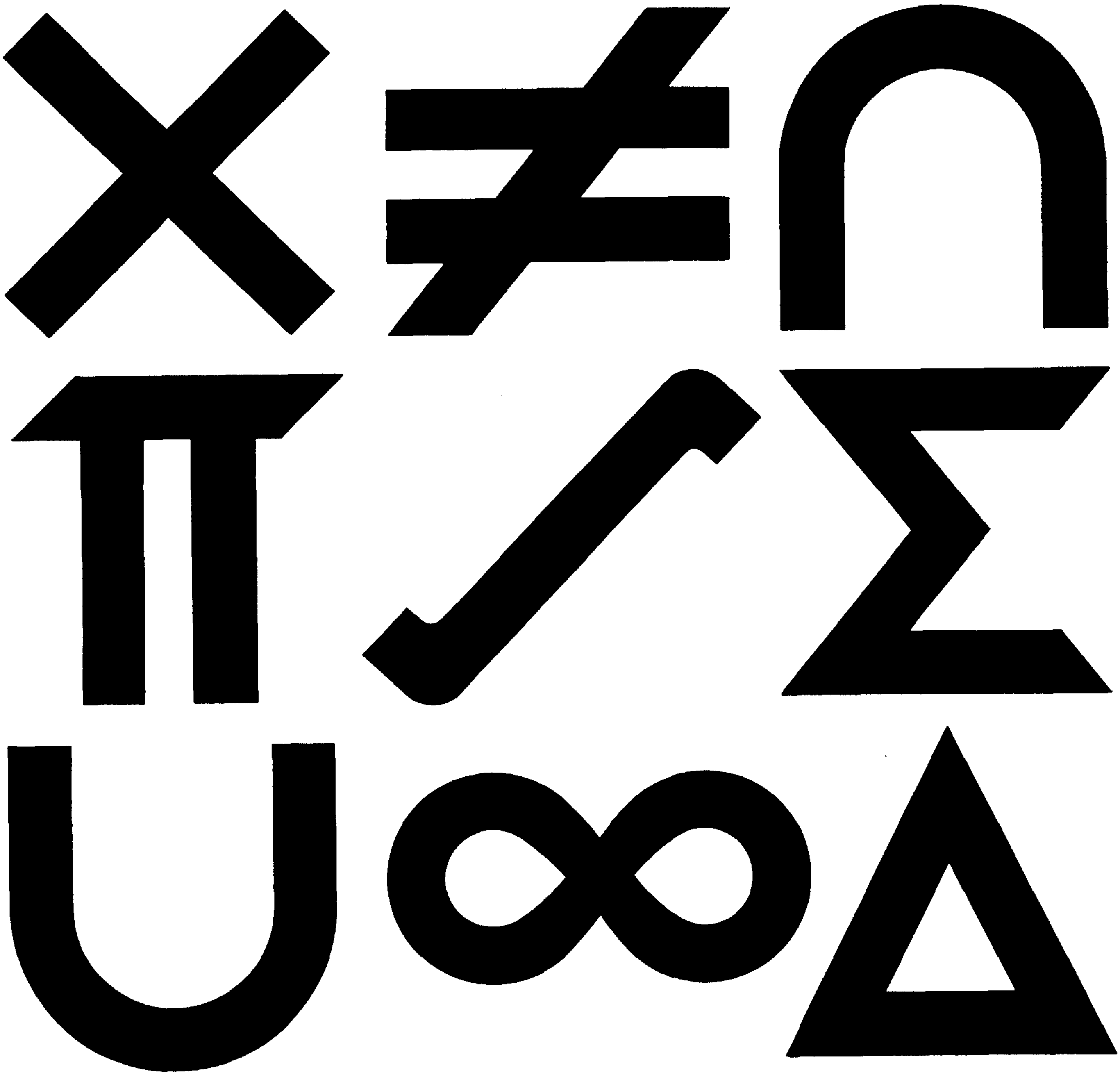
IBM

IBM CAMBRIDGE SCIENTIFIC CENTER
320-2096, January 1974

XRM
AN EXTENDED (N-ARY) RELATIONAL MEMORY

R. A. LORIE



IBM CAMBRIDGE SCIENTIFIC CENTER
TECHNICAL REPORT NO. 320-2096
JANUARY 1974

XRM
AN EXTENDED (N-ARY) RELATIONAL MEMORY

R. A. LORIE*

IBM CAMBRIDGE SCIENTIFIC CENTER
545 TECHNOLOGY SQUARE
CAMBRIDGE, MASSACHUSETTS 02139

*Present Address: IBM Research Laboratory, San Jose, California

ABSTRACT

The paper presents a low level interface for handling n-ary relations. An n-ary relation is a set of tuples of values. Values are encoded into integers. Operators are supplied to create and drop a relation, to add or delete tuples in a relation, to scan a relation, to retrieve a subset of a relation.

An implementation is described. It uses a binary relation processor as a base. Hashing and inversions are used to speed up the processing.

Some experiments are also described.

XRM

AN EXTENDED (N-ARY) RELATIONAL MEMORY

1. INTRODUCTION

Research in the area of data structures and data bases started in 1968 at the Cambridge Scientific Center as part of a project for graphics support. After an experimental implementation based on the J. Feldman and P. Rovner "triple" approach (6) a new system design was made, keeping some of the original concepts, but at the same time, introducing a more powerful relational structure (4)(5)(7)(8)(9).

The central design objective was to provide a low level data management system where the user could define entities (records) and relate them in an arbitrary number of ways. Many applications have a natural formulation in terms of networks and this justifies the emphasis put on the implementation of an efficient binary relation processor. This processor - called RM - has been used successfully in various applications since mid-1970.

