The Database Language GEM

Carlo Zaniolo

Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

GEM (an acronym for General Entity Manipulator) is a general-purpose query and update language for the DSIS data model, which is a semantic data model of the Entity-Relationship type. GEM is designed as an easy-to-use extension of the relational language QUEL, providing support for the notions of entities with surrogates, aggregation, generalization, null values, and set-valued attributes.

1. INTRODUCTION

A main thrust of computer technology is towards simplicity and ease of use. Database management systems have come a long way in this respect, particularly after the introduction of the relational approach [Ullm], which provides users with a simple tabular view of data and powerful and convenient query languages for interrogating and manipulating the database. These features were shown to be the key to reducing the cost of database-intensive application programming [Codd1] and to providing a sound environment for back-end support and distributed databases.

The main limitation of the relational model is its semantic scantiness, that often prevents relational schemas from modeling completely and expressively the natural relationships and mutual constraints between entities. This shortcoming, acknowledged by most supporters of the relational approach [Codd2], has motivated the introduction of new *semantic data models*, such as that described in [Chen] where reality is modeled in terms of entities and relationships among entities, and that presented in [SmSm] where relationships are characterized along the orthogonal coordinates of *aggregation* and *generalization*. The possibility of extending the relational model to capture more meaning — as opposed to introducing a new model — was investigated in [Codd2], where *surrogates* and *null values* were found necessary for the task.

Most previous work with semantic data models has concentrated on the problem of modeling reality and on schema design; also the problem of integrating the database into a programming environment supporting abstract data types has received considerable attention [Brod, KiMc]. However, the problem of providing easy to use queries and friendly user-interfaces for semantic data models has received comparatively little attention[1]. Thus the question not yet answered is whether semantic data models can retain the advantages of the relational model with respect to ease of use, friendly query languages and user interfaces, back-end support and distributed databases.

This work continues the DSIS effort [DSIS] to enhance the UNIX* environment with a DBMS combining the advantages of the relational approach with those of semantic data models. Thus, we begin by extending the relational model to a rich semantic model supporting the notions of entities with surrogates, generalization and aggregation, null values and set-valued attributes. Then we show that simple extensions to the relational language QUEL are sufficient to provide an easy-to-use and general-purpose user interface for the specification of both queries and updates on this semantic model.

© 1983 ACM-0-89791-104-0/83/005/0207 $00.75

1. To the extent that the functional data model [SiKe] can be viewed as a semantic data model, DAPLEX [Ship] supplies a remarkable exception to this trend.

* UNIX is a trademark of Bell Laboratories.

```
ITEM( Name: c, Type: c, Colors: {c}) key(Name);
DEPT (Dname: c, Floor: i2) key(Dname) ;
SUPPLIER (Company: c, Address: c) key(Company);
SALES ( Dept: DEPT, Item: ITEM, Vol: i2) key(Dept, Item) ;
SUPPLY (Comp: SUPPLIER, Dept: DEPT, Item: ITEM, Vol: i2) ;
EMP (Name: c, Spv: EXMPT null allowed, Dept: DEPT,
    [EXMPT(Sal: i4), NEXMPT(Hrlwg: i4, Ovrt: i4)],
    [EMARRIED (Spouse#: i4), others]) key (Name), key (Spouse#) ;
```

Figure 1. A GEM schema describing the following database:

ITEM:	*for each item, its name, its type, and a set of colors*
DEPT:	*for each department its name and the floor where it is located.*
SUPPLIER:	*the names and addresses of supplier companies.*
SALES:	*for each department and item the volume of sales.*
SUPPLY:	*what company supplies what item to what department in what volume (of current stock).*
EMP:	*the name, the supervisor, and the department of each employee;*
EXMPT:	*employees can either be exempt (all supervisors are) or*
NEXMPT:	*non-exempt; the former earn a monthly salary while the latter have an hourly wage with an overtime rate.*
EMARRIED:	*Employees can either be married or not; the spouse's social security number is of interest for the married ones.*

2. THE DATA MODEL

Figure 1 gives a GEM schema for an example adapted from that used in [LaPi]. The attributes Dept and Item in SALES illustrate how an aggregation is specified by declaring these two to be of type DEPT and ITEM, respectively. Therefore, Dept and Item have occurrences of the entities DEPT and ITEM as their respective values. The entity EMP supplies an example of generalization hierarchy consisting of EMP and two generalization sublists shown in brackets. The attributes Name, Spv and Dept are common to EMP and its subentities in brackets. The first generalization sublist captures the employment status of an employee and consists of the two mutually exclusive subentities EXMPT and NEXMPT (an employee cannot be at the same time exempt and nonexempt). The second generalization sublist describes the marital status of an employees who can either be EMARRIED or belong to the **others** category. Although not shown in this example, each subentity can be further subclassified in the same way as shown here, and so on.

The Colors attribute of entity ITEM is of the set type, meaning that a set of (zero of more) colors may be associated with each ITEM instance; each member of that set is of type c (character string).

Name and Spouse# are the two keys for this family. However, since the uniqueness constraint is waived for keys that are partially or totally null Spouse# is in effect a key for the subentity EMARRIED only.

We will next define GEM's Data Definition Language, using the same meta-notation as in [IDM] to define its syntax. Thus {...} denote a set of zero or more occurrences, while [...] denotes one or zero occurrences. Symbols enclosed in semiquotes denote themselves.

A GEM *schema* consists of