A recent approach to representing relations between entities in a graphics data structure has been to store information as triples in the form Attribute (Object) = Value.

This paper describes an associative technique for holding a universe of triples on auxiliary storage and then accessing a triple in response to an inquiry.

The paper also shows how relational operations have been performed—on an experimental basis—with PL/I *as the language for the controlling program, using machine-language subroutines to perform only the basic functions on associative storage.*

INTERACTIVE GRAPHICS IN DATA PROCESSING

Auxiliary-storage associative data structure for PL/I

by A. J. Symonds

Computer graphics is usually associated with the interaction of a person with a complex data base via a display console. The computational problem involved is to devise a method of representing a set of related items of information in such a way that, on demand, any subset of related items can be transmitted to the user, and that the user can transmit any desired modification to the data base. In this respect, this situation is common to all types of information retrieval. The display console user, however, has the additional demand of a very fast response.

In designing a data structure for computer graphics, we must therefore pay much attention to a quick response to inquiries. Also, experience has indicated that the hierarchical data structure often used for retrieval systems is not adequate to represent complicated three-dimensional geometry. For computer graphics, we must be able to represent a generalized directed graph[1] in the data structure.

The data structure described here attempts to achieve these aims and was originally conceived for graphics. However, this data structure could equally well be used for most information retrieval applications. Clearly, such a large, complex data structure must reside on direct-access storage and must be organized in such a way that related items of information can be extracted with the minimum of disk accesses.

Previous techniques for modeling objects (for visual display, for instance) have been based on list structures (including ring structures[2]), an item of information such as a line being represented by a storage block containing some descriptive information and a number of pointers. One pointer might link the line into a ring composed of all the lines in the drawing; others might link the line to blocks describing its terminal points which in turn are linked to other lines. Thus, we see how a drawing could be described, for the purposes of visual display, by a ring-like structure residing in main storage. Further research in graphics has, however, indicated a number of defects in the ring technique, of which some important ones are:

- The size of the structure is limited by the amount of main storage available.
- Transferring a ring structure to auxiliary storage causes problems when extensive searching of a ring is required in order to locate a particular member. The possibly large number of auxiliary-storage accesses drastically increases the search time.
- The maximum number of relations (i.e., rings) for a data item is equal to the number of pointers in the information block. This number is often fixed when the system is designed, thus severely limiting the facilities available to a user. Although blocks can be rewritten dynamically, this requires excessive storage.

The work described in this paper represents the first phase of a project to evaluate the possibilities of building a complex data structure and accessing it, all within the environment of PL/I.[3] As a first step, an experimental system, consisting of subroutines to be called from PL/I compiled code, has been built and is discussed here.

System concepts

Before presenting a detailed account of the system implementation, we now give an outline of some of the underlying system concepts.

The generalized associative storage can be represented as

structure of data base

$$\text{Location (X)} = \text{F (Identifier (X))} \qquad (1)$$
$$\text{L(X)} = \text{F(I(X))}$$

where X is a collection of information, I(X) is a unique identifier associated with X, L(X) is the physical location of X, and F transforms I(X) to L(X). An example of a hardware associative storage is found on paging computers (such as the IBM SYSTEM/360 Model 67), where the dynamic relocation hardware converts a virtual address, I, to a real address, L, in main storage. The principal difference between associative and conventional storage is that in the former, the location of information is dependent on the information itself.

In 1965, Feldman[4] suggested that the information X should be in the form of an ordered triple of binary numbers, each of which

identifies data associated with it, such that the meaning would be:

Attribute (Object) = Value
A (O) = V

The triple represents the basic unit of information and is stored in associative storage. A typical triple which might be stored in a graphics application is STARTPOINT (LINE1) = POINT3, where the two items LINE1 and POINT3 are related by the fact that POINT3 is the starting point of LINE1. The number of ways a triple can be specified by one or more elements is seven, as shown in Table 1. A question mark indicates that the particular element in the triple is unspecified. The forms contained in Table 1 are known as Simple Associative Forms (SAF's) which represent the seven basic ways an inquiry can be made of the store of triples. The result of an inquiry can be a Boolean value indicating existence or nonexistence of a triple satisfying the SAF in the associative storage. Alternatively, the result can be a collection of items comprising the unknown elements in the SAF (except in the case where all elements are specified). When two elements in an SAF are specified, the address of the triple in associative storage is found by performing an associative function, F, on the specified elements, as in Equation 1.

Table 1 Specifying triples

A (O) = V
A (?) = V
A (?) = ?
A (O) = ?
? (?) = V
? (O) = V
? (O) = ?