The shift from the "triple" logical view towards a relational model was triggered by the work of E. F. Codd on the representation of data bases in terms of n-ary relations (2)(3). We also recognize that a large spectrum of applications require a relational model involving relations of degree higher than 2. We felt that the basic Relational Memory would provide an excellent tool for implementing n-ary relations. In mid-1972 we coded a prototype set of functions for creating and updating n-ary relations . The encouraging results stimulated a new project aimed at defining an n-ary relation interface.

At the same time D. Bjorner et al. (1) at IBM Research started a project with the same goal. Discussions with the group, and later on with J. Gray, also of IBM Research, lead us to the interface here presented.

The implementation of a prototype based on the Relational Memory will be presented. An application will be considered and some measurements discussed.

2. DESIGN OBJECTIVES OF XRM

The following objectives were set:

- XRM is a low level n-ary relation processor which can be used for implementing a relational algebra and/or calculus. It can also be used directly by a more sophisticated application programmer.
- Low level means that internal names (or identifiers) of data elements are visible to the user. It also means that parameters controlling the physical implementation may still appear in the interface. Inversions for speeding up retrieval are also visible.
- Relations are identified by internal names and domains are identified by their position index.
- For performance reasons some descriptive information needed for XRM for which the access path is known and constant should not necessarily be accessed in a relational way.
- The notion of unary and binary relation as known in RM must be preserved in order to take advantage of the high performance of the binary relation processor.
- The notion of ordering is not introduced at this level. We feel that a separate ordering mechanism provides much flexibility. The primitive operators would deal with maintaining sequences of identifiers independently of the data elements themselves. This allows multiple orderings for example. Such a facility is provided in RM but more work should be done in that area.

3. THE INTERFACE

3.1. The Data Model

One defines

3.1.1. Class-relations

A class-relation is a collection of data elements. A data element is a string of bytes. Every entry in a class-relation is given an identifier (id). An id is a fixed integer; it is the internal name of the data element. A class establishes a mapping between a string and its identifier.

Example: class-relation LOCATION

```
     id       data-element
           +------------------+
     (1)---'     NEW-YORK     '
           +------------------+

              +------------+
     (2)------'  BOSTON    '
              +------------+

              +------------+
     (3)------'  CHICAGO   '
              +------------+
```

There are several types of class-relations depending upon the properties of the mapping:

- One-to-one permanent mapping: to a particular data element corresponds one and only one id. Once a data-element has been assigned an id this assignment is permanent: if the data element is deleted, the id will not be reused; if it is recreated, it will be assigned the same id.

- One-to-one non permanent mapping: to a particular data element corresponds one and only one id. When a data element is deleted, the id is freed and returned to the common pool of id's. It can be assigned to any data element later defined.

- One-to-n mapping: When a data element is created, it is given a new id. No checking is made to see if such data element exists already or not.

Notes:

A one-to-one mapping is actually an encoding of a data element. A one-to-n mapping is not an encoding of the data element but provides an addressing scheme. Consider a piece of text; every data element is a sentence of the text. By chance two lines have the same contents but they are logically two different entities and should be referenced by two different id's. They could later be updated independently.

As id's will be used to maintain relationships among data elements, the difference between permanent or non-permanent mapping is significant. When a data element is deleted in a permanent mapping all references to its id should be deleted from the data base. However, if such references are kept, an error will be signaled when one tries to decode an id corresponding to a deleted data element. When a data element is deleted from a non-permanent mapping all references to its id should be carefully removed from the data base. Otherwise, the id will be reused for a different data element and false relationships will exist.

3.1.2. Regular Relations

An n-ary relation or relation of degree n is a set of n-tuples. Each n-tuple (or tuple) is a vector of n integers (actual data element values or identifiers or "undefined"). The ith value of the tuple corresponds to the ith domain of the relation. The primary key of a relation is a subset of domains such that the values corresponding to these domains uniquely identify the tuple. A relation is identified by a relation identifier (rid).

There are several types of relations depending upon the fact that identifiers are associated with each tuple in the relation (relation with tuple-identifiers) or not (relation without tuple-identifiers). Relations without tuple-identifiers are defined only for unary (degree 1) or binary (degree 2) relations. A tuple identifier will be called a tid.

We need at this point to discuss the introduction of tuple-identifiers and why we introduce a special case for unary and binary relations although they can be thought of as n-ary relations with n=1 or 2. The justification for such a particular case lies in the relative simplicity of unary or binary relation processing with the present technology. A relation can be represented as a sorted list of tuples. A search operation can take advantage of the ordering when the value of the sorting key is specified. To be able to search efficiently on any subset of the domains, one needs to store multiple permutations of the domains. The price in storage becomes rapidly prohibitive when the degree increases and another technique must be used.

We adopt the following representation: a tuple is stored as

```
tid e1 e2 e3 e4 ... en
```

where tid is the tuple-identifier. With the jth domain one can associate an *inversion* binary relation rj. For each tuple the pairs

```
e1 tid
e2 tid
e3 tid
...
en tid
```

are added respectively to r1, r2, ... rn. We use the first representation for unary and binary relations. The second representation is used for $n\underline{!}3$.

This justifies the use of tid's for building inversion. In fact it only illustrates the use of tid's as synonyms of the primary keys, much easier to manipulate. Note also that in some cases (one-to-n mapping) the tid is the only way of uniquely identifying the data element.

Two remarks for relations with tuple-identifiers:

- The mapping is always a one-to-one mapping but can be permanent or non-permanent as for class-relations.
- The primary key may be specified to be *external* to the tuple. In this case the relation is equivalent to a regular relation and a class relation. To each tuple in the relation corresponds one and only one data element of the class relation and they are both identified by the same id. Depending upon the function the relation is seen as regular or class relation (Figure 1).

