The desire to express interrelationships between symbolic objects has been with us for some time, along with exploration of relationship systems which are operational in a computer. These systems coalesced under the term relational data systems (RDS), and a technology for dealing with this kind of data evolved. Relational systems have been through feasibility tests, experimental usage and should become generally available to the computing community in the near future. The advantages which account for the expanding use of RDSs are a simple, formal definition which allows associative processing, extreme flexibility in both structure and use, an ability to be efficiently implemented, and a notation and conception which is not dependent upon any particular physical data representation.

Earlier stages saw the use of relational systems in computer graphics, natural language systems, general data management tasks, and artificial intelligence research. Experience with such systems has shown, however, that while the technology is adequate for small problems, for practical work it needs further development, especially in the area of quantification over relational forms.

By quantification here we mean substitutional quantification, as opposed to other kinds such as referential or objective quantification. These others involve general problems of deduction, a topic much too large for this paper. The limitation to quantification over describable entities leaves us none of the philosophical problems, but only ones of engineering. Substitutional quantification is still important: it is our only means of summarizing, and searching across large portions of a data base. Thus, “Is there some NCC paper longer than 20 pages?” or “Which NCC papers discuss data base problems?” are examples of simple queries with quantifiers. A more complex example might be: “How many NCC papers in this conference reference at least four NCC papers which reference at most three other NCC papers?” Methods for making queries like these efficient have been investigated as part of the REL project and are incorporated into the current REL English system. This paper will discuss the nature of the problems and the types of solutions, along with some implications for relational systems design.

**RELATIONAL DATA SYSTEMS**

The following brief overview presents a vocabulary of relational systems. An RDS consists of a set of objects (often called entities, items, atoms, etc.) and a set of relations. Objects are primitive in the sense that they have no further structure, but can only be distinguished from one another and enter into interrelationships with each other. A relation is a set of ordered tuples (with the same number of elements in each). The degree of a relation $R$, written $\deg(R)$, is the number of elements in each of its tuples. A relationship is any single tuple in a relation, denoted by $[R A_1 \ldots A_n]$, where $R$ is the relation name and $A_1$ through $A_n$ form the tuple which is contained in $R$. Mathematically, given sets $S_1, \ldots, S_n$, an $n$-ary relation is a subset of $S_1 \times \ldots \times S_n$. $S_i$ is called the $i$th domain of the relation. Most RDSs single out the relations of degree 1 and call them sets or classes, and this convention will be followed here.

The primitive operations usually allowed in an RDS are:

1. creation and deletion of objects (or equivalently, the acquiring of the name of a heretofore unused object from a presumably infinite but fixed universe of discourse);
2. creation and deletion of relations;
3. the addition or deletion of a given relationship to a given relation;
4. a predicate which determines if a given relation contains a given relationship;
5. a retrieval function which, given a partially specified relationship, finds all relationships which match. A partially specified relationship means one in which some subset of the components of that relationship have been replaced by free variables.

Implementation questions in the past have dealt mainly with problems of efficient representation and access algorithms. The earliest representations were the LISP property list: each atom had associated with it a list of the form $\langle prop\_value1 \, prop\_value2 \ldots prop\_valuen \rangle$. Semantically, binary relations $\langle prop \rangle$ associate the atom with $\langle value \rangle$. 

---

*This research was carried out as part of the REL Project, principal investigators Bozena H. Dostert and Frederick B. Thompson, at the California Institute of Technology. It is supported by Office of Naval Research Contract N00014-67-A-0094-0024, National Science Foundation Grant #GH-31573, Rome Air Development Center Contract #F30002-72-C-0249.*
Operationally, given an atom and a "property," a linear search was conducted to find the value. Furthermore, only this particular access path is facilitated; to find the set of all pairs associated by a given property requires an inordinate amount of effort.

The need for symmetrical access in an efficient manner was first recognized by the LEAP originators.8 This system basically used hash-coding and redundant data storage to achieve outstanding performance. The implementation saves binary relations (roughly) by hash-coding any two of the three elements involved (in a relationship) together to get a location to store the third element. Thus, the data is triply redundant and access by any two items is fast. Access by any single item was facilitated by further structure, essentially a threaded list through the hashed items. Other implementations of relational systems represent relations as vectors of n-tuples, or matrices.4,12 This representation is, essentially, relatively simple and slow (by comparison) for retrieval of single items, though fast for bulk retrieval or update.

