THE RELATIONAL APPROACH TO THE MANAGEMENT OF DATA BASES

MAC Technical Memorandum 23

Alois J. Strnad

April 1971

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N00014-69-A-0276-0002.

PROJECT MAC
Massachusetts Institute of Technology

Cambridge Massachusetts 02139
THE RELATIONAL APPROACH TO THE MANAGEMENT OF DATA BASES

MAC Technical Memorandum 73

ALOIS L. SIERUJ

April 1971

RESEARCH INSTITUTE OF TECHNOLOGY
Massachusetts Institute of Technology
Cambridge
1. Introduction.

The ultimate goal of Project MacAIMS (MAC Advanced Interactive Management System) is to build a computer facility which will be able to support non-trivial decision making processes. (See reference 4). In the early stages of our experiments we discovered that traditional approaches to the management of data bases do not satisfy our needs. We have determined the following requirements for the management of Large Data Bases (LDB) in a dynamically varying environment such as an interactive Management Information System:

- high degree of flexibility
- data independence (i.e. programs are unaffected by changes of data representation or by addition of new data types)
- ability to operate on different data structures
- access path independent of data structure
- access control below the file level
- uniform retrieval time and (possibly) update time

The relational approach to management of LDB offers a solution which satisfies most of the above requirements.
This paper is concerned primarily with our implementation of this approach. The theoretical basis for the relational approach to management of LDB can be found in recently published papers (1,2).

We are gradually implementing our system on MULTICS (Multiplexed Information and Computing Service). MULTICS is a major achievement in the area of multi-programming, multi-processing and time-sharing computer systems. It is a prototype of the computer utility, where "utility" is used in the same sense as the telephone or power utility. It can run seven days a week, twenty four hours a day, and it is capable of meeting a wide range of service demands.

Two aspects of MULTICS are very important for our implementation -- memory organization and access control. In MULTICS, the traditional hierarchy of storage -- ranging from core through disks, tapes etc.-- appears to the user as a single level, two-dimensional memory organization. Each user process has available up to 256,000 identical memories called segments, and each segment may contain up to 64,000 36-bit words. The entire range of this space \((2^{34})\) is directly addressable.

The other important aspect of MULTICS for our purposes is the facility for access control. Again, a
two-dimensional approach has been taken. Each segment has associated with it an access control list which identifies all the users authorized to use that segment and which specifies the mode of access for each user. The second dimension of access control is provided by concentric protection rings. Any attempt to access a segment from an insufficiently privileged ring (i.e., an outer ring) will cause a trap, and a routine supplied by the owner of the segment will be called to decide whether or not to permit the access.

Let me also mention that our system is entirely programmed in PL/I, the primary language on MULTICS and the language in which MULTICS system itself is written.

II. General Information.

We take the view that information we might store in our data base consists of sets of Data Elements (DE) and sets of relations among them. The basic set theoretic primitive operations are used for manipulating the Relation Data Sets (RDS).

Given Data Element Sets \( \{S_1, S_2, \ldots, S_n\} \), corresponding Relation Data Sets consist of \( n \)-tuples (tuples
of degree n), each of which has its first element from set S1, its second element from S2, and so on. The Relation Descriptor (RD) is, in our terminology, the n-tuple composed of the names of the sets S1, S2, ....Sn. Suppose, for example, that there are Data Element Sets for persons' names, for addresses and for telephone numbers. We might construct an RDS which will represent the relations among members of these sets. The Relation Descriptor for this RDS will be the 3-tuple <person-name, address, telephone-number>. All other tuples will express the relation among the members of these sets. In our implementation, the relations are stored exclusively in terms of Reference Numbers.

The whole system is logically divided into two major parts. In the first, Data Element Sets are stored, Reference Numbers are assigned to the Data Elements, and operations are performed on DES's. In the second part of the system, Relation Data Sets are created and stored and basic set theoretic primitive operations are performed on them.

Reference Numbers (identification numbers) play an important role in our implementation. Whenever a new DE enters the system, it is immediately assigned a Reference Number. The RN is used for all subsequent operations on
that DE. The method used for storing and assigning RM's to DE's guarantees that the particular DE is stored only one time within the system. This fact is a significant contribution to our goal of storage and operational efficiency. In our particular implementation, RM's are all 36-bit quantities, which is the word length of the GE 645 computer on which MULTICS runs. Because the most frequent operations are performed on the Relation Data Sets and because all relations are expressed in terms of RM's, the operational efficiency of using the fixed length numbers is obvious. We also save storage space since the average length of a Data Element is almost always greater than one machine word.