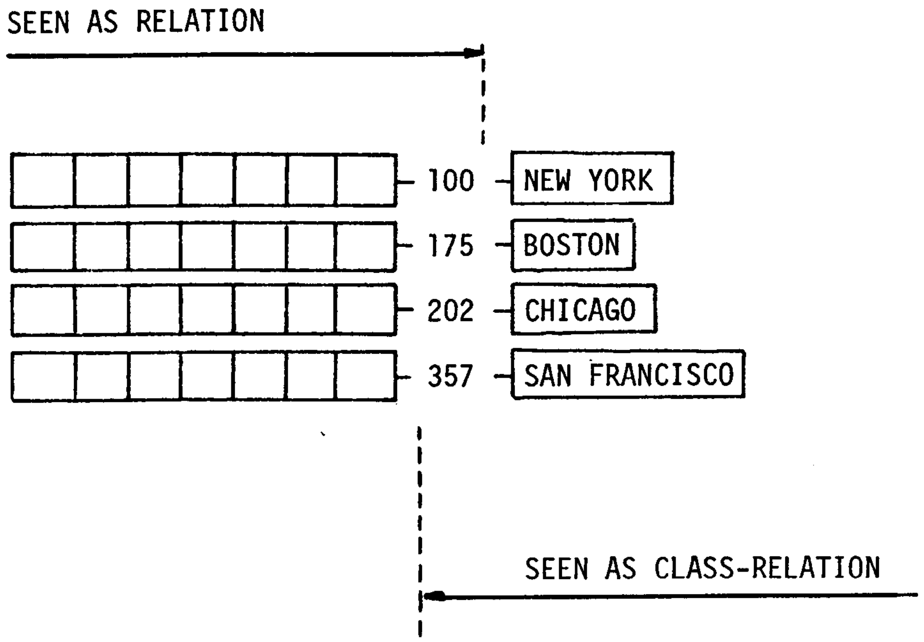


Figure 1.

3.1.3 Special Relations

Two special types of relations are defined. They are binary by nature and do not have tuple-identifiers. They are:

- the hash relations
- the inversion relations

Although they are used internally and cannot be explicitly updated by the user, they are made available for retrieval. Inversion relations have been defined above.

A hash relation contains pairs of the form (h,i) where i is the identifier of a data element (or tuple) and h a hash value obtained by applying a hash function to the data-element (or the key in the tuple).

3.1.4. The Master Relation

The master relation is unique and is used for storing information about all relations defined in the memory. A relation before being used must be defined by adding a tuple to the master relation. The master relation is predefined in the system. It is referred to by a unique master relation identifier. Note that any rid is also a tid of a tuple in the master relation.

The domains in the master relation specify for each relation:

- the type of relation
- the degree
- the primary key
- some control information
- some user's information.

3.1.5. Summary

The types of relations can be grouped and numbered in a tree-like form as in Figure 2.

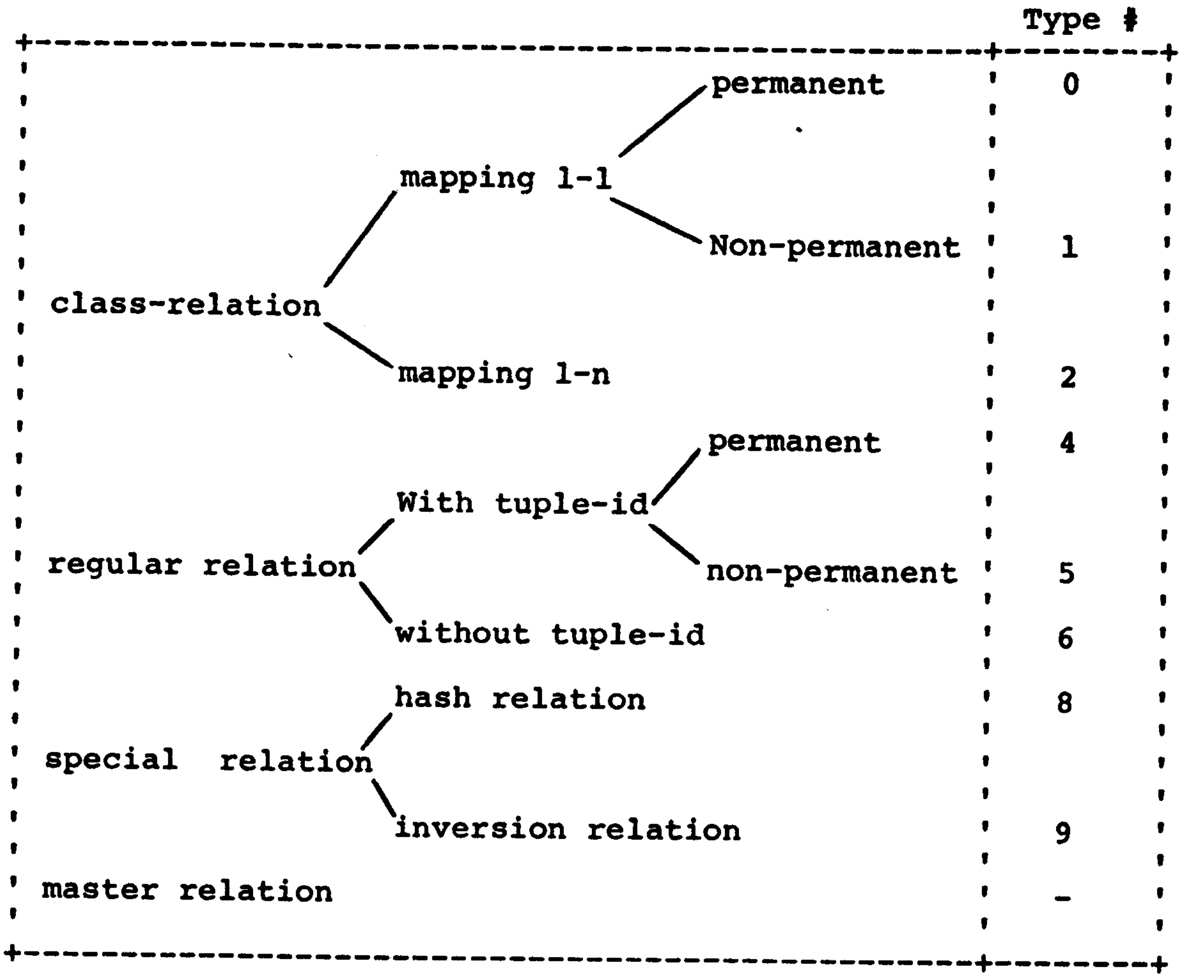


Figure 2

Note the logical similarity between class relation and regular relation with tuple id. In the next section the terms relation and tid will also be used to denote class relation and data element identifier.

