System/U: A Database System Based on the Universal Relation Assumption

HENRY F. KORTH
IBM Watson Research Center
and
GABRIEL M. KUPER, JOAN FEIGENBAUM, ALLEN VAN GELDER, and JEFFREY D. ULLMAN
Stanford University

System/U is a universal relation database system under development at Stanford University which uses the language C on UNIX. The system is intended to test the use of the universal view, in which the entire database is seen as one relation. This paper decribes the theory behind System/U, in particular the theory of maximal objects and the connection between a set of attributes. We also describe the implementation of the DDL (Data Description Language) and the DML (Data Manipulation Language), and discuss in detail how the DDL finds maximal objects and how the DML determines the connection between the attributes that appear in a query.

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Theory—*path and circuit problems*; H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.3 [**Database Management**]: Languages—*data description languages (DDL), data manipulation languages (DML)*; H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Relational database, universal relation, hypergraphs, maximal objects

1. INTRODUCTION

In this paper we describe the design of System/U, a universal relation database system under development at Stanford University. The system is intended to test the use of the universal view, in which the entire database is seen as one relation. The system consists of two parts: the *DDL* (Data Definition Language) and the *DML* (Data Manipulation Language). The DDL is used to define the database, and the DML to process queries on it.

This work was supported by AFOSR grant 80-0212.
Authors' addresses: H. F. Korth, Dept. of Computer Science, University of Texas at Austin, Austin, TX 78712; G. M. Kuper, J. Feigenbaum, A. Van Gelder, and J. D. Ullman, Dept of Computer Science, Stanford University, Stanford, CA 94305

This paper describes the theory behind System/U and its implementation. Section 2 discusses the motivation for using the universal relation approach; Sections 3 and 4 describe the approach taken in designing System/U; Sections 5 and 6 describe, respectively, the DDL and DML from the user's point of view; Section 7 describes the algorithms used in the implementation of the DDL and DML; and, finally, Section 8 contains some concluding remarks about the implementation.

2. THE CASE FOR THE UNIVERSAL RELATION VIEW

We are interested in building a universal relation system because it relieves users of the responsibility for logical navigation. That is, just as the relational model relieves users of the responsibility for navigation within the physical database, a universal relation system relieves them of responsibility for navigation among the relations. (Further arguments in favor of the U. R. assumption are contained in [26] and [19].

Example 1. Suppose the database has attributes E (employee), M (manager), and D (department), that is, EDM is the universal relation scheme. The user should be able to say something like:

retrieve(D)
where E = 'Jones'

without concern for whether in the actual database scheme there is a single relation with scheme EDM, or two relations ED and DM, or even EM and DM. The response, regardless of the database structures, should be the same: the department or departments of Jones.

This universal relation does not actually exist, except in the user's mind. It may have nulls in certain components of certain tuples, and these nulls should be marked; that is, all nulls are different, unless their equality follows from a given functional dependency. For example, if we don't know Jones' address then there is a symbol that stands for "the address of Jones" in every tuple of the universal relation in which that address should logically appear, and in no others. Since the universal relation does not actually exist, the nulls may not have to appear in the actual database.

Let us not infer from Example 1 that no knowledge of the database's semantics is required of the user. It is a matter of taste whether understanding concepts like "employee" and "department" and their expected relationship is much easier on the user than understanding what an ED relation means. We think there is some gain in intuition to be had from the universal relation viewpoint, and, surely, many queries are easier to express in this way than in a query language that deals with individual relations. There is also some empirical evidence that the universal relation user view justifies the attention given to the concept:

(1) There has been a history of success in a variety of database applications using Brian Kernighan's "q" at Bell Laboratories [1]. This system supports a universal relation by means of a *rel file*, which is a list of joins (or other formulas) that could be taken if the query requires it; the first join on the list that covers all the needed attributes is taken. If there is no such join on the list, the join of all the relations is taken.

(2) A variety of natural language systems such as [8], [9], and [20] tacitly use the universal relation as a user view. Indeed, it is hard to see how a natural language system could reliably use anything else, although certain words could and are used to infer that certain relations are being talked about.

Let us therefore take as a point of departure that there is some utility in considering a system that supports the universal relation as a user view. We are not saying that every database should be implemented in this way, or even that they could be; we just contend that there is enough of a chance that the concept will be useful in a given context, with enough benefit to the user if it is, that the approach should be considered.

3. OBJECTS AND MAXIMAL OBJECTS

The System/U data model is based on the concepts of attributes, objects, and maximal objects. The *objects* in the database will be the edges of the hypergraph that defines the join dependencies assumed to hold in the universal relation (see [10]). They are, intuitively, the minimal sets of attributes that have collective meaning. The term is taken from [23], where the concept was first developed. For example, in terms of the entity-relationship model [7], an attribute or attributes that form a key for an entity set will be found in one object for each of the properties of that entity set; the object includes only the key and the one property. Relationships are represented by objects consisting of the keys for the related entity sets.

Each object is assumed to be contained in one relation, perhaps properly contained if the relation is unnormalized, or if the relation consists of a key and many properties of that key. We allow renaming of attributes so that the same relation can be used for many objects that are effectively identical, as in the following example.