The problem we face, then, is to map ordered pairs of numbers (the known elements of a triple) into the storage area containing the universe of triples. We could obviously reserve a cell for each ordered pair; but this would mean allocating an enormous amount of storage, which would then probably be very sparsely populated with triples. A useful mapping function must therefore effectively "compress" all possible triples into a smaller space, designed to accommodate only the number of triples likely to be encountered in real applications. A suggested method of achieving this is to perform a binary operation on the ordered pair, generating an address that falls inside the associative storage; this is known as *hashing*. Hashing the two specified elements of a triple makes it possible to locate the triple immediately.

A consequence of "compressing" the triples in this way is that more than one pair of items can hash to the same address, causing a situation known as *conflict*. In this case, one triple can be situated at the hashed address, and conflicting triples must occupy spare cells in associative storage. In order to identify conflicting triples, all conflicts must be linked together in a conflict list.

We have thus far considered a situation where a pair of known items generates only one triple. Suppose the Attribute and Object are known: obviously there can be more than one Value, as in the following example:

CHILD (BILL) = MARY
CHILD (BILL) = JOHN etc.

The triples CHILD (BILL) = MARY, JOHN, etc., are called *multiple hits*, and we again have a situation where more than one triple is contending for the same cell, whether it be a cell located at a hashed

address or a conflict cell. Multiple hits are allocated to spare cells in associative storage and are also organized in a list; thus every hit can be accessed in answer to an SAF.

As a further complication, some SAF's only specify one element, A, for example, of a triple. Given A, we must locate all triples containing this value of A and extract all the ordered pairs (O, V). To be able to do this, all triples with a common element, whether it be A, O, or V, must be linked together in a ring.

We thus see that, in reality, an associative collection of triples can be quite a complicated structure. Also, if the conflict lists become too large, this technique shares one of the disadvantages of list processing, namely, the necessity for a long search before a triple can be found. The associative storage must, therefore, be large enough to prevent the buildup of too many conflicts.

Suppose we have an associative storage with N_A cells addressable by hashing, containing N_T triples. Then the probability P of a cell having at least one conflict cell attached to it is given by

$$P = 1 - [1 + (N_T - 1/N_A)][1 - (1/N_A)]^{N_T - 1}$$

This equation can be used—as an aid in designing associative storage—to decide on the best values for the following parameters:

- Ratio of conflict cells to addressable cells
- Number of available addressable cells
- Number of triples that can be stored before the value of P becomes intolerably high

The initial hashing to locate an addressable cell in an associative storage can be achieved by hardware or programming. Clearly, the hashing could be achieved much faster by using hardware. But since the likelihood of conflicts exists, it would be necessary to execute code to resolve such conflicts. The time required to execute this code obscures the advantage gained by performing the initial hashing in a few machine cycles. For this reason, all implementations of this type of associative storage have thus far used programming to achieve the hashing. We therefore talk about "program-simulated associative storage."

methods of interrogation

Thus far we have considered a collection of triples which is interrogated by specifying a simple associative form; Feldman,[4,5] Rovner,[5,6] and Johnson[7] have all made contributions towards the development of an associative language in which inquiries to the data base are made via the associative FOR statement. For example,

```
FOR (A(O) = #X),
BEGIN;
(procedure-1);
END;
ELSE BEGIN;
(procedure-2);
END;
```

where #X indicates that the items satisfying the "value" element of the SAF be successively allocated to the previously free variable X; procedure-1 is then executed once for each X value. Should there be no hits, the ELSE clause is invoked.

An obvious extension is to link FOR statements together, the output of one being part of the input to the next:

FOR (A (O) = #X),
AND FOR (A′ (X) = #Y),
AND FOR (A″ (Y) = #Z),
BEGIN; (procedure-1); END;
ELSE BEGIN; (procedure-2); END:

The second FOR statement has a bound variable X, as one of the elements in its argument. This means that all the items allocated to X are successively substituted for X in the argument, and the FOR statement is executed for each resulting SAF. The results of the successive iterations of the second FOR statement are all allocated to the unbound variable #Y, but a link is preserved between each item X_i in X, and all the Y items generated by the SAF A′ (X_i) = #Y. In other words, we keep track of the X value corresponding to each Y value. Similarly, the third FOR statement produces a number of Z values corresponding to each Y value, which in turn corresponds to an X value. The result of executing the linked FOR statements is the creation of a *set of correspondences*, where a *correspondence* is defined as a unique value of the ordered list (X, Y, Z) in this example. Procedure-1 is then executed once for each correspondence.

Having established the idea of extracting information from the associative storage in the form of a set of correspondences, features to improve the sophistication of the inquiry can be added. A few examples are given below.

The argument of a FOR statement can be a number of SAF's linked by Boolean connectives, e.g.,

FOR (A (O) = #X OR A′ (O) = #X AND L (X) = M), BEGIN; . . .