3.2. Functions

The functions operating on the data model may be divided in three groups.

- Functions which deal with the existence of a relation in the memory:

  - DEFINE defines a new relation by making an entry in the master relation.

  - DROP deletes the whole relation from the memory and suppresses the entry in the master relation.

  The master relation cannot be defined or dropped.

- Functions which deal with one single entry of a relation:

  - ADD and DELETE makes or suppresses an entry in a relation (and creates or frees its tid when needed); these functions are invalid for the master relation and special relations.

  - FETCH returns a tuple when its tid is given.

  - TID returns the tid when the key is given.

  - UPDATE allows modification of some domains of a tuple; the key cannot be altered.

- Functions which deal with the relation as a whole:

  - OPEN/NEXT/CLOSE allow successive retrieval of every tid (and the tuple if wanted) in the relation.

  - NUMBER returns the number of entries in a relation.

  - EMPTY deletes all entries in a relation.

  - INVERT associates one or several inversion relation(s) - previously defined - with one or several domain(s) of a relation. The inversion relation is said to be active for that domain. If the relation is not empty the inversion is automatically updated; an entry is made for each tuple in the relation. When subsequent tuples are made in or deleted from the relation corresponding entries are made or deleted automatically in the inversion relation.

- RETRIEVE performs a search for all tuples in a relation which have specified values for some domains. For relations with tuple-identifiers the result is a set of tid's (unary relation without tid). This operation does not apply to class-relations. Note that TID should be used when the values of all domains constituting the primary key are specified, as it uses the hashing mechanism and is therefore faster.

  For relations without tuple-identifiers the value of the unspecified domain of the binary relation is added to the set instead of the (unexisting) tid. The operation is meaningless for unary relation.

  RETRIEVE requires that inversions exist on all domains for which a value is specified.

4. IMPLEMENTATION

The implementation of the n-ary relation processor is entirely based on the binary relation processor (RM). It uses from RM the capabilities of defining entities and maintaining unary and binary relations of the form R(a) and R(a,b). The only retrieval capability which is used is: find all entries in R or find all b's associated with a given a in a relation R. The reader who is not familiar with RM should refer to the appendix, where those functions are briefly explained or to the papers cited in the bibliography.

As there are also id's and rid's in RM we will always qualify them by the prefix RM when they could be confused with an id or rid at the XRM level.

4.1 The master relation.

The master relation is used to store the logical characteristics of the relation defined in the system like the type, the degree, the primary key but also some control information used by the implementation. For ease of reference we number those as follows:

d1 tentative RM entity-id
d2 tentative RM relation-id
d3 last RM entity-id used for the relation
d4 main RM relation-id
d5 alternate RM relation-id
d6 Block RM relation-id
d7 encoding control tuple id
d8 inversion control tuple id
d9 retrieval control tuple id

Their usage is explained in the following sections.

4.2. Representation of a Class-relation, 1-n Mapping

When the class-relation is defined a RM unary relation C is created and its RM-rid stored in the master relation (d4). The clustering mechanism of RM is used by specifying a tentative id which is also stored in the master relation (d2).

Every creation of a data element in that class implies

- The creation of an entity in RM, where the data element is stored; its RM-id becomes the id of the data element. The clustering mechanism of RM can be used by specifying a tentative id when a data element is created. An automatic clustering can be used in which one simple tentative id is provided by the user when the relation is defined and the last id allocated to a data element of that class is used as tentative id for the creation of the next element. The single tentative id is stored in the master relation (d1); so is the last id (d3).

- The addition of this id to the relation C.

4.3 Representation of a Class-relation, 1-1 Mapping, Non Permanent

When the class-relation is defined a RM binary relation is created and its RM-rid (call it C) is stored in the master relation as in 4.2.

Every creation of a data element in that class implies

- A test for existence. If negative:
- The creation of an entity in RM, where the data element is stored; its RM-id becomes the id of the data element.
- The addition of the pair (h,d) to C, where h is the value of a hashing function applied to the data element.

The test for existence proceeds as follows: The data element is hashed to find the value h. The RM retrieval mechanism is used to find all id's associated with h in C. The entities corresponding to these id's are fetched to see which one, if any, contains actually the data element.

Note that the hash value does not correspond to any physical slot; its range is very large as it is limited only by the size of an integer in the particular implementation. Therefore the probability of "multiple hit" (conflict) is very small.

4.4 Representation of a Permanent Mapping

The representation of the class is identical to the one used for a non-permanent mapping. However, a second RM binary relation is created when the class relation is defined and stored (alternate RM relation C') in the master relation (d5).

When a data element is deleted from the class the RM entity is not erased but only flagged; the entry (h,d) is deleted from C and added to C'.

When a data element is added to the class the normal existence test is made by using C. If negative another existence test must be made by using C'. If positive the entity already exists, the flag is removed and the pair (h,d) transferred from C' to C.

4.5. Representation of a Relation Without Tuple Identifiers

When a relation without tuple identifiers is defined, a RM binary relation of identical degree (1 or 2) is created and its RM-id stored in the master relation (d4). The RM relation is then used to store the contents of the relation.