Example 2. A genealogy can be based on a single relation *CP*, the child–parent relationship. We might declare attributes *PERSON*, *PARENT*, *GRANDPARENT*, and *GGPARENT* with the objects *PERSON–PARENT*, *PARENT–GRANDPARENT*, and *GRANDPARENT–GGPARENT*, each defined to be the *CP* relation with the obvious correspondence of attributes. We could then ask

retrieve(*GGPARENT*)
where *PERSON*='Jones'

and find the great grandparents of Jones in the obvious way, taking what the system thinks are natural joins, but are really equijoins on the *CP* relation.

Example 3. The banking example is the database scheme shown in Figure 1. The objects are *ACCT–BALANCE*, *ACCT–CUST*, *ACCT–BANK*, *CUST–ADDR*, *CUST–LOAN*, *LOAN–BANK*, *LOAN–AMT*.

In general we would like database schemes to be acyclic (see [4]). One reason is that there would then be a unique connection between attributes. The current example could be made acyclic by splitting *CUST* into *DEPOSITOR* and *BORROWER* and splitting *ADDR* into *DADDR* and *BADDR*. We would then use the objects *ACCT–DEPOSITOR*, *LOAN–BORROWER*, *DEPOSITOR–DADDR*, and *BORROWER–BADDR*. We do not recommend this approach for the following reasons: (1) It forces the user to remember the unimportant difference between

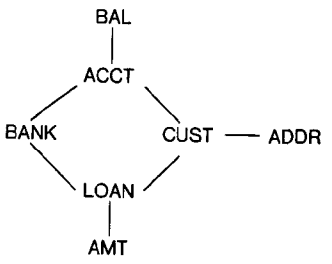

Fig. 1. The banking example.

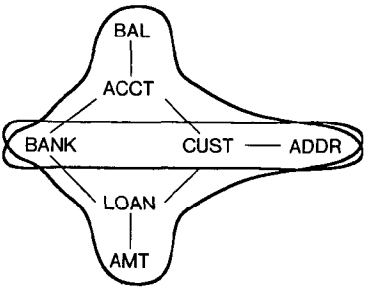

Fig. 2. Maximal objects in the banking example.

BADDR and *DADDR*. (2) We appear to need two relations recording names and addresses, many of which will be duplicates, when a customer is both a borrower and depositor. This problem, however, can be alleviated by creating one relation of names and addresses and declaring both objects *DEPOSITOR-DADDR* and *BORROWER-BADDR* to be this relation.

Maximal objects are the largest sets of objects in which we are willing to navigate. We currently require that these be acyclic. Declaring a maximal object is equivalent to declaring a join dependency on these objects, which enables us to define embedded multivalued dependencies. Various algorithms exist (see [13, 17]) for finding suitable maximal objects, using the functional dependencies. However, these do not necessarily result in the most intuitive maximal objects (though they seem to do so most of the time). The DDL uses the algorithm described in Section 7, which is based on [17]. The system then allows the user to examine the resulting maximal objects, and to redefine them if he or she wishes to. Their use will be described in the next section.

Example 4. Suppose that in the banking example of Figure 1 we decide that the functional dependencies:

$ACCT \rightarrow BANK, ACCT \rightarrow BAL, LOAN \rightarrow BANK, LOAN \rightarrow AMT,$

and $CUST \rightarrow ADDR$ hold. Then the two maximal objects shown in Figure 2 would be constructed. In this case these are probably the objects the user has in mind. The next section shows how these maximal objects lead to the expected response to queries.

4. [X]—THE CONNECTION ON X

Let t be a tuple variable in a query, and let X be the set of attributes A such that $t.A$ appears in the query. The *connection* on X, denoted $[X]$, is the relation beween the attributes X that is implied by the relations of the database. This is not necessarily the only possible relation on X, but rather the one that we regard as being intuitively the most basic. Each relational database system implicitly defines $[X]$ for each set of attributes, but the definitions in different systems are not always the same. A special case is that in which there is no natural connection between the attributes, in which case $[X]$ will be empty, and the system should return an error message.

Example 5. In the bank example, Figure 2, we could ask:

retrieve(*BANK*)
where *CUSTOMER* = 'Jones'

Among the possible meanings for this query are the following:

(i) Find all banks at which Jones has both an account and a loan.
(ii) Find all banks at which Jones has either an account or a loan.

The answer System/U will give to this query is (ii). We feel that this is what users would expect in this case. If users really want (i) or have some other meaning in mind, they can still force the system to give that answer. To get answer (i), two tuple variables are needed, one ranging over the top maximal object and one ranging over the bottom one.

The theoretical definition of $[X]$ used by System/U is the following: First find all maximal objects that contain all the attributes in X. If there are no such maximal objects, then there is no natural connection between the attributes in X, and $[X]$ is empty. If there is one such maximal object, then $[X]$ is the projection of the natural join of all the objects that make it up. If there are several such maximal objects, we first take the natural join for each maximal object, project onto the attributes in X, and take the union of the results. We make use here of the assumption that a join dependency holds on the objects in each maximal object.