Both parts of the system are described in detail in the following sections of this paper.

III. Data Element Sets.

In our system each category of data, such as name, address, telephone number, pointer, tuple, names of sets, etc. might be stored in one Data Element Set (DES). Note that tuples, pointers, etc. are treated in the same fashion as other categories. For example, persons' names can be
stored in one DES, and the name of this set is simply "person's name".

Moreover, the name of a set is just another data category. Therefore there is also one DES for names of Data Element Sets, with its own name being "set name". The names of all DES's are stored in this DES. One reason for this provision is that the system is self-descriptive -- i.e. all information about the data is stored within the system itself. Furthermore this descriptive data is treated exactly the same way as any other data. This is one significant advantage to our approach.

The total number of Data Element Sets depends on the particular application. For example, a Library Information System probably will have a DES for titles.

The reader might be somewhat confused at this point, but figure 1 and the rest of the paper will give a clearer explanation of the above concepts.

A Data Element Module (DEM) is a procedure which operates on a Data Element Set. Each DES may have its "own" DEM. We have defined the following functions of a DEM:

read - read the input stream and produce the Standard Form (SF). For example, the input to the DEM for dates might be "8, Febr., 1969". The SF returned by this
entry is "19690208". Data Elements can be stored in the DES in SF only.

write - write the given SF in natural, human-oriented form. Taking the Standard Form from the previous example, the output would be "February 8, 1969".

get reference number - the Reference Number corresponding to the given SF will be returned.

get data element - the SF corresponding to the given RN will be returned.

insert - the given SF will be inserted into the specified DES and RN assigned.

delete - the specified SF is deleted from the DES.

Corresponding functions of all DEM's except "read" and "write" perform the same processing. Therefore a common procedure called a Data Strategy Module (DSM) may be invoked as an intermediary between the DEM and the DES. However the mechanism for assigning the RN to the DE (actually to the SF) is different for each data type. Therefore there are different DSM's for each data type. The most frequently occurring data types are integers, character strings, and real numbers. For example, data categories such as salary or date have SF's which fall into the data type integer.

The Standard Forms of Data Elements are physically
stored in the DFS in the form of binary trees. The main
distinction between them is that for integers the SF and RN
are identical.

Figures 2 and 3 show the mechanism for storing the SF's
and assigning the RN's to them for both types.

As the Data Element Sets grow, the binary trees may
become unbalanced, resulting in less efficient operation of
the system. Programs which run in background mode handle
this situation by rebalancing the trees.

We have also implemented two "special" Reference
Numbers which are very useful. One is used to indicate a
null, or absent DE. The other one acts as a "wild card"--
that is, upon comparison with other Reference Numbers, it
will match any RN regardless of value. In the following
examples the null RN will be represented by "0" and the
"wild card" by "*".

The left part of figure 1 illustrates the interaction
between procedures and Data Element Sets. Notice that the
DEM's provide the direct interface between the outside world
and the part of the system described above.
IV. Relation Data Sets.

In section II the notion of Relation Data Set (RDS), Relation Descriptor (RD), relation, and tuple were introduced. In section III, the mechanism for keeping track of Data Elements (DE) and assigning Reference Numbers (RM) to them was presented.

The exact structure of the RDS's is irrelevant to the theory underlying the relational approach to data management. However, we believe that data (tuples in the case of RDS) should be stored using the representation most efficient for the particular application. Therefore the procedure, called a Relation Strategy Module (RSM), which operates on a particular RDS is designed for a particular data structure. Thus there is an RSM for lists, an RSM for trees, an RSM for arrays, etc. Note the difference between DEM's and RSM's: each DEM operates on Data Element Sets and is designed for a particular category of DE's; each RSM operates on Relational Data Sets, and is designed for a particular data structure.

The basic set primitive operations and several non-set theoretic operations are defined. These operations are performed by an RSM on a particular RDS. Before we describe
these operations in detail, we present examples of two RDS's. For the purposes of the following explanations, it is sufficient to represent RDS's as arrays, and the array elements as DE's. Keep in mind, however, that other representations are possible and that, in any case, the DE's are actually represented by RN's.