4.6. Representation of a Relation With Tuple-Identifiers

When the relation is defined a RM binary relation is created and its RM-id (call it C) is stored in the master relation as in 4.2.

The addition of a tuple (e1, e2 ...en) to the relation rid implies:

- The creation of an entity (suppose its id is k) in which the tuple is stored. Clustering applies as in 4.2.
- The addition of the pair (h,k) to C, where h is the value of a hashing function applied to the values of the domains which constitute the primary key of the relation.

  The procedure becomes identical to the one described for a class relation in 4.3. The permanent mapping can be implemented exactly as in 4.4.

4.7. Inversions

The creation of a tuple in a relation as explained in 4.6 assumes there are no inversions active at that moment for the relation rid.

We have mentioned in the interface description that an inversion relation must be defined explicitly and then associated with a particular domain of a particular relation. Let us identify the relation by its rid and the inversion relation corresponding to its kth domain by rik.

The associations between domains and inversion relations are kept in an <u>inversion control tuple</u>.

The tid of the control tuple is kept in the master relation (d8) in the tuple corresponding to the relation rid. The kth item in the control tuple contains the identifier rik. The structure is displayed in Figure 3.

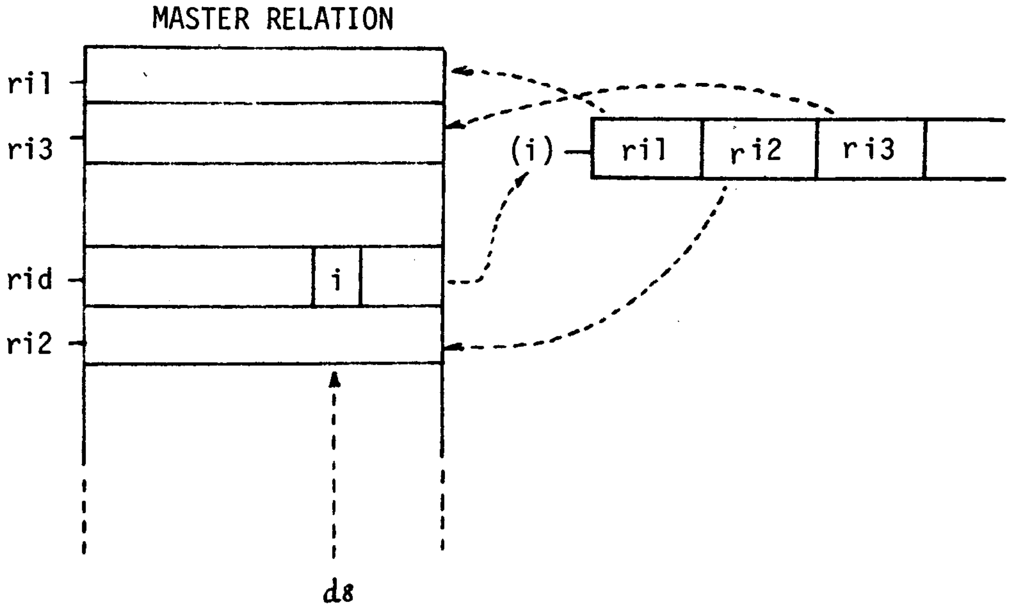


Figure 3

When the inversion relation rik is defined it is associated a RM binary relation identified by its RM-id. This id is stored in the master relation (d4) in the tuple relative to rik.

Anytime an entry (e1, e2, ... ek ... en) is made in the relation identified by rid and is given the tid x for example, a pair (ek,x) is added to the RM binary relation associated with rk. The converse process is done when an entry is deleted from rid.

4.8 Block Relation

The operation of retrieving all the tuples in a relation is very common not only at the user's level but also at the system level (see 4.9). It is also convenient to obtain the identifiers of such tuples in some predetermined order. For performance reasons this order should follow the physical relative position of the tuples so that every block is fetched only once during a scanning of the relation. This is achieved by associating another RM unary relation (called a block-relation) with every relation. Its RM-rid is stored in the master relation (d5). The unary block-relation contains a block identifier for each block which contains at least one tuple of the relation. We choose as block identifier the lowest possible RM-id in the block.

4.9. Inversions Revisited

When an inversion relation is associated with the kth domain of a relation rid by an INVERT command it must be updated for each tuple already existing in the relation. This implies the scanning of all tuples already in the relation. The scanning uses the block relation to find each block and then scan all entities in the block to find the ones which correspond to tuples in the relation. The pairs (value of kth domain, tid) are stored in a working buffer. When the buffer is full the pairs are sorted on the values and entered in the inversion relation. Several buffers can be used at once, corresponding to several domains to be inverted in order to reduce the amount of scanning.

4.10. Retrieval

When a relation is defined a retrieval control tuple is created and its tid stored in the master relation (d9). The control tuple can be updated by the user to specify an ordering of the domains in decreasing order of selectivity.

Let us suppose that the retrieval command specifies the values of the domains j, k, l and that the decreasing order of selectivity is k, l, j. The inversion relation rik is used to find all tid's of tuples which satisfy the constraint on domain k. These id's are stored in a buffer until all have been stored or until buffer is full. The inversion relation ril is then used to test if tid's in the buffer also satisfy the constraint on domain l. The rij is used for a similar process. The tid's which satisfy the specified constraints are added to the answer set. If all entries in rik have not been processed the procedure iterates.

5. XRM APPLICATION

In order to debug the XRM package with a non-trivial data base and get some performance estimates we generated a data base automatically.

The data base describes a company organization. The main relation is an 8-ary relation EMPLOYEE. The other relations are essentially classes. Their characteristics are given in tables 1 and 2. Type refers to the numbers in Figure 2. The abbreviation Ext. refers to external key as explained in 3.1.2.