In implementing System/U we modify this slightly. In order to avoid having to take the natural join of all the objects in each maximal object, we first take the *Graham Reduction* which throws away some objects and attributes, allowing $[X]$ to be computed from what is left (see Section 7 and [18]) of each maximal object, with sacred nodes X. This reduces the number of joins we have to take, and does not affect the result. After doing this, we take the natural joins and proceed as above.

5. THE SYSTEM/U DATA DEFINITION LANGUAGE

We now sketch the design of System/U and see how the ideas discussed above can be made to work in practice. Earlier descriptions of the system appear in [14] and in [15]. This section describes the Data Definition Language (DDL), used to define the database scheme, including the maximal objects. In the first

phase of the DDL, the user declares the following:

(1) Attributes and their data types. The types available are integer, floating point, and character string of length n.
(2) Relation names and their schemes (sets of attributes).
(3) Functional dependencies.
(4) Objects. Objects are sets of attributes. The set of attributes in each object must be a subset of the attributes in some relation. Users must specify from which relation the object is taken, with possible renaming of attributes allowed.
(5) Indices on the relations, to be used in the file access stage.

Example 6. The banking example would be entered as follows:

```
integer loan account;
float amount balance;
char [20] bank;
char [25] customer;
char [50] address;
relation rcust = customer, address;
relation rloan = customer, bank, loan, amount;
relation racct = customer, bank, account, balance;
object ocust in rcust = customer, address;
object oloancust in rloan = customer, loan;
object oloanbank in rloan = bank, loan;
object oloanamt in rloan = loan, amount;
object oacctcust in racct = customer, account;
object oacctbank in racct = account, bank;
object oacctbal in racct = account, balance;
account → bank;
account → balance;
loan → bank;
loan → amount;
customer → address;
```

The second phase is the computation of the maximal objects: The system first computes maximal objects itself, using the declared functional dependencies and the multivalued dependencies implied by the join dependency on the objects.

Example 7. Continuing Example 6, we type the `compute` command, and the system computes the maximal objects as follows:

```
List of all maximal objects
symbol: 2    type: maximal object
objects:   oloanbank oloanamt ocust oloancust
symbol: 1    type: maximal object
objects:   oacctbank oacctbal ocust oacctcust
```

The "symbol names" 1 and 2 are given by the system, since the user has not declared them explicitly. Since we are satisfied with the resulting maximal objects, we type `end` and the DDL terminates.

If users are not satisfied with the maximal objects that the system produces, they can be overriden by declaring additional maximal objects, and undeclaring system-defined ones.

Example 8. In Example 6, suppose the functional dependency $LOAN \rightarrow BANK$ did not hold. That means, intuitively, that loans can be made by a consortia of banks. We then get

```
List of all maximal objects
symbol: 2    type: maximal object
objects:   oloanamt ocust oloancust
symbol: 0    type: maximal object
objects:   oloanamt oloanbank
symbol: 1    type: maximal object
objects:   oacctbank oacctbal ocust oacctcust
```

that is, the lower maximal object in Figure 2 is now replaced by two, *BANK-LOAN-AMT* and *CUST-ADDR-LOAN-AMT*. If we now ask the query:

retrieve (*BANK*)
where *CUSTOMER* = 'Jones'

we will only get the banks at which Jones has accounts, because *CUST* and *BANK* are now connected only by the top maximal object.

Perhaps readers feel that this answer is satisfactory, because the connection between *BANK* and *CUST* through *LOAN* is now "weaker" than through *ACCT*. More likely, readers feel that the connection through *LOAN* is still just as valid as that through *ACCT*. Intuitively, readers probably have in mind the model in which each bank in a consortium has made the loan to each borrower of that loan. If that is the case, then there is an embedded multivalued dependency $LOAN \rightarrow\!\!\!\!\rightarrow BANK \mid CUST$ (see [25]). It turns out that the practical effect of this multivalued dependency can be achieved by explicitly declaring the lower maximal object of Figure 2 to hold, even though it will not follow from the given functional dependencies or from the join dependency on the objects. In this case users would let the system find these three maximal objects, then they would delete the lower two (using the `unmaxobj` command), and declare the new maximal object explicitly. This is done as follows:

```
unmaxobj 2;
unmaxobj 0;
maxobj lower = oloanamt, ocust, oloancust, oloanbank;
end;
```

6. THE SYSTEM/U QUERY LANGUAGE

6.1 General Description

We now sketch the ideas behind the query language. The language itself is essentially QUEL [24], with the following important difference: Since all tuple variables range over the universal relation, there is no need for a range statement or declaration of tuple variables. Furthermore, an attribute A by itself is deemed to stand for $b.A$, where b is the *blank tuple variable*, a tuple variable that never appears but is inserted when we translate queries. In the vast majority of queries, all one needs is the blank tuple variable, but we provide others to be used when necessary, for example:

retrieve(*EMP*)
where $MGR = t.EMP$ and $SAL > t.SAL$

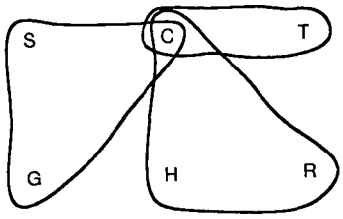

Fig. 3. Courses example.