Also, by giving a triple a unique identifier and by nesting SAF's, information can be stored in associative storage as an ordered n-tuple. For example,

DATE (SPOUSE (BILL) = MARY) = 1948

Here the relation SPOUSE (BILL) = MARY can be considered to represent an item of information, namely a marriage, and the 5-tuple indicates the date BILL and MARY were married.

It is possible to process more than one set of correspondences simultaneously. Suppose we have two sets A and B of correspondences, whose elements are a_i (i = 1 to N_A) and b_j (j = 1 to N_B) respectively; then we can enter the specified procedure once for each combination (a_i, b_j) of correspondences in the sets A and B. For example, if we wish to display all the lines and points in a

drawing in several different windows, we would have:

```
FOR (WINDOW (DRAWING) = #W),
F1: BEGIN;
FOR (LINE (DRAWING) = #L),
AND FOR (STARTPOINT (L) = #P OR ENDPOINT (L) = #P),
F2: BEGIN;
(display-procedure);
END F2;
END F1;
```

The display procedure is iterated for each pair of correspondences (W) and (L, P) by executing FOR statements recursively.

Above, we considered a set of correspondences as a set of ordered n-tuples; we can alternatively consider the collection of all the different values of a particular element in a correspondence as a set in its own right. In the above example, we can consider all the different values of L as a set, and for some problems this is a fruitful approach. It is interesting to note that Childs[8] has approached the entire problem of relations in terms of set-theoretic operations. This approach represents all triples with a particular attribute as a set of ordered pairs, but does not explicitly define the correspondence between elements of two or more such sets.

To date, two versions of an associative language have been implemented using hashing techniques to access a collection of triples. Feldman and Rovner[5] have designed the LEAP language involving associative extensions to ALGOL, using the VITAL compiler-compiler on the TX2 at MIT Lincoln Laboratories; Johnson[7] has used the SYSTEM/360 macro assembler as a preprocessor to build a Relational Processing Language (RPL) for a SYSTEM/360 version of SKETCHPAD III.

auxiliary storage considerations

Thus far we have discussed ways of manipulating a relational data structure residing in program-simulated associative storage, without concerning ourselves with the details of its implementation.

Rovner[6] made the first investigations into putting the universe of triples on an auxiliary storage device, when he considered the operation of such a system in a paged environment, and Johnson has extended his work[7] to propose a design for a system that does its own paging. The ideas of these authors have been taken as a basis for the system discussed in this paper, and a summary of their conclusions is now given.

The collection of triples is segmented into fixed-size blocks on the auxiliary storage device, and the algorithm to decide where a given triple is stored is as follows. Consider the triple (A, O, V); the attribute (or access-word) determines the cell that holds the triple, and the block can then be brought into main storage. A is then hashed with O (check-word) to find the address within the block in which the triple resides (assuming no conflicts or multiple hits). Thus, we see that all triples with the same access-word will be found in the same block. However, this scheme only allows speedy access to a triple if the following SAF's are specified:

A (O) = V
A (O) = ?
A (?) = ?

To satisfy a request specifying any of the other SAF's might require many auxiliary-storage accesses to search the associative storage before the required triple could be located. Rovner's solution was to represent the universe of triples in three different ways:

	Access-word	*Check-word*
A-space	A	O
O-space	O	V
V-space	V	A

A typical triple (A, O, V) thus resides at three different locations in associative storage. It can be located either by using A as the access-word and hashing A with O, or using O as access-word and hashing with V, or V as access-word and hashing with A. The answer to any SAF can be found in one access to a block in the appropriate representation of the store of triples.

System implementation

Using the concepts discussed thus far, we now consider the implementation of the data structure.

auxiliary storage design

A schematic diagram of the auxiliary storage organization as currently implemented on the IBM 2311 disk storage unit is shown in Figure 1. It should be noted that—although the items A, O, and V represent Attribute, Object, and Value respectively in the figure—there is in fact no restriction on the use of an item in the associative map. O could occupy the Value position in another triple, for example. The associative storage is segmented into fixed-length blocks which can be any multiple of 4096 bytes up to a maximum of 32,768. A brief description of the logical sections of the storage follows.