<table>
<thead>
<tr>
<th>RDS (A)</th>
<th>RDS (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>city</td>
</tr>
<tr>
<td>Wels</td>
<td>Cambr</td>
</tr>
<tr>
<td>Owens</td>
<td>Boston</td>
</tr>
<tr>
<td>Niles</td>
<td>Waban</td>
</tr>
<tr>
<td>Jones</td>
<td>Lynn</td>
</tr>
<tr>
<td>Strand</td>
<td>Lexin.</td>
</tr>
</tbody>
</table>

The above RDS's have the following properties:

1. Each row represents an n-tuple.
2. All rows are distinct.
3. The ordering of rows and columns is immaterial.

In our implementation, all of the set primitive operations require two RDS's as input and produce a third
RDS which is the result of the operation. The other operations do not necessarily follow this same scheme.

In order to have the capability of creating RDS's and filling them with data we have had to define two "non-set theoretic" operations:

**create set**

Input: RD  
Output: RDS

Example: RD = <name,city,phone>

Result:    
          RDS(C)

          name   city   phone

The result is an RDS with one tuple -- the RD.

**insert tuple**

Input: RDS,tuple  
Output: RDS

Example: RDS(C), tuple = <Golub,Boston,888542>

Result:    
          RDS(C)

          name   city   phone

          Golub   Boston  888542

The following operations are basic set primitives:
Intersection

Input: RDS, RDS
Output: RDS

Example: RDS(A), RDS(E)

Suppose we created RDS(E)

<table>
<thead>
<tr>
<th>name</th>
<th>city</th>
<th>prof.</th>
<th>sal.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result: RDS(F)

<table>
<thead>
<tr>
<th>name</th>
<th>city</th>
<th>prof.</th>
<th>sal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owens</td>
<td>Boston</td>
<td>prog</td>
<td>16000</td>
</tr>
<tr>
<td>Niles</td>
<td>Waban</td>
<td>prog</td>
<td>15000</td>
</tr>
</tbody>
</table>

Note the use of the "wild card".

difference

Input: RDS, RDS
Output: RDS

Example: RDS(A), RDS(F)

Result: RDS(G)

<table>
<thead>
<tr>
<th>name</th>
<th>city</th>
<th>prof.</th>
<th>sal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wels</td>
<td>Camb</td>
<td>eng.</td>
<td>18000</td>
</tr>
<tr>
<td>Jones</td>
<td>Lynn</td>
<td>dir.</td>
<td>24000</td>
</tr>
<tr>
<td>Strand</td>
<td>Lexin</td>
<td>clerk</td>
<td>12000</td>
</tr>
</tbody>
</table>
projection

Input: RDS, RDS

Output: RDS

Example: RDS(A), RDS(H)

Suppose we have RDS(H)

\[
\begin{array}{|c|c|}
\hline
\text{name} & \text{sal} \\
\hline
Wels & 18000 \\
Owens & 16000 \\
Niles & 15000 \\
Jones & 24000 \\
Strand & 12000 \\
\hline
\end{array}
\]

join

Input: RDS, RDS

Output: RDS

Example: RDS(A), RDS(B)

Result: RDS(J)

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{name} & \text{city} & \text{prof} & \text{sal} & \text{phone} & \text{age} \\
\hline
Jones & Lynn & dir. & 24000 & 6205 & 30 \\
Strand & Lexin & clerk & 12000 & 5857 & 34 \\
\hline
\end{array}
\]
**composition**

Input: RDS, RDS  
Output: RDS

Example: RDS(A), RDS(K)

Suppose, we have the RDS(K)

<table>
<thead>
<tr>
<th>sal.</th>
<th>seniority</th>
</tr>
</thead>
<tbody>
<tr>
<td>8500</td>
<td>2</td>
</tr>
<tr>
<td>32000</td>
<td>45</td>
</tr>
<tr>
<td>16000</td>
<td>10</td>
</tr>
<tr>
<td>12000</td>
<td>14</td>
</tr>
<tr>
<td>24000</td>
<td>16</td>
</tr>
</tbody>
</table>

Result: RDS(M)

<table>
<thead>
<tr>
<th>name</th>
<th>city</th>
<th>prof.</th>
<th>seniority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owens</td>
<td>Boston</td>
<td>prog</td>
<td>10</td>
</tr>
<tr>
<td>Jones</td>
<td>Lynn</td>
<td>dir.</td>
<td>16</td>
</tr>
<tr>
<td>Strand</td>
<td>Lexin</td>
<td>clerk</td>
<td>14</td>
</tr>
</tbody>
</table>

**cartesian product**

Input: RDS, RDS  
Output: RDS

Example: RDS(B), RDS(N)

Suppose we have the RDS(N)

<table>
<thead>
<tr>
<th>affiliate</th>
<th>building</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIT</td>
<td>0</td>
</tr>
<tr>
<td>MAC</td>
<td>E-58</td>
</tr>
</tbody>
</table>
Result: 