The translation process we use for queries is

(1) For each tuple variable, including the "blank" tuple variable that we associate with attributes standing alone, assign a copy of the universal relation. Begin by forming an expression that is the Cartesian product of all these copies of the universal relation.

(2) Substitute for the copy of the universal relation associated with tuple variable t (which may be the blank tuple variable), the union of all those maximal objects that include all the attributes A such that $t.A$ appears in the query.

(3) Replace each maximal object by its Graham reduction, with the attributes A mentioned in (2) sacred. (The Graham reduction will be described in detail in Section 7.)

(4) Substitute for each reduced maximal object the natural join of all the objects in that reduced maximal object.

(5) The first four steps in this algorithm have been at the scheme level. We now pass to the instance level, and replace each object by an expression involving the actual relations in the database. Recall that each object is the projection (perhaps with renaming of attributes) of a relation in the database.

(6) Modify the above expression by applying to it the selections that are implied by the where–clause and the projection implied by the list of attributes in the retrieve–clause.

(7) The resulting expression is optimized by standard query optimization techniques, which are briefly described in Section 7.

Example 9. Consider the database scheme of Figure 3, where the objects are *CT*, *CHR*, and *CSG*, and let the relations in the actual database be *CTHR* and *CSG*; note that the first of these happens not to be normalized. This database scheme is the courses, teachers, hours, rooms, students, and grades example from [25], and the meanings of the attributes should be obvious. Let the query in question be

retrieve($t.C$)
where $s.S$ = 'Jones' and $s.R = t.R$

That is, print the courses that sometimes meet in rooms in which some course taken by Jones meets.

In step (1), we begin by using $C_1T_1H_1R_1S_1G_1$ as the copy of the universal relation corresponding to the tuple variable s, and the same set of attributes subscripted by 2 for the copy of the universal relation for the tuple variable t.

The database scheme of Figure 3 being acyclic, the only maximal object is the entire database [17]. As both t and s are surely associated only with attributes

that are in this one maximal object, the union at step (2) is simply this one maximal object in each case. In steps (3) and (4) we substitute the reduced maximal objects for the maximal objects in this union. The attributes which the tuple variable s mentions are S and R, and so the reduced maximal object is $\pi_{C_1S_1}C_1G_1S_1 \bowtie \pi_{C_1R_1}C_1H_1R_1$. The tuple variable t mentions C and R and so the reduced maximal object is $\pi_{C_2R_2}C_2H_2R_2$. Then, in step (5), we replace $C_1H_1R_1$ by $\pi_{C_1H_1R_1}(C_1T_1H_1R_1)$ and do the same for the object $C_2H_2R_2$.

In step (6), only C_2 is mentioned in the retrieve–clause, and the selection condition is that S_1 = 'Jones', and that $R_1 = R_2$. Thus we get, after slight simplifications,

$$\pi_{C_2}(\sigma_{S_1='\text{Jones}'\wedge R_1=R_2}((\pi_{C_1R_1}C_1T_1H_1R_1 \bowtie \pi_{C_1S_1}C_1S_1G_1) \times (\pi_{C_2S_2}C_2T_2H_2R_2))),$$

where π stands for projection and σ for selection. Finally, in step (7) we optimize this query.

Example 10. Let us consider an example where the universal relation must be composed of the union of several maximal objects because the underlying structure is cyclic. The banking example of Figure 2 will serve nicely. Consider the query

retrieve(*BANK*)
where *CUST* = 'Jones'

There is only one tuple variable, the blank one, so in step (1) we create a single copy of the universal relation. In step (2), we see that both maximal objects in Figure 2 include the attributes *BANK* and *CUST* mentioned by the query. Thus we get

BANK ACCT BAL CUST ADDR ∪ *BANK LOAN AMT CUST ADDR*

In steps (3) and (4) the upper maximal object is replaced by the join *BANK–ACCT* ⋈ *ACCT–CUST*, and the lower one by *BANK–LOAN* ⋈ *LOAN–CUST*. On the assumption that each of the objects in Figure 1 is also a relation, step (5) has no effect. Thus we get

(*BANK* − *ACCT* ⋈ *ACCT* − *CUST*) ∪ (*BANK* − *LOAN* ⋈ *LOAN* − *CUST*)

In step (6) we apply to this relation the operators $\pi_{BANK}\sigma_{CUST='\text{Jones}'}$, and get

$\pi_{BANK}\sigma_{CUST='\text{Jones}'}$((*BANK* − *ACCT* ⋈ *ACCT* − *CUST*) ∪ (*BANK* − *LOAN* ⋈ *LOAN* − *CUST*)).

6.2 Motivation Behind the Query Interpretation Algorithm

While many other interpretations of queries are possible, we feel that there are sound arguments in favor of the seven-step algorithm proposed in the previous section. The first two steps, construction of a Cartesian product of universal relations and application of selection and projection are analogous to the way QUEL handles queries [27].