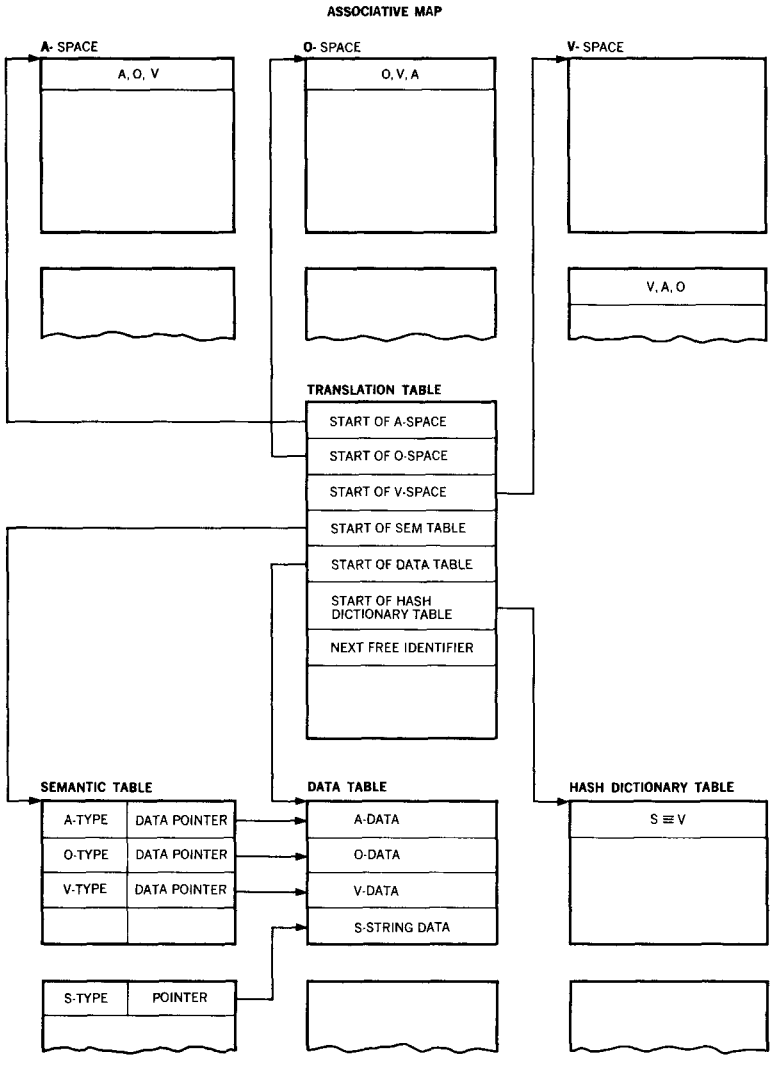
associative map

The associative map contains the collection of triples, which is triplicated for the reasons given above. Each block is divided up into 32-byte cycles, which are formatted as shown in Figure 2. Addresses generated by hashing will address only the leading cell of the cycle. The second 10-byte cell is reserved for conflicts, and the 6-byte cells are reserved for multiple hits. A description of the cells, as formatted in Figure 2, follows.

Cell 1. This is an addressable cell that has been assigned to a triple. Twenty-one bits are reserved for the elements of a triple, which means that there is, theoretically, a capacity for two million unique items and attributes in associative storage. The 2-byte link pointer chains triples with the same access word. The head of this ring is found at a location, L, in the block where

L = R (A/[2**N]) * [2** (NB − N)] + 16

Figure 1 Relational structure on auxiliary storage

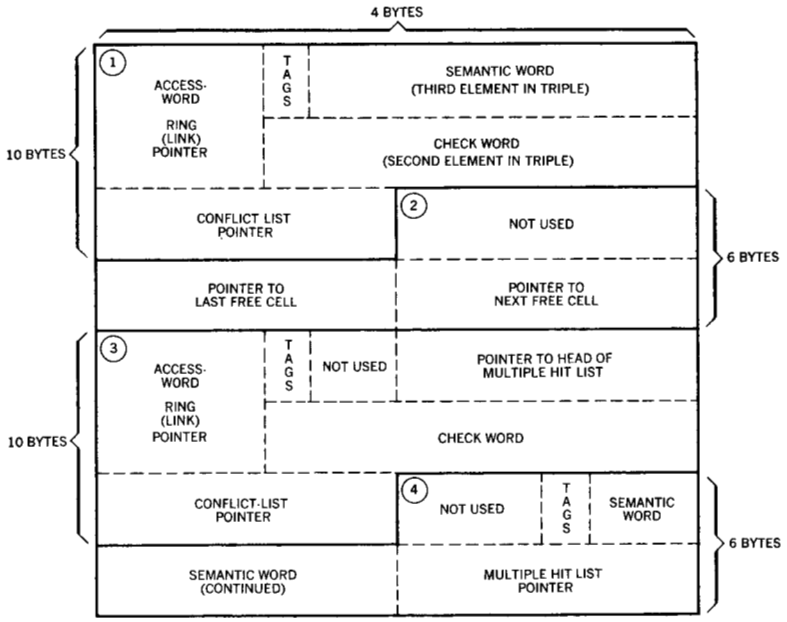


A ≡ access-word
R (X/Y) ≡ remainder after performing X/Y
2**N ≡ no. of access-words/block
2**NB ≡ no. of bytes/block

