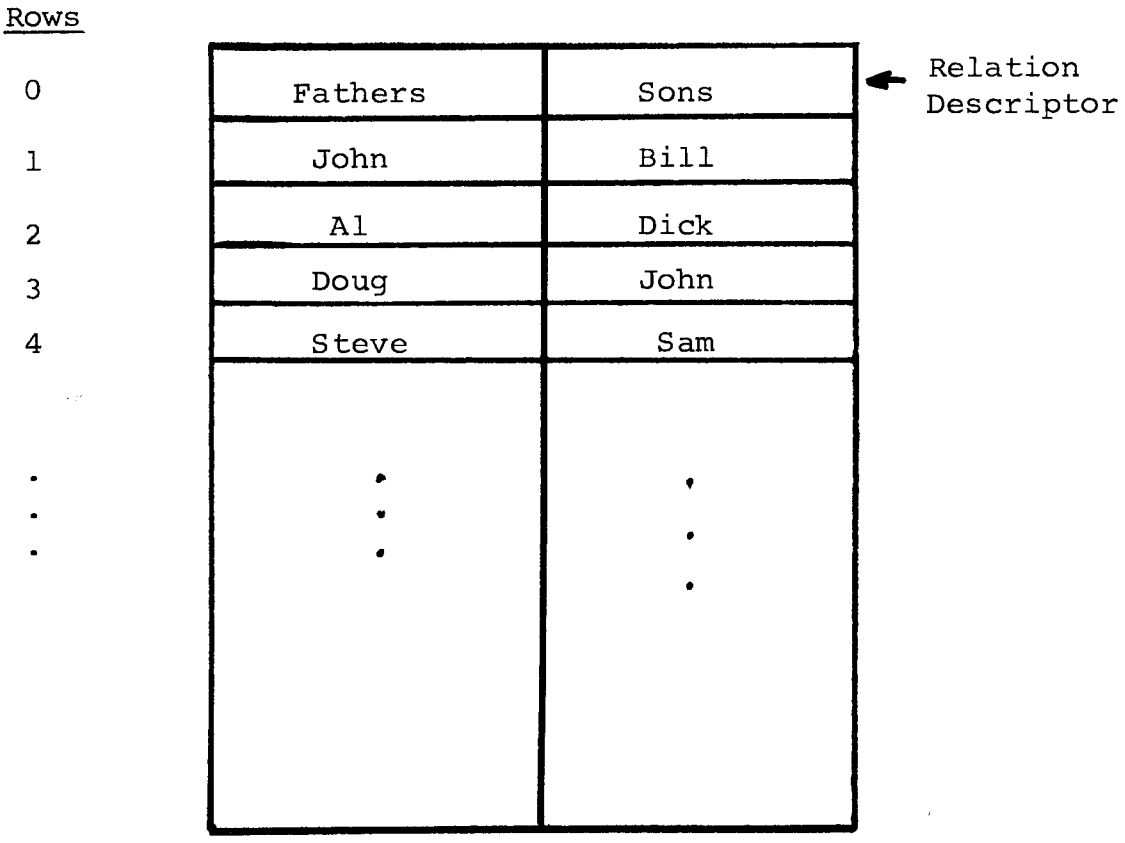
```
declare   1   node based,
              2 lp pointer,    /*pointer to the next left node*/
              2 rp pointer,    /*pointer to the next right node*/
              2 rn fixed bin, /*reference number*/
              2 dl fixed bin, /*data  length  in  number  of*/
                               /*characters*/
              2 data char(a_dl refer(node.dl)), /*actual data*/
              2 st bit(1);     /*status bit for logical deletion*/

declare    a_dl fixed bin;    /*maximum data length*/
```

Figure 4a.

Structure of the node for a integer.

```
declare  1    node based,
              2 lp pointer,    /*pointer to the next left node*/
              2 rp pointer,    /*pointer to the next right node*/
              2 in fixed bin, /*integer data*/
              2 st bit(1);     /*status bit for logical deletion*/
```

Figure 4b.



Notes: Both field names and instances of them are actually represented as reference numbers.

"John" appears both as a father and a son.

Figure 5 - The Canonical Form

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP None

3. REPORT TITLE

The MacAims Data Management System

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical Memorandum

5. AUTHOR(S) *(Last name, first name, initial)*

Goldstein, Robert C. and Strnad, Alois J.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April 1971	33	2

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
N00014-69-A-0276-0002	MAC TM-24
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

This paper describes the MacAIMS Data Management System (MADAM). It begins with a brief discussion of the overall goals of the project, its operating environment (Multics), and its data management requirements.

The MADAM system is based on a relational model of data, and employs set-theoretic primitive operations for manipulating data. The basic philosophy of the system and some issues involved in its implementation are described.

14. KEY WORDS

Data Management System	Data Manipulation	Management Information System
Information Retrieval	Access Control	Data Base Management
Data Structure	Data Organization	Set-Theoretic Data Structures

DD FORM 1 NOV 68 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

MIT/LCS/TM-24

THE MACAIMS DATA MANAGEMENT SYSTEM

Robert C. Goldstein
Alois J. Strnad

April 1971