Step (3), the replacement of the universal relation by the union of the maximal objects that are relevant to the query, is a matter for judgment. As mentioned previously, the idea that the union of possible connections is what we want has been expressed by several authors. For example, the extension join method for interpreting queries [22] when the only dependencies are functional, based on a key within one object (key dependencies), takes a union of connections to interpret queries.

Step (4), the Graham reduction of each maximal object is done for efficiency. In theory, we could have left the whole maximal object at this point and then removed unnecessary objects as part of the global optimization of the query.

Step (5), the replacement of the maximal object by the natural join of its member objects, follows from the fact that our construction of maximal objects guarantees the lossless join property for this decomposition of the maximal object [17].

Step (6) is mandatory, since we must ultimately talk about relations, not objects, and step (7) is a part of any query interpretation system.

6.3 The Query Language

As mentioned above, the syntax of the query language is based on QUEL, the query language in INGRES [24]. The language is an extension of the initial proposal for a query language contained in [15]. The entire language is parsed by the query interpreter; however, some of the features are not yet implemented, and are ignored by the rest of the program.

A query retrieves a set of tuples from the database that satisfy some predicate. The user must specify the attributes to be used to form tuples of this set and must give the predicate. The *retrieve* clause contains the list of attributes and the *where* clause contains the predicate.

The *retrieve* clause consists of the keyword `retrieve` followed by a list of attributes and aggregate operations. Each attribute name is preceded by a tuple variable (possibly the blank one). Members of this list are separated by a space or comma. The *where* clause consists of the keyword `where` followed by a Boolean expression. The following standard operators may appear in expressions:

(1) The arithmetic operators: +, −, /, *, % (modulo)
(2) The comparison operators: <, =, >, ≤, ≥

Operands may be constants, attributes, or aggregate operations (see below). String constants appear in double quotes. Parentheses may be used to dictate the order of evaluation. In the absence of parentheses, the precedence rules of C are followed. The system recognizes `and`, `or`, and `not` as Boolean operators. A null or missing *where* clause is the predicate TRUE.

Aggregate operators compute a set of tuples from a set of tuples. System/U offers the following aggregate operators: `average` (`avg`), `maximum` (`max`), `minimum` (`min`), `count` (`cnt`), and `sum` (`sum`). Aggregate operators must be performed with respect to a set of attributes, as explained in the following example.

Consider the universal relation U = (*EMP*, *CHILD*, *SAL*). What do we mean by the average salary? If we mean "sum the salary column and divide by the cardinality of the maximal objects," then the salaries of employees are weighted by the number of children that they have. In fact what we really mean by average salary is `average` *SAL* `of` *EMP*. We also include the "group by" feature for aggregate operators, as in most query languages. The general syntax for an aggregate operation is

aggop (*attribute* `of` *attributelist* `group by` *attributelist*)

```
for (all intersections i) {
  ctgobj = all objects containing i;
  for (all attributes a of i)
    mark a as being (temporarily) deleted from the hypergraph;
  for (all blocks b such that b contains i and
    b is not known to be acyclic) {
    for (all objects p in b) {
      if (p is not visited) {
        visit and mark all objects q such that there
        is a path from p to q in the hypergraph;
        m = list of marked objects;
        if (m is acyclic) {
          for (each c in ctgobj)
            add m to local maximal object list of c;
        }
      }
    }
  }
}
```

Fig. 4. Main procedure.

7. DESCRIPTION OF THE ALGORITHMS

7.1 DDL: Automatic Generation of Maximal Objects

Maier and Ullman [17] describe a method for constructing maximal objects from objects, functional dependencies, and multivalued dependencies. Fagin, Mendelzon, and Ullman [10] describe a method for deducing MVDs from a join dependency over all objects. The DDL implements both of these algorithms.

As part of the action performed when an object declaration is input, the DDL computes the intersection of the new object with all others and adds this list of attributes to a list of intersection lists. The first part of the maximal objects algorithm considers each intersection in turn. If the removal of the intersection disconnects a formerly connected component, then as shown in [10], the intersection multidetermines each newly formed connected component.

Suppose the removal of intersection I splits connected component C into $C_1, \ldots, C_n$. Each object containing I multidetermines C_i. If C_i is acyclic, it must be either a subset of, or disjoint from, any maximal object. Therefore, for each object P containing I, we add all objects of C_i to the *"local maximal object list"* of P, which holds objects that must be in any maximal object that P is in (see Figure 4). If object P_1 is on the local maximal object list of P_2, then by starting at P_2 and recursively following the local maximal object lists, we obtain at least as many objects as if we had started at P_1. At the end of the algorithm, the *root* of a maximal object will be an object with the property that if we start from it, and recursively follow its local maximal object lists, we get a maximal object. In our current situation, we clearly only know that if P_1 is a root, then P_2 must also be a root. Since we only need to find one root for each maximal object, we may as well assume that P_1 is not a root of a maximal object, and set its root field accordingly.

```
for (all objects p) {
  if (ROOT of p is set) {
    m = new, null maximal object;
    /* add local maximal object list of p to m */
    addin (m, p);
  }
}
```

Fig. 5. Collecting maximal objects.

```
addin(m, p) {/* Adds object p to the maximal object m */
  if (p already is in m) return;
  if (ROOT of p not set) {
    if (a maximal object n rooted at p has been generated)
      delmo(n); /* delete max obj n */
  }
  add p to object list of m;
  for (all attributes a in p) fdproc (m, a);
  for (all objects q on local maximal object list of p) addin (m, q);
  return;
}
```

Fig. 6. `addin(m, p)`.