Therefore, 2**N 32-byte cycles have a slightly different format from that shown in Figure 2, namely

bytes 0 – 9: addressable cell
bytes 10 – 15: six byte cell
bytes 16 – 17: number of cells in ring linking cells with same access-word
bytes 18 – 19: points to head of ring linking cells with same access-word
bytes 20 – 31: two 6-byte cells

Figure 2 Format of 32-byte cycle in associative map



Cell 2. This is a free 6-byte cell which is chained into the free list of 6-byte cells. When a multiple-hit situation arises due to the addition of a triple, the first cell on the free list is removed and given the format shown in Cell 4 to represent the addition of the triple.

Cell 3. This cell is not directly addressable by hashing and is reserved for a triple that hashes to the address of an already occupied cell. This conflict cell is then inserted in a list that links all triples hashing to the same address. The particular conflict cell shown in Figure 2 is also the head of a multiple-hit list. This serves to identify the access-word and check-word, and points to the list of 6-byte cells containing the multiple semantic words.

Cell 4. This is a member of a multiple-hit list which is chained from a 10-byte cell at its head.

A free 10-byte cell is a member of the 10-byte free list and is similar to the 6-byte free cell. Each distinct block on auxiliary storage contains its own free list of 10-byte and 6-byte cells. All conflicts and multiple hits are then stored in the same block as the cell located at the original hashed address. Initially, the associative map is formatted by constructing completely empty blocks and writing them out to disk. At this time, the number of access-words to be allocated to each block is defined, usually larger for A-blocks than for O- or V-blocks. Adding a triple then requires accessing all three parts of the associative map to store it. The scheme described is based on the assumption that it is preferable to have at worst six disk accesses for updating the associative map in order to be able to satisfy an SAF with only one access.

The hashing scheme is based on exclusive ORing of the access-word and check-word, and then masking and shifting the result to lie on a 32-byte boundary within a block.

The main problem arising from this implementation of the associative map occurs when the number of triples involving a particular access-word becomes so large that the block is filled. Clearly, there must be an overflow procedure. Johnson[9] has suggested that the associative map be completely unstructured initially, and that—as triples are created—access-words are allocated to blocks depending on the observed density of triples. Also, when the conflict population increases in a particular block beyond a certain limit, triples are automatically redistributed between the original block and an additional block, which is dynamically allocated. This system is obviously attractive, because a user does not have to make a prediction as to what the triple density is likely to be before setting up the associative storage. However, the system adds considerably to the task of correlating the value of an access-word with the address of the block to which it is allocated.

An alternative approach would be to devise a means to distribute triples as uniformly as possible over the entire associative map, and to handle overflow more crudely as an exception condition. As yet, the overflow problem has not been approached, although it is recognized that a useful system must handle it. However, some of Johnson's proposals have been incorporated in the design of the present system, so that a more open-ended associative map can be constructed in the future. As the allocation of access-words to blocks becomes more random, a table-lookup procedure is required to correlate an access-word value with the location of a block on a disk. Also, as the number of access-words per block becomes variable, so do the hashing parameters. Thus, each block must carry information within itself to determine the hashing procedure needed to calculate the address of a triple.

semantic table

The semantic table contains pointers to the PL/I data aggregates corresponding to the items in the associative map. Each entry also has a type field which can be used to indicate the generic class to which an item belongs. The value of the identifier describing an item in the associative map is used to index the semantic table in order to find the appropriate entry.

data table

The data table is regarded as a contiguous store segmented into fixed-length blocks. There is provision in the semantic table for addressing up to sixteen million words of data. Storage in the data table is allocated to items as required, and standard garbage collection techniques are used when necessary.

translation table

The translation table is found at a predefined location on auxiliary storage, is read in before associative processing can take place, and is written back before closing down the system. This table contains the following housekeeping information necessary for system operation:

- Disk addresses of the start of different storage areas

- Next available identifier to be allocated to a data item
- Next free location in data table
- Tables linking access-words with block addresses when we have dynamic allocation of associative map blocks

hash
dictionary
table

Up to this point, we have described a data item by a 21-bit binary number, which is used to identify it in associative storage. This is an adequate representation for operations within a computer or for communications from one computer to another. However, a human being at a display terminal may wish to access an item in the associative storage by specifying its identifier directly. A convenient way for him to do this would be to activate a light-button containing a character-string mnemonic for the item or to type the mnemonic from the keyboard. Thus, we need a table to relate a character string to an internal identifier; this table, called hash dictionary table, functions as follows. The string is stored in the data table, and an identifier is allocated to it (*string-id*). The hash dictionary table resides on auxiliary storage, and the address of an 8-byte cell within it is computed by hashing the character string. The string-id and the internal-id to which it corresponds are stored in this cell so that, given the string itself, we can find the internal identifier to which it corresponds. The hashing technique used at the moment is to add significant bytes of the string to an accumulator (the number of bytes depending on the size of the hashed dictionary table), mask nonsignificant bits and use the result as the hashed address. Conflict is handled in an analogous fashion to the associative map.

data
structure
manipulation

Basic functions to enable the PL/I user to manipulate the data structure are now described. Before inquiries can be made of a data structure, it must first be built. The PL/I subroutine to store some information is

CALL ALLOCID (id-variable, type, data).