The relational systems mentioned above have all implemented substitutional quantification by means of explicit generators. The meaning of the term is clear from the following algorithmic interpretation of the predicate calculus statement "For all x, P(x)":

1. generate first (next) object in the universe
2. if no more to generate, exit with value "true"
3. bind the variable x to the name of the generated object
4. evaluate P(x)
5. if value is "false," exit with value "false"
6. otherwise (value is "true"), continue at Step 1.

Thus the classic quantifiers, for-all and for-some, can be interpreted as rather simple algorithmic forms. Note that the interpretation of P in no way depends upon its being used inside the scope of a quantifier, an important system simplification. The diversity of desired quantifiers has required other forms, and to describe their interpretation as generators we will use LEAP as the prototypical example.

LEAP has items, sets, and triples. Items are atomic, and objects are either items or numbers. Sets are unary relations, distinguished in syntax and implementation. A triple is a notation whose first element is the name of a binary relation and whose second and third elements are the components of that relation.

LEAP is embedded in an ALGOL-like language and uses the following syntax for a triple: [A.O = V], signifying "Attribute of Object = Value." A retrieval can be requested by replacing any one or two elements of a triple by a variable name in an appropriate LEAP statement.

LEAP has one construction for a quantified expression, the (loop statement) whose syntax is

FOREACH (associative context) DO (statement)

The (associative context) is a conjunction of Boolean expressions and retrieval triples. The operation of this construct can be described more easily in terms of a paraphrase:

FOREACH (binding list) SUCH THAT (associative context) DO (statement)

where (binding list) contains those variables mentioned in (associative context) that are not already bound to some value. (This latter syntax is actually used in SAIL,11 a descendant of LEAP.) The processing of this statement entails considering each of the conjuncts in the (associative context) in sequence, and using them both to filter values of variables already found and to retrieve possible values of other variables. The result of this process is a set of simultaneous values for all the variables in (binding list), each of which satisfy (associative context). The iterative execution of (statement) then takes place, with (binding list) variables being bound appropriately each time.

Note that in this LEAP operation, (associative context) is stated in terms of the primitive relational retrieval request of single relationships, and the system in fact implements the combined request by translating to that level. Both this imposed conceptual view and the implementation originate problems, which will be discussed in the next section.

PROBLEMS OF SIZE

The implementation of relational structures described above and the interpretation of quantifiers as generators have proved adequate in the past, but new applications with new requirements have revealed deficiencies. In most cases the problems are ones of efficiency, though there are also some conceptual implications.

The most immediate efficiency problem is one of size: today's data bases dwarf yesterday's. This large size means that a hierarchical memory environment is important, and dramatically influences the relevant operational characteristics of algorithms. Since the larger memory stores are slower and have more inertia than the smaller, primary stores, an algorithm's reference pattern to memory influences its elapsed time for execution. Particularly in the case of huge amounts of data, a slow execution time may be deemed equivalent to "impossible."

The LEAP hash-coded data structure was designed specifically with these problems in mind. For any single request to the data base, usually only one block of data need be brought from secondary to primary memory. That is, the algorithm references locations that are close to the desired data. None of the other implementations are as efficient for this purpose.

However, another important time for efficiency occurs during a quantified search, when a great deal of the data base must be checked. This is where the explicit generator method may be ineffective. To take the simplest case, for example, suppose we wish to find the image of a given class C, already known, under a given binary relation R. The LEAP statement would be

FOREACH x IN C AND [R.x = y] DO PUT y IN IMaGE.
Execution consists of generating the next member of the set C and binding that value to x, then doing an associative lookup on R and x, and for each value y found, adding it to the class IMAGE. In this simple case the best that LEAP can do is to reference some part of the relation R for each x value. If the data about R all fits into main memory simultaneously, then it will likely be brought in once and left there. However, when the relation R contains more data than will fit, a frequent occurrence, the reference pattern of this generator algorithm becomes unbearable.

To make this clear, suppose the routine has available K areas in main memory in which to place blocks of data, and suppose the data about relation R occupies M*K blocks. To find the value of R.x for a given, generated x, the system determines which block of R contains the information, and if that block is not already in main memory, brings it in. Since hash coding algorithms work best with uniformly distributed hashing functions, we can assume that any of the M*K blocks is equally likely to be required. Thus there is a probability of 1/M that a block must be brought in from the slow secondary memory for each value given x, that is, each member of the class.

The implications of this statement become more apparent when one considers that it is now easy to find data bases with classes containing from 1,000 to 100,000 members, and relations which are 10 to 100 times larger than available main memory. In these circumstances the hash technique will perform 1,000 to 100,000 input/output operations, while other methods, described below, require from 100 to 1,000.