If the removal of I splits a connected component into more than one connected component, I is an "*articulation point*". Following Fagin, Mendelzon, and Ullman [10], we put the objects of each such component into a separate block. Initially, all objects are in one block. In processing an intersection of objects we iteratively attempt to split each block not known to be acyclic. The definitions of [10] actually require us to split acyclic blocks as well. We do not do so, however, since such splitting does not cause any additions to the local maximal object lists of any root objects.

Once all intersections have been processed, we have to collect maximal objects by looking for root objects and scanning their local maximal object lists. (Figure 5). It may be that this process produces "maximal" objects that are contained in other maximal objects. It is easy to discover and eliminate such nonmaximal maximal objects.

The collection of maximal objects is complicated by the need to take functional dependencies into account. If the attributes in the objects we have so far collected functionally determine an object that is not part of our collection, we add this object and members of its local maximal object to our collection. This may permit still more functional dependencies to be used. We implement this process by a pair of mutually recursive functions to gather objects from local maximal object lists and to gather objects by the application of functional dependencies (Figures 6 and 7). These figures show the two functions `addin(m, p)` which adds the local maximal object list of object `p` to maximal object `m`, and will call `fdproc`; `fdproc(m, a)` adds each attribute `a` of `p` to the set of attributes in the closure of `m`, and will, if necessary, call `addin` recursively. These procedures compute two things, the maximal object `m`, which is a collection of objects, and its closure, which is a collection of attributes. The objects are added to `m` by `addin(m, p)`, and the attributes by `fdproc(m, a)`.

```
fdproc(m, a) {/* Adds attribute a to the closure of maximal object m */
  if (a already is in the closure of m) return;
  add a to the closure of m
  for (all FDs f with a on left side)
    if (all attributes on the left side of f are in m's closure)
      for (all attributes x in the right side of f) fdproc(m, x);
  for (all objects p containing a)
    if (all attributes of p are in the closure of m) addin(m, p);
  return;
}
```

Fig. 7. Processing FDs.

7.2 DDL: Creation of Join Trees

The algorithm used to compute $[X]$ requires a join tree for each maximal object to be created by the DDL. As shown in [4], such a tree exists iff the scheme is acyclic. We need the following definitions, in which G is the hypergraph representing a maximal object (assumed to be acyclic).

DEFINITION 1. *Let G be an acyclic hypergraph. A tree T whose nodes are the edges of G is a join tree for G, iff for all edges R_i and R_j in G and all nodes A of G, if R_k is on the path in T connecting R_i and R_j and A is in both R_i and R_j, then A is in R_k.*

DEFINITION 2. *Let R and S be edges in a hypergraph, and X a fixed set of nodes called sacred nodes. Then we say that R covers S (relative to the set X), COVERS(S, R) if every attribute A in S is either isolated and nonsacred (in no other edge and not in X) or is in R.*

DEFINITION 3. *Let G be an acyclic hypergraph, and X a set of nodes in G. The Graham reduction of G with sacred nodes X, denoted $GR(G, X)$ (see [18]), is obtained by repeatedly deleting any edge that is covered by another edge until no more deletions can be performed. We then delete all nonsacred isolated nodes from the remaining edges, and finally delete any edge that becomes empty.*

Note: It is shown in [4] that if G is acyclic, then $GR(G, \emptyset)$ is empty. They also show that the order in which we delete the edges does not affect the result. The notion of sacred nodes is due to [18].

After finding the maximal objects, the DDL creates a join tree for each. The join tree is created using a straightforward implementation of the algorithm of [4]. This consists of carrying out a Graham reduction with no sacred nodes. At each step of the reduction, if R_i is deleted because it is covered by R_j, then make R_i a child of R_j in the tree. The last edge to be deleted at the end of the Graham reduction is the root of the tree.

7.3 DML: Computation of $[X]$.

Given a set of attributes X, the following algorithm is used to compute $[X]$:

(1) Find all maximal objects containing X.
(2) For each such maximal object M, let G be its hypergraph. Compute $GR(G, X)$.

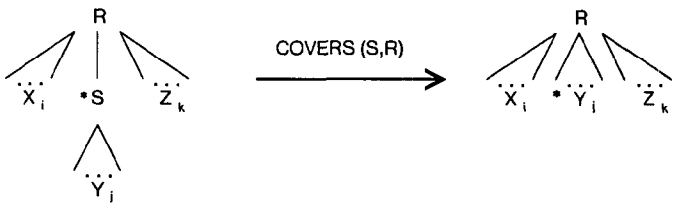

Fig. 8. Deletion of S in step 1 of the algorithm.

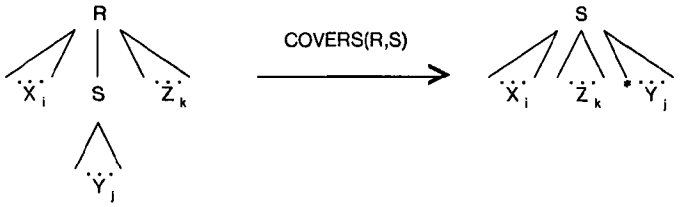

Fig. 9. Deletion of R in step 2 of the algorithm.

(3) If one set of nodes (objects) computed in (2) contains another one, then delete the smaller.
(4) $[X]$ is the union of the joins of these sets of objects.

In step (2), we use the following 2-step algorithm, where T is the join tree for the hypergraph:

(2a) Scan T from the bottom up. For each node R, examine its children from left to right. Let the current child be S, and let its children (if it has any) be Y_j, $j = 1, \ldots, m$. Also, let R's children to the left of S be X_i, $i = 1, \ldots, l$, and those to S's right be Z_k, $k = 1, \ldots, n$. If $COVERS(S, R)$ (see definition 2), carry out the transformation in Figure 8, and continue comparing R with its children from the node marked '$*$', the leftmost child of S. (If S has no children then continue from Z_1.) Node R is processed either when we compare R with its rightmost child and cannot delete the child, or when the rightmost child is a leaf, and we delete it.

(2b) Scan the tree from the top down. For each node R, examine its children from left to right. Let the current child be S, and let X_i, Y_j, Z_k be as in step 1. If $COVERS(R, S)$, carry out the transformation in Figure 9, and continue from the node marked '$*$', the leftmost child of S, with R replaced by S. (We do not compare S with the Z_ks.) A node is processed either when we compare it with its rightmost child and cannot delete it, or if it is a leaf and we delete its parent.

Note that in step 2b we do not have to test if $COVERS(S, Z_k)$ for $k = 1, \ldots, n$, since this can be shown to be impossible. A proof that the algorithm is correct can be found in [16], and Figures 10 and 11 show the implementation of the algorithm more precisely. These are recursive functions, initially called by the main program with their argument the root of the tree.

```
firstpass(R) {
  if (R is not a leaf)
     for (all children T of R) {
     firstpass(T);
     }
  S=leftmost child of R;
  repeat {
    if (the object at R covers the object at S) {
      delete S from the tree as in Fig. 8.
      repeat the loop, with S replaced
        by the node marked * in Fig. 8.
      if (no such node exists) end firstpass;
    }
    else {
      repeat the loop with S replaced by its right sibling.
      if (it doesn't have one) end firstpass;
    }
  }
}
```

Fig. 10. First pass of the reduction algorithm.

```
secondpass(R) {
  S=leftmost child of R;
  mainloop: repeat {
    if (the object at R is covered by the object at S) {
      delete R from the tree as in Fig. 9.
      repeat mainloop with R replaced by S
        and S replaced by the node marked * in Fig. 9.
      if (no such node exists) end mainloop;
    }
    else {
      repeat mainloop with S replaced by its right sibling.
      if (it doesn't have one) end mainloop;
    }
  }
  if (R is not a leaf)
    for (all children T of R) {
      firstpass(T);
    }
}
```

Fig. 11. Second pass of the reduction algorithm.

7.4 Optimization.

The phase of the query processor which is referred to as the *optimizer* performs step 7 of the translation process described in Section 6.1, as well as some nonstandard optimizations that prove useful under the assumptions of System/U. The output of the optimizer is a binary tree representing a relational calculus expression which is evaluated bottom-up by the execution phase of the System. The design and implementation of the optimizer are discussed at length in [11]; we summarize the main ideas here.

The optimizer starts with an unoptimized relational calculus expression tree for the query whose leaves represent objects of the database. First, the selection criteria from the query's **where** clause are "pushed" as far down the tree as possible, using the relational algebraic identities of Section 8.2 of [25]. The second and most interesting task of the optimizer is to choose the orders in which to

(1) join the objects within each maximal object,
(2) union the maximal objects for each tuple variable, and
(3) take the Cartesian product over all the tuple variables. Our goal when ordering the binary operations of the query is to keep the intermediate relations created during execution as small as possible; their sizes are estimated by heuristics. Finally, the optimizer "projects out" all the attributes of the objects which are neither mentioned in the query nor necessary for the joins, also according to the identities in [25].

When two objects to be joined "live in" the same relation, the join can often be eliminated. The conditions under which $O_1 \bowtie O_2$ can be eliminated are

(1) O_1 and O_2 are projections of the same physical relation R,
(2) the mapping of {attributes of O_1} $\cup$ {attributes of O_2} into {attributes of R} is injective, and
(3) a subset of {attributes of O_1} $\cap$ {attributes of O_2} functionally determines either all of the attributes in O_1 or all of the attributes in O_2. In this case, $O_1 \bowtie O_2$ is identical to the projection of R onto {attributes of O_1} $\cup$ {attributes of O_2}, as a direct consequence of Theorem 7.5 in [25]; hence these pairs are united before any others in the join of the whole reduced maximal object.

The universal relation oriented programs have been combined with an interface to *eris* [21] to provide a way to set up databases and answer queries for experimental purposes. Users use the System/U language described here; it is processed and translated to the *eris* language, which actually retrieves the information; then System/U displays the information to the users and stores it in a file. Both System/U and *eris* are written in C and run under the UNIX operating system.