The PL/I major structure specified by "data" is stored in the data table, and an identifier is allocated to the data item and returned in "id-variable." Having allocated identifiers to items of data, we can then insert triples into, and delete them from, the associative map by issuing respectively

CALL AMMAKE (A, O, V)
CALL AMDELTE (A, O, V)

Having put some data into associative storage and stored some triples that relate data items, we need to devise a method for extracting correspondences. At the moment, this is accomplished entirely by calls to subroutines written in assembly language, which perform the primitive functions from which complex logical inquiries can be built. No effort has yet been made to design a meta-language that can be translated into PL/I source code by a preprocessor such as ML/I,[10] or interpreted at execution time by a

syntax analyzer. It is hoped that experience with this system will give some insight into a suitable structure for a meta-language based on FOR statements. For clarity, FOR statements will be used in the following discussion.

The execution of an associative FOR statement proceeds in two separate phases. The first phase involves the construction of a set of correspondences in main storage by repeated accesses to the associative storage, and the second phase is the iteration through a processing routine for each correspondence in the set. The correspondences are built in an area called *Correspondence Storage.* This is similar to *controlled* storage in the formal PL/I sense, with the difference that, once allocated, it expands automatically from 2048 bytes, in steps of 2048, up to a maximum of 32,768 bytes, as more space is required for correspondences. The calls to allocate and delete Correspondence Storage are CALL BUILDCS and CALL RELCS, respectively.

A set of correspondences is a set of ordered lists of elements. We shall define the total number of elements in a correspondence as its *depth,* and the position of an element within a correspondence as the element's *level.*

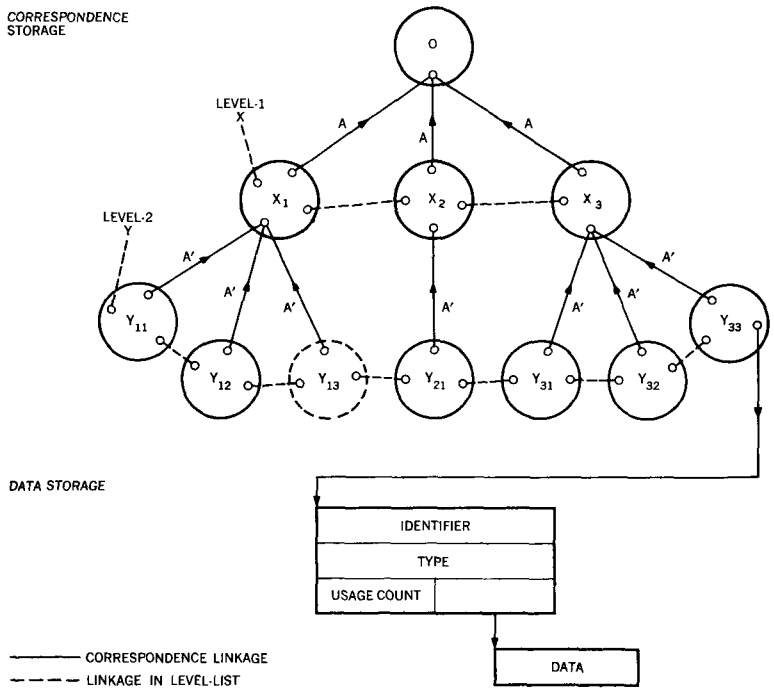
Let us consider the structure that is built up in Correspondence Storage as a result of the following string of FOR statements.

```
1) FOR (A (O) = #X),
2) AND FOR (A' (#Y) = X),
3) AND FOR (L (X) = Y OR L' (X) = Y),
   BEGIN; (proc-1); END;
   ELSE BEGIN; (proc-2); END;
```

The structure is shown schematically in Figure 3. The SAF serving as argument for FOR statement 1 has Attribute and Object as specified elements; therefore we access A-space (using A as access-word) in the associative map and hash A and O to find the multiple hit list of all triples A (O) = ?. The three hits X_1, X_2, X_3 are allocated to level-1 of the correspondence structure. They are also linked in a *level-list* and ordered in ascending sequence of identifier value, for reasons which will become apparent later.