Relation Names	Description	Type	Degree	Primary Key	Domains	Data-item for Classes
DEPC	Department code	4	1	Ext	tid in DEPD	Dept-code
DEPD	Department description	4	1	Ext	tid in DEPC	Dept description
JOBC	Job code	4	1	Ext	tid in JOBD	Job-code
JOBD	Job description	4	1	Ext	tid in JOBC	Job description
LOC	Location	5	1	Ext	area code	Location
FNAME	first name	1	-	-	-	first name
NAME	name	1	-	-	-	name
EMPLOYEE	employee	5	8	First Domain	See Table 2	-

TABLE 1

The EMPLOYEE relation:

domain 1	employee serial number
domain 2	tid in NAME
domain 3	tid in FNAME
domain 4	tid in LOC
domain 5	salary
domain 6	tid in JOBC
domain 7	tid in DEPC
domain 8	tid of manager (in EMPLOYEE)

TABLE 2

The lengths of the data items are (in characters):

dept code	3	± 0
job code	4	± 0
dept description	32	± 16
job description	18	± 10
location	8	± 4
first name	7	± 3
name	10	± 6

The structure of the data base consists of:

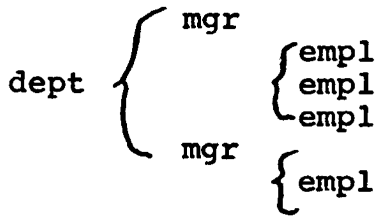
nd	departments
nj	jobs
nl	locations
nf	first names
nn	names
n	employees

There is one second level manager. There are nm first level managers in each department and ne employees reporting to each manager.

Jobs are assigned randomly. Salaries are assigned randomly although a constant is added to the salaries of managers. All employees working in a given department are working at the same location except the employees reporting to one of the managers of the department. These employees are working at locations chosen randomly.

Clustering is organized as follows:

- The pairs department code - department description are clustered together
- The same is true for jobs
- Locations are clustered together
- First names are clustered together
- All employees in a same department are clustered together in a tree-like manner.



- A name is clustered together with the first employee with such a name.

The program consists of a set of transactions. Each of these transactions is coded in PL/I. There is, therefore, no mapping from a language into XRM.

We shall describe the sequence of XRM calls for each transaction and record the time and number of disk accesses required by each transaction for four or five different data base sizes (s1,s2,...).

These sizes are defined in terms of the parameters described above.

Data Base Size	nd	nj	nl	nf	nm	ne	nn	n
S1	4	4	4	4	3	2	36	37
S2	30	20	10	40	5	3	557	601
S3	50	60	15	70	5	5	1254	1501
S4	98	200	19	93	5	7	2542	3921
S5	117	200	19	97	5	8	2932	5266

Notes

- The system uses a pool of ten pages of 4k in core.
- The first two buffers are always occupied by the same pages (containing the master relation).
- The fact that the buffers are used for write and read operations explains why a buffer can be written back during a query involving reads only.
- The number of read and write operations is not significant when it is very small. This is due to the buffering mechanism.
- The times recorded for the transactions do not include the destruction of work relations as this operation should be almost immediate in a multiple segments environment.
- The application program is written in PL/I
- The program is run in CP-CMS on a 360/67. Times are given in seconds (virtual time) if not otherwise specified.

T1: create departments

- Create entry in DEPC
- Create entry in DEPD
- Update entry in DEPC with tid of entry in DEPD

time: 31 to 33 ms per department

T2: create jobs

Same as T1 but using relations JOBC and JOBD

time: 30 to 34 ms per job

T3: create locations

- Create entry in LOC

time: 14 to 15 ms per location

T4: create first names

- Create entry in FNAME

  time: 11 to 13 ms per first name

As there are only a small number of entries involved in these transactions the number of disk accesses is very small and no meaningful results may be derived from them.

When created the tid's of entries in DEPC, JOBC, LOC and FNAME are kept in a table in core. T5 draws tid's randomly from this table.

T5: create employees

- triple DO loop on departments, managers in the department, employees reporting to manager.
- For each employee:
  - Generate name
  - Find tid of name or create entry for such name in NAME
  - Create entry in EMPLOYEE

Measurements:

	n	CPU Time	R	W
S1	37	0.8	8	11
S2	601	20	72	277
S3	1501	52	1904	577
S4	3921	142	6368	3759
S5	5266	186	10069	5569

time: propotional to n (as soon as the size is not trivial); constant per employee = 35 ms

R represents the number of disk accesses in read mode.

W represents the number of disk accesses in write mode.

R and W increase more than linearly because their are more employees with the same name in the largest data bases.

T6: generate inversions on all domains of the relation EMPLOYEE

- Generate inversions on domains 2, 3, 4
- Generate inversions on domains 5, 6
- Generate inversions on domains 7,8

Measurements:

	n	CPU Time	R	W
S1	37	0.9	8	16
S2	601	18	48	28
S3	1501	56	149	94
S4	3921	154	497	346
S5	5266	213	732	538

Rough estimate: 5.5 ms per entry and per inversion.

T7: Find the salary of the manager of employee whose serial number is X.

- Find the tid of entry in EMPLOYEE for which key is X
- Fetch tuple
- Get manager's id, fetch tuple and get salary

Measurements:

	CPU Time	R	W
S1	0.20	0	0
S2	0.14	2	0
S3	0.14	2	0
S4	0.16	3	0
S5	0.16	3	0

T8: Find the names of employees in dept X

- Encode dept x using DEPC
- Associative retrieval on domain 7 (tid in DEPC) of relation EMPLOYEE
- For each tid in answer set fetch tuple and find tid of name - then decode.