The argument can be made, of course, that future generations of computers will have much larger primary memories and so this particular problem will disappear. What is not taken into account is that as our machine capabilities grow, the problems we wish to tackle will grow proportionally or even faster. In the case of memory size, larger capacity will allow us to consider data bases of the near future will be larger than available main memory, the hash-code technique must be built and utilized over a variety of domains.

To consider the problem of efficient implementation, we use the restriction operator as a concrete example. In general, the algorithm must find all relationships in R whose initial components match some relationship in S. Because of the statement of the operation, an obvious implementation suggests itself. If a relation is stored as a vector of tuples, the classic technique of sort/merge works marvelously.

The analysis of input/output activity is enlightening. Suppose that R is stored on Rb blocks of secondary memory and contains Rr relationships. In most systems, the number of relationships per block ranges from 100 to 1,000. Similarly, S occupies Sb blocks and contains Sr members. Since we are considering the case in which R and S are very large compared with available main memory, the hash-code technique must input approximately Sr blocks of R (one for each element of S). The dual algorithm will require Rr blocks of S.

A sort of R needs about 2Rb*log(Rb) blocks to be input and output to a temporary file, and so the entire process of sorting R and S, and then merging the two will require

\[ 2Rb\log(Rb) + 2Sb\log(Sb) + (Rb+Sb) \]

input/output operations. To compare these figures, assume for the moment that Sb = Rb = n and Sr = Rr = Kn (with K between 100 and 1,000). Then the sort/merge performs on the order of n*log(n) operations, while the hash-code requires Kn. Under these circumstances, assuming K at the minimum of 100 and only 10 frames of main memory available for the sort, the break-even point between these two algorithms is

\[ n = 10^*24 \]

For any smaller relations, sort/merge is better, and for any larger ones the hash coding is again more efficient. Obviously even data bases of the near future will be much smaller.

There is another algorithm which beats both of the previously mentioned ones over a certain range of data base sizes. Called the SUBSET algorithm, it also requires the storage of a relation as a vector of tuples. Using the same

**SIZE PROBLEM SOLUTION**

The solution to the memory reference problem has been rediscovered many times: group requests both spatially and temporally. For special purposes one can arrange the physical representation of the data and the accessing algorithms to maintain a locality of reference. In an RDS this means identification of those basic operations performed on relations by quantifiers and implementation with algorithms that have the appropriate characteristics. These operations become new "primitives" to the system and thus force new conceptualizations of the environment which have impact beyond the immediate reason for their introduction. Some of these implications will be mentioned in succeeding sections.

A complete list of primitive operations for a relational system falls beyond the scope of this paper, but some examples will convey the intent. Suppose R and S are n-ary and i-ary relations, respectively.

1. the permutation of R by k [k a permutation of the integers 1 through n] is that relation T such that if \((R_1, \ldots, R_n)\) is in R then \((R(k_1), \ldots, R(k_n))\) is in T.
2. the union of R and S (both n-ary) is that relation T which contains \((T_1, \ldots, T_n)\) if that relationship is in either R or S.
3. the restriction of R by S is that relation T such that if \((R_1, \ldots, R_n)\) is in R and \((R_1, \ldots, R_i)\) is in S (and \(i < = n\), then \((R_1, \ldots, R_n)\) is in T.

These new operations subsume certain quantified statements, similar in effect to a Skolemization. The use of these operators allows system recognition of particular quantification circumstances, and thus efficient handling. If, in fact, most use of quantifiers can be buried within such primitives, then extremely effective relational systems can be built and utilized over a variety of domains.
notation as before, suppose the algorithm can ascertain that it has enough main memory available for K blocks of a relation. It can divide the relation R into subrelations, each of which occupies (K-2) blocks or less. The algorithm then iterates over these subrelations, and for each, brings it into main memory in its entirety. It uses one frame for input of S (one block at a time), and the second free frame for an output area. (There is a dual algorithm that subdivides S.) This algorithm obviously displays more knowledge and control of its environment, but in return for this complexity, the algorithm can be a factor of five or so better than the sort/merge algorithm. This algorithm requires $Sb^\times \text{ceil}(Rb/K)$ (ceil means next-greatest-integer) input/output operations. This algorithm is of order $n^{1/2}$ in I/O operations (and CPU usage), and thus sort/merge is theoretically better in both. However, experimental results with the REL system prove that the SUBSET algorithm is the one to use for relations just a few times larger than available main memory. The REL data base manager actually computes the expected resource drain from each of the above algorithms and dynamically invokes the best one.