Execution of the second FOR statement involves values of X; we thus proceed through the ring of X's, substituting the values of X_i (i = 1, 2, 3) in turn. Starting with the first item in level-1, namely X_1, we first satisfy the SAF A' (?) = X_1. Here the specified elements occupy the Value and Attribute positions in the triple so that we access V-space with X_1 as access-word. The multiple hit list in the relevant block is found by hashing as before. Y_{11}, Y_{12}, and Y_{13} are obtained from this multiple hit list and allocated to the level-2 list in the correspondence structure. The above procedure is then repeated for X_2 and X_3 using these values to locate the appropriate block in V-space. We can see immediately that ordering the items in level-list 1 will tend to reduce the number of disk-accesses required to process the list.

Figure 3 Correspondence structure



The resulting set of correspondences is a linear graph that can be viewed in two distinct ways. We can consider the tree formed by links between different levels, where each branch in the tree forms a correspondence. Alternatively, we can consider a particular level as a separate set in its own right and ignore correspondence links between levels. In the following discussion, we confine ourselves to the correspondence viewpoint, because this is the only approach to relational processing that has thus far been investigated in this work.

Once constructed, a set of correspondences can become the argument of a function which operates successively on each element of the set. The algorithm for executing a processing routine once for each set member is as follows. The set of correspondences is accessed at the lowest level, starting at the element at the head of the lowest level-list. The structure is designed so that we can back up through successive levels of each branch in the tree, as far as level-1; thus by backing up in this way, we can extract an individual correspondence from the set in the form of an ordered list, starting from the lowest level. By traversing the lowest level-list, each correspondence can be extracted in turn and made available to a processing routine.

In the example, the first operation on the correspondence structure consists of two existence tests connected by a Boolean OR. Each correspondence is processed in turn; the appropriate elements are inserted into each SAF, and the associative map is tested for the existence of the resulting triples. We see in Figure 3 that the corre-

spondence terminating in Y_{12} does not meet the specified conditions and is therefore deleted from the structure. After the Boolean operations, a test is made to see if any correspondences are left in the set; if there had been none in the example, the ELSE clause would have been executed. In fact, correspondences are still present, so proc-1 is then performed on each correspondence in turn.

The logic described above can be effected from a PL/I main program, using subroutines to perform the primitive functions on the associative storage and correspondence structure. A controlling program written in PL/I is used to traverse the level lists in the correspondence structure and pass control to the processing routine. Let us first consider the operation of the controlling program.

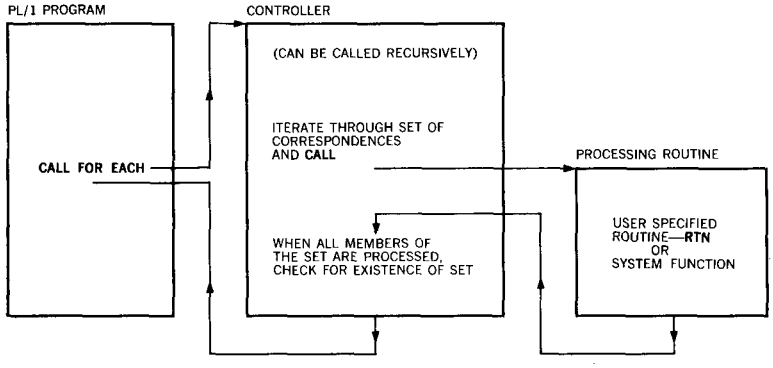
A single controller (FOREACH) called from the main PL/I program schedules entry to the program (RTN, say) designed to operate on each correspondence in the structure, and the flow of control from one routine to another is shown in Figure 4. The controller extracts correspondences individually from the set, and calls RTN once for each correspondence. The elements of the particular correspondence are made available to RTN, as well as further information which may have been passed from the main PL/I program. When the end of the set of correspondences is detected (i.e., the end of the lowest level-list is found), the controller checks that at least one member of the set remains and relays the result back to the main program. This is to enable an ELSE procedure to be invoked (proc-2 in the example) when the set is null.

The user's processing routine (RTN) is written with only one correspondence in mind and can perform a number of different functions. First, it can access the associative storage or correspondence structure using the primitives provided. The latter have been defined as follows:

- One element of a triple is specified. The function is to extract the unspecified items from the associative map in the form of ordered pairs, and then to add one member of each pair to the set of correspondences at a specified level. The second member of the pair is added to the correspondence structure at the next level down.
- Two elements of a triple are specified. The unknown element (several elements in the case of multiple hits) is extracted from the associative map and added to the correspondence structure at a specified level.
- Test existence of a triple in the associative map when all three elements are specified. The result is Boolean depending on the result of the test.
- Test existence of a triple when one element is unspecified.
- Test existence of a triple when two elements are unspecified.