Measurements:

	CPU Time	R	W	K
S1	.83	0	0	10
S2	.17	3	0	21
S3	0.25	4	0	31
S4	0.31	18	0	40
S5	0.34	3	0	46

K is the number of names which satisfy the query. The time increases only with K. Disk accesses occur mainly during the decoding of names in the answer. Remember that names are clustered together with the first occurrence of an employee with such a name. For S1, S2, S3, S5 we specified one of the first departments created. For employees in such departments the names have a very high probability of being adjacent to the employee tuples. For S4 we specified a department created later and many employees in such a department have names that have been previously defined and cannot therefore be physically adjacent to the employee tuples.

T9: Find the locations where at least one employee of dept X is working

- Encode dept x using DEPC
- Associative retrieval on domain 7 (departments) of relation EMPLOYEE
- For each tid in answer set fetch tuple, get value of domain 4 (location) and put value in a working set.
- Decode each tid in working set using relation LOCATION.

Measurements:

	CPU TIME	R	W	K
S1	0.13	1	1	3
S2	0.22	1	1	4
S3	0.24	1	1	6
S4	0.35	4	1	6
S5	0.41	1	1	8

K is the number of items in the answer.

The time increases less than linearly with the number of employees per department (nm+nm*ne) because K does not increase linearly with the number of departments.

<u>T10</u>: Find the number of employees with job x in location y.

- Encode x
- Encode y
- Associative retrieval on domains 4 and 6 in relation EMPLOYEE
- Get cardinality of answer set

<u>Measurements</u>:

	CPU TIME	R	W	K
S1	0.11	0	0	5
S2	0.21	2	0	3
S3	0.20	3	0	0
S4	0.5	6	0	3
S5	1.3	7	1	6

This example illustrates the performance of the associative retrieval. K is the answer.

T11: Find the jobs existing in all locations

- Read successively all entries in the relation EMPLOYEE. For each of these entries get the value of the 4th domain (location) and the value of the 6th domain (job) and add to a work binary relation r1 the pair

  (tid JOB, tid LOC)

- Find K the number of locations
- By reading and counting the entries in r1 find jobs for which there are k entries.

Measurements:

	CPU Time	R	W
S1	0.3	0	2
S2	4.5	14	5
S3	31.5	29	6
S4	31.6	137	70
S5	41.3	285	182

The increase in time is proportional to the number of employees.

T12: Find the names of employees who make more than their managers.

- Consider the inversion relation on the domain 8 of employee. Read successively each entry of the form.

  (manager's tid, employee's tid)

- For the first occurrence of a manager tid fetch tuple and get salary.
- For each employer's tid fetch tuple and salary.
- Compare - If tid is satisfied get tid of name and decode

Measurements:

	CPU Time	R	W	K
S1	0.2	0	0	1
S2	3.0	26	1	19
S3	7.2	59	4	71
S4	16.0	123	5	140
S5	21.6	187	4	189

Time should be mainly proportional to the number of employees but is influenced by the number of employees in the answer (k).

T13: List the whole relation EMPLOYEE

- Read successively all entries in EMPLOYEE
- Decode tid's using NAME, FNAME, LOC, JOBC and DEPC.

Measurements:

	CPU Time	R	W
S1	1.4	0	1
S2	15	22	0
S3	37	274	0
S4	103	3618	0
S5	139	5956	0

Time is proportional to the number of employees.

Constant per employees = 29 ms.

When the data base becomes large the number of disk accesses increases. This is due to the decoding of data elements (mainly the names) and is the price to be paid for avoiding redundancy of information in the application. Note, however, that XRM can be used with a different trade-off in mind. For example, when a second occurrence of a name is encountered a copy can be made, adjacent to the employee tuple, and its tid stored in the tuple (2d domain). A binary relation associates the tid of the copy with the tid of the first occurrence which is the only one to appear in the class NAME.

Space Requirements Analysis:

Let us discuss the space requirements in the case of S4.

The master relation and entries in classes DEPC, DEPD, JOBC, JOBD, LOC and FNAME have been defined on 8 pages (1 page = 4K bytes). But 40% of the capacity is still free.

The entries in the relations NAME and EMPLOYEE occupy completely 61 pages. A sequential file with fields accommodating the maximum length of data elements and no extra information would require 68 pages. So the overhead for handling variable entity sizes and providing permanent identifiers is well offset.

All binary relations used for implementing the n-ary relations, classes and all inversions start on 4 relation pages and use 88 overflow pages. They contain 37000 entries. The overflow pages are used at 57% of their capacity.

6. CONCLUSION:

The following conclusions can be drawn from the experiment:

- The time to create an entry in a relation is practically independent of the size of the data base.
- This is also true for transactions using the inversion relations directly like T7, T8, T9, T10.
- The advantage of using id's internally is illustrated by many queries. Data element appearing in the query are encoded and tid's which satisfy the query are decoded. The processing itself is done entirely by using tid's.
- The time should not be taken too strictly. The 12 ms required to create an entry in a class can be divided as follows:
  - 5% application program
  - 45% processing in XRM
  - 50% processing in RM

XRM is written in a higher level language and requires some optimization. Note also that the interface requires the master relation to be interrogated at each call to find the characteristics of the relations. A slightly modified interface could include a "batch mode" option where these characteristics could be kept in a work area. All subsequent calls involving the same relation could refer to the area.

But most of the savings can be achieved in RM. RM as it stands now has an interface oriented towards the direct use of RM by an application programmer. XRM only uses a subset of that interface. By stripping RM of unnecessary features performance could be improved.

We are also designing a new representation of binary relations in RM. This design would use a vector representation instead of rings. The maximum size of a ring in RM allows for roughly 500 entries. In a vector representation binary search could be used to locate an entry (very frequent operation). All together we expect such a new implementation to provide a drastic improvement in performance. We hope to proceed with such enhancements and publish new results in a following paper.