Notice that the physical representation of relations depends upon the algorithm used, and this in turn upon the predominant operations applied. As seems to be the case elsewhere, no general technique satisfies all requirements and circumstances. A well-designed interface to an RDS, however, with additional knowledge of the peculiarities of the situation, may enable the RDS itself to select the appropriate representation.

OTHER KINDS OF GENERAL OPTIMIZATION

Once the door is opened on the processing of quantified relational expressions, several general kinds of possible optimizations emerge. These have little relationship to the problem of size discussed above.

The first, and somewhat obvious, optimization is the removal of constant expressions from quantified phrases. This “do-loop optimization” is well-known by compiler writers, and consists of moving expressions in a hierarchical iterative structure outward as far as possible. Thus the statement

```plaintext
FOREACH x IN set1 DO
    FOREACH y IN set2 DO
        If sister of x = mother of y THEN PRINT (x,y);
```

should be translated to:

```plaintext
FOREACH x IN set1 DO
    BEGIN temp ← sister of x;
        FOREACH y IN set2 DO
            IF temp = mother of y THEN PRINT (x,y);
        END;
END;
```

In the particular case of relational systems, the primitives are small in number and have no side effects, and thus are ideal candidates for this kind of iterative optimization. There are basically two means for this. The first is a pass over the quantification form which identifies constant expressions and moves them, creating temporary variables as needed. The second method is akin to the operation of the Vienna Definition Language. Here each expression, when evaluated, replaces itself on the expression tree. Before every iteration, those expressions dependent upon the iteration variable get restored in the tree. Thus constants, relative to that particular iterative block, get evaluated only once, the first time through. This is the technique used by REL English language system, since the semantic operations are large and relatively few in number. With either method, though, there are large potential savings of execution effort.

The second general optimization area deals with ordering. This term covers several different problems and these especially pinpoint the need for considering relations as real entities with properties and primitive manipulatory operations. As an example of one type, consider the question of how to find the image of a class C under the composite relation RS. The system can either (1) find the image of C under S, then the image of the result under R, or (2) it can compute the composite relation RS and then directly find the image of C. This type of problem can be termed a “linear ordering” optimization. It is exactly here that the “explicit generator” view of quantification tends to hide the problem. The linear ordering optimizations deal with the associativity, commutativity, and distributivity of high order relational primitives. No research on linear ordering optimization has yet emerged in the literature.

Another type of ordering problem is one of simultaneous relational equations. In this case the system is asked to retrieve some objects that satisfy multiple relationship constraints. Thus, “find all x such that P(x)” and P(x) is a set of relationships, possibly including some internal quantification. These constraints, in fact, are used to guide the search. The problem concerns finding an optimum evaluation order, or, if the system allows simultaneous execution streams, finding the combination of parallelism and sequentiality that is optimum. The problem complications are that any particular atomic constraint can be used either to retrieve items or to filter already-found items, and there are differential cost functions of many parameters. A simple example, again from LEAP, is that of finding the sons of Bill:

```plaintext
FOREACH father.x = Bill AND sex.x = male DO
    PUT x IN sons
```

The order in which the clauses are processed may have a marked effect on performance, since it is expected that there will be many more males than children of Bill. But the analysis is much more complex, depending in part upon the asymmetries of access. Simultaneity optimizations are particularly needed in the pattern matching programs contained in modern artificial intelligence languages12 (“patterns” being partially specified relationships). Some research on the problem in the static, compile-time environment has been done,18 and the Automatic Programming project at the University...
of Southern California's Information Sciences Institute is currently investigating this area in the dynamic context.

CONCLUSION

This discussion of relational forms in quantificational situations has shown that efficient processing is possible, but at the cost of complexity. Each data representation and processing algorithm pair have a certain range of circumstances where that pair is appropriate. Relational Data Systems, then, can be built with one and only one such pair present, but this means a restricted domain of applicability. For a widely diverse domain, the system must be able to incorporate a variety of forms. To prevent difficulties in the use of such a collection, a stable, general interface is needed. This relational language should be independent of the particular representations chosen, and certainly should not hinder the use of some class of representations. It is the contention of this paper that the simple paradigm of quantifiers as explicit generators obstructs system recognition of some important situations and programmer recognition of the independent reality of relations.

The goal for future designers of Relational Data Systems consists of both a general descriptive language for relationship structures, and a system that implements those structures in appropriate ways, depending on the total environment. In the near future, of course, configuration decisions will be made by the programmer, but research into dynamic system decision-making may enable the programmer to ignore the problems of physical data representation, knowing that this will be done efficiently, and concentrate instead on the harder problems of logical structure.

REFERENCES


From the collection of the Computer History Museum (www.computerhistory.org)