8. CONCLUDING REMARKS

In this paper we describe the theory and implementation of the System/U database system. We describe the DDL and its implementation, including the automatic generation of maximal objects. We also describe the design of the DML. We have tested it on various real databases schemes, such as the one in [3], and the results have always corresponded to our intuition as to what answer users would expect from the query.

ACKNOWLEDGMENTS

F. Sadri wrote the first version of the DML. The query parser in the present version is based on his work.

REFERENCES

1. AHO, A. V. Private communication, June 1981.
2. AHO, A. V., BEERI, C., AND ULLMAN, J. D. The theory of joins in relational databases. ACM Trans. Database Syst. 4, 3 (1979), 297–314.
3. ATZENI, P., AND PARKER, D. S. Assumptions in relational database theory. In *Proceedings ACM Symposium on the Principles of Database Systems* (Los Angeles, Mar. 29–31, 1982), 1–9.
4. BEERI, C., FAGIN, R., MAIER, D., AND YANNAKAKIS, M. On the desirability of acyclic database schemes. *J. ACM 30*, 3 (1983), 489–513.
5. BEERI, C., AND KORTH, H. K. Compatible attributes in a universal relation. In *Proceedings ACM Symposium on the Principles of Database Systems* (Los Angeles, Mar. 29–31, 1982), 55–62.
6. BERNSTEIN, P. A. Synthesizing third normal form relations from functional dependencies. *ACM Trans. Database Syst. 1*, 4 (1976), 277–298.
7. CHEN, P. P. The entity-relationship model: Toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (1976), 9–36.
8. CODD, E. F., ARNOLD, R. S., CADIOU, J.-M., CHANG, C.-L., AND ROUSSOPOLOUS, N. Rendezvous version I: An experimental English language query formulation system for casual users of relational databases. Rep. RJ2144, IBM, San Jose, 1978.
9. DELL'ORCO, P., SPADAVECCHIO, V. N., AND KING, M. Using knowledge of a database world in interpreting natural language queries. In *Proceedings 1977 IFIP Congress*, North Holland, Amsterdam.
10. FAGIN, R., MENDELZON, A. O., AND ULLMAN, J. D. A simplified universal relation assumption and its properties. *ACM Trans. Database Syst. 7*, 3 (1982), 343–360.
11. FEIGENBAUM, J. The second phase of the System/U query processor. Unpublished manuscript.
12. HONEYMAN, P., LADNER, R. E. AND YANNAKAKIS, M. Testing the universal instance assumption. *Inf. Proc. Lett. 10*, 1 (1980), 14–19.
13. KORTH, H. F. On the implementation of the System/U data definition facility. Unpublished memorandum, Stanford Univ., Stanford, Calif., 1981.
14. KORTH, H. F. System/U: A progress report. In *Proceedings XP/2 Conference*, June, 1981.
15. KORTH, H. F., AND ULLMAN, J. D. System/U: A database system based on the universal relation assumption. In *Proceedings XP/1 Conference* (Stony Brook, N.Y., 1980).
16. KUPER, G. M. An algorithm for reducing acyclic hypergraphs. STAN-CS-82-892, Dept. of Computer Science, Stanford Univ., Stanford, Calif., 1981.
17. MAIER, D. A., AND ULLMAN, J. D. Maximal objects and the semantics of universal relation databases. *ACM Trans. Database Syst. 8*, 1 (1983), 1–14.
18. MAIER, D., AND ULLMAN, J. D. Connections in acyclic hypergraphs. STAN-CS-853, Dept. of Computer Science, Stanford Univ., Stanford, Calif., 1981.
19. MAIER, D., ULLMAN, J. D., AND VARDI, M. The revenge of the JD. In *Proceedings 2nd ACM Symposium on the Principles of Database Systems*, (Atlanta, Ga., Mar. 21–23, 1983), 279–287.
20. MOORE, R. E. Handling complex queries in a distributed database. Tech. Note 170, Artificial Intelligence, SRI International, Menlo Park, Calif., 1979.
21. REISS, S. P. Eris: The design and implementation of an experimental relational information system. CS-83-02, Dept. of Computer Science, Brown Univ., Jan. 1983.
22. SAGIV, Y. A characterization of globally consistent databases and their correct access paths. Unpublished memorandum, Dept. of Computer Science, Univ. of Illinois, 1981.
23. SCIORE, E. Null values, updates, and normalization in relational databases. Ph.D. dissertation, Princeton Univ., Princeton, 1980.
24. STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (1976), 189–222.
25. ULLMAN, J. D. *Principles of Database Systems.* Computer Science Press, Rockville, Md., 1982.
26. ULLMAN, J. D. The universal relation strikes back. In *Proceedings ACM Symposium on the Principles of Database Systems* (Los Angeles, Mar. 29–31, 1982), 10–22.
27. WONG, E., AND YOUSSEFI, K. Decomposition—a strategy for query processing. *ACM Trans. Database Syst. 1*, 3 (1976), 223–241.

Received February 1983; revised January 1984; accepted February 1984