A standard routine decides the section of the associative map to be accessed (A, O, or V) on the basis of which items in a triple are specified and which are not. Once the access-word, check-word, and

Figure 4 Flow of control to process a FOR statement



semantic-word are known, and the appropriate block brought into main storage, operation of the primitives is independent of which section of the associative map is being accessed. A final primitive, which can be called by RTN, deletes the correspondence under consideration from the set. This might be called when certain Boolean operations indicate that the particular correspondence does not meet the conditions defined in a FOR statement. Results of existence tests can be combined, using the Boolean facilities of PL/I, to create logical tests of any complexity whatever.

The user routine can itself contain relational statements, as the controller is a PL/I procedure with the attribute RECURSIVE.

Finally, the user routine can perform an application-dependent function on a set of correspondences. An example of this might occur in a graphics environment where a set of correspondences defines a drawing, each individual correspondence defining a line and its terminal points. To display the drawing, a user would simply write the code necessary to convert the coordinates of two points, and the parameters of the line connecting them, to display orders for a cathode-ray tube. Retrieving all the interconnected elements of the drawing and entering the display program for each element would be accomplished by relational statements.

The processing routine just described performs calculations on point coordinates and line parameters; but in our discussion of correspondences, we have thus far only considered accesses to the associative map of triples. Here an item of data is represented simply by a binary identifier, and this is also its only representation in the correspondence structure. Thus, before processing can take place, the data corresponding to each node in the current correspondence must be made available to the processing program.

Such data are stored in *data* storage, which is analagous to correspondence storage, and the data are brought into main storage, a level at a time, in two stages. First, the level-list is traversed and semantic table look-ups are performed for each member to obtain the pointer to the appropriate location in the data table.

The list is then traversed again, and the items of data are transferred from the data table to data storage. Duplication of identifier-values in different members of a level-list does not result in duplication of the data in data storage. We see again that ordering the identifiers in each level-list tends to reduce the number of disk accesses to the semantic and data tables.

Figure 3 shows how the node Y_{33} in the correspondence structure relates to its binary identifier, its type, and the data it represents.

Summary

A data structure to enable relational operations to be performed from PL/I has been implemented under the SYSTEM/360 Operating System (OS/360) using a 2311 disk unit as the storage medium for associative storage. At present, a user manipulates the data structure by means of subroutine calls, using a well-defined algorithm to construct them from a number of high-level FOR statements. They also enable him to access an auxiliary-storage-based data aggregation without explicitly requesting disk input/output. This work represents the first phase of an investigation into relational extensions to PL/I.

Future work could include the design of a command language which would enable the user to execute, via a display console, on-line relational statements of the type described. An attempt could be made to solve a real problem involving the manipulation of a large data base with complex relations between its components; the objective being to test feasibility of this approach to information retrieval. In addition, operation of the present system could be improved, particularly with regard to T. E. Johnson's proposals for handling the overflow problem. It would also be desirable to extend the facilities offered in the present system to include set-theoretic operations and the ability to store information in the form of n-tuples.

CITED REFERENCES AND FOOTNOTES

1. A hierarchical tree structure is a particular example of a graph, where there is one and only one path between two vertices.
2. A list is characterized by the requirement that access to its elements requires traversal of the list to obtain addressing information. A list structure occurs when we have a set of lists in which information from one set is related to another set by the header function of the second set. A ring structure exists if the information can be related—in general—through any member of either set.
3. PL/I *Compiler under the* SYSTEM/360 *Operating System,* IBM Systems Reference Library S360-29, C28-6571, PL/I Language, International Business Machines Corporation, Branch Office.
4. J. A. Feldman, "Aspects of associative processing," MIT Lincoln Laboratory Technical Note 1965-13, Cambridge, Massachusetts (1965).
5. P. D. Rovner and J. A. Feldman, "LEAP language and data structure," to be published in *Information Processing 1968, Proceedings of IFIP Congress 1968.*

6. P. D. Rovner, *Investigation into Paging a Software-Simulated Associative Memory System*, Master of Science Degree thesis at University of California at Berkeley, 1966.
7. T. E. Johnson, *Mass Storage Relational Data Structure for Computer Graphics and Other Arbitrary Data Stores*, MIT Department of Architecture Report, Cambridge, Massachusetts (1967).
8. D. L. Childs, *Description of a Set-Theoretic Data Structure*, University of Michigan Technical Report, 3, (March 1968).
9. T. E. Johnson, private communication.
10. P. J. Brown, "The ML/I macro processor," *Communications of the ACM* **10**, No. 10, 618–623 (1967).