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>age</th>
<th>affil</th>
<th>bldg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>6205</td>
<td>30</td>
<td>MIT</td>
<td>0</td>
</tr>
<tr>
<td>Jones</td>
<td>6205</td>
<td>30</td>
<td>MAC</td>
<td>E-58</td>
</tr>
<tr>
<td>Smith</td>
<td>5432</td>
<td>20</td>
<td>MIT</td>
<td>0</td>
</tr>
<tr>
<td>Smith</td>
<td>5432</td>
<td>20</td>
<td>MAC</td>
<td>E-58</td>
</tr>
<tr>
<td>Strand</td>
<td>5857</td>
<td>34</td>
<td>MIT</td>
<td>0</td>
</tr>
<tr>
<td>Strand</td>
<td>5857</td>
<td>34</td>
<td>MAC</td>
<td>E-58</td>
</tr>
<tr>
<td>Wyly</td>
<td>6669</td>
<td>42</td>
<td>MIT</td>
<td>0</td>
</tr>
<tr>
<td>Wyly</td>
<td>6669</td>
<td>42</td>
<td>MAC</td>
<td>E-58</td>
</tr>
</tbody>
</table>

union

Input: RDS(RDS) Output: RDS

Example: RDS(B), RDS(P)

Suppose, we created RDS(P)

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sames</td>
<td>5555</td>
<td>52</td>
</tr>
<tr>
<td>Strand</td>
<td>5857</td>
<td>34</td>
</tr>
<tr>
<td>Buck</td>
<td>8765</td>
<td>64</td>
</tr>
</tbody>
</table>
Result:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>6205</td>
<td>30</td>
</tr>
<tr>
<td>Smith</td>
<td>5432</td>
<td>20</td>
</tr>
<tr>
<td>Strand</td>
<td>5857</td>
<td>34</td>
</tr>
<tr>
<td>Wyly</td>
<td>6669</td>
<td>42</td>
</tr>
<tr>
<td>Sames</td>
<td>5555</td>
<td>52</td>
</tr>
<tr>
<td>Buck</td>
<td>8765</td>
<td>64</td>
</tr>
</tbody>
</table>

In order to improve efficiency and to make the usage of the system simpler we have defined several redundant operations i.e. they can be performed by several calls of primitive operations.

sort

get successor

replace tuple

As has been previously mentioned, a Relation Strategy Module is designed to operate on a particular data structure. However, one of our requirements for management of data bases is to have the capability to combine and operate on data which have different representations. To accomplish this, each RSM has the capability to process not only the particular structure for which it is designed, but
also to operate on the canonical form of an RDS. A RSM is able to reproduce its "own" type of RDS in the canonical form, which in our current implementation is a list.

V. Final Remarks.

We believe that our implementation of a relational system will give a solution for the management of Large Data Bases which will be able to support our overall research objectives. For experimental purposes, we have developed a simple user interface, and we are just now starting to move data into the system. We expect that during a relatively short period of time we will know enough about the behavior and efficiency of the system that we will be able to make substantial improvements. We should be able to report the experimental results of this preliminary system in late February 1971.

VI. List of Abbreviations.

DE - Data Element
DES - Data Element Set
DEM - Data Element Module
DSM - Data Strategy Module
RDS - Relation Data Set
RD - Relation Descriptor
RDS - Relation Data Set
RSM - Relation Strategy Module
MULTICS - Multiplexed Information and Computing Service.

VII. References.


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
The Relational Approach to the Management of Data Bases

Technical Memorandum

Strnad, Alois J.

April 1971

N00014-69-A-0276-0002

MAC TM-23

Distribution of this document is unlimited.

Advanced Research Projects Agency
3D-200 Pentagon
Washington, D.C. 20301

This paper is concerned with the design and implementation of a relational system for management of Large Data Bases (LDB) at M.I.T., Project MAC.

We have determined the following six major requirements for the management of LDB in a dynamically varying environment, such as an Interactive Management System: high degree of flexibility; data independence; ability to operate on different data structures; access path independent of data structure; access control below the file level; uniform retrieval time.

We take the view that information we might store in our LDB consists of sets of data elements and sets of relations among data elements. The basic set theoretic operations are used for manipulating and operating upon these sets.

Data Management System
Information Retrieval
Data Structure
Data Manipulation
Access Control
Data Organization
Management Information System
Data Base Management
Set-Theoretic Data Structures
THE RELATIONAL APPROACH TO
THE MANAGEMENT OF DATA BASES

Alois J. Strnad

April 1971