REFERENCES

(1) Bjorner, D., E.F. Codd, K.L. Deckert, I.L. Traiger, The Gamma-Zero n-ary Relational Data Base interface, IBM Research, Report No. RJ-1200, April 1973.

(2) Codd, E.F., A relational model of data for large shared data banks, Communications of the ACM, Vol. 13, No 6, June 1970.

(3) ---, Normalized data base structure: a brief tutorial, IBM Research, Report No RJ 935, November 1971.

(4) Crick, M.F.C., Lorie, R.A., Mosher, E.J., Symonds, A.J. A data-base system for interactive applications, CSC(*), Report No G320-2058, July 1970.

(5) Crick, M.F.C., Symonds, A.J., A software Associative Memory for complex data structures, CSC Report No G320-2060, August 1970.

(6) Feldman, J.A., Rovner, P.D., An Algol-based associative language, Communications of the ACM, Vol. 12, No 8, August 1969.

(7) Lorie, R.A., Symonds, A.J., Use of a relational access method under APL, CSC Report No G320-2071, May 1971.

(8) ---, A schema for describing a relational data-base, CSC Report No G320-2059, July 1971.

(9) User's guide for the Relational Memory, CSC, July 1972.

* IBM Cambridge Scientific Center, Cambridge (Mass.)

APPENDIX

RM - Binary Relational Memory

1. Logical data model

1.1. Entities

An entity is a record of arbitrary length, identified by an entity-identifier (id). The record is a string of bytes; the system is unaware of its content. An id is a positive integer. Functions are provided to create, delete and modify entities.

1.2. Relations

Relations can be of different types. The most common one is the directed binary relation. It is a set of ordered pairs ei - ej (also called entries) where ei and ej are identifiers (or positive integers). An empty relation must always be created before any pair can be added to it. It is identified by a relation identifier (rid). Functions are provided

- to create or release a relation,
- to add or suppress an entry in a relation,
- to retrieve successively all pairs ei-ej, all ei's or all ej's associated with a given ei.

2. Physical implementation

Entities and relations are stored in two different <u>spaces</u>. Both entity and relation spaces in the data base consist of a series of blocks of equal size (4k bytes), numbered from zero. During processing, a pool of buffers of the same size as a page is maintained in core storage. Blocks are brought in and rolled out when needed. This operation is transparent to the user.

2.1. The entity space

Direct addressing is used: an entity with id I is found on page p. The number p is the largest integer such that $p \leq I/k$ where k is a system parameter. Inside a page, pointers are used to accommodate variable length records. The user can monitor the clustering of entities when he creates them. An overflow procedure exists.

2.2. The relation space

A similar direct addressing scheme is used to locate the beginning of a relation. Inside a page, pointers are used to link together (in lexicographic order) the entries in a same relation (see Fig. 4). When an overflow occurs an index of overflow pages is kept on the original page to provide quick access to any entry.

relation

rid e_1 - e_4

e_2 - e_5

e_2 - e_6

e_2 - e_7

e_3 - e_8

$e_1 < e_2 < e_3$

$e_5 < e_6 < e_7$

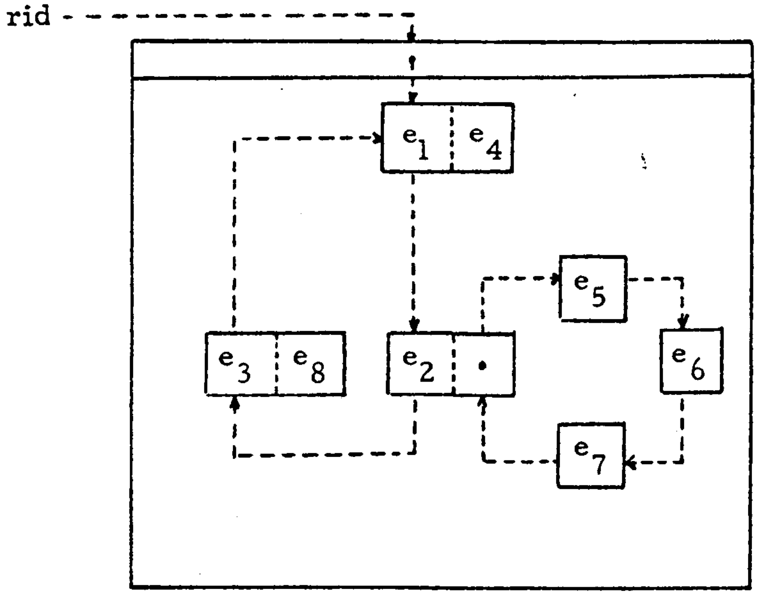


Figure 4

TECHNICAL REPORT INDEXING INFORMATION

1. AUTHOR(S):
R. A. Lorie

2. TITLE:
XRM - An Extended (n-ary)
Relational Memory

3. ORIGINATING DEPARTMENT:
Cambridge Scientific Center

4. REPORT NUMBER:
G320-2096

5a. NO. OF PAGES
34

5b. NO. OF REFERENCES
9

9. INDEX TERMS FOR THE IBM SUBJECT INDEX:

Data-Base
Data-Bank
Relational Model
Information Retrieval
Binary Relation
N-ary Relation
Files
Relation

05 - Computer Application
21 - Programming

6a. DATE COMPLETED
December 14, 1973

6b. DATE OF INITIAL PRINTING
January 1974

6c. DATE OF LAST PRINTING
November 1975

7. ABSTRACT:

The paper presents a low level interface for handling n-ary relations. An n-ary relation is a set of tuples of values. Values are encoded into integers. Operators are supplied to create and drop a relation, to add or delete tuples in a relation, to scan a relation, to retrieve a subset of a relation.

An implementation is described. It uses a binary relation processor as a base. Hashing and inversions are used to speed up the processing.

Some experiments are also described

8. REMARKS: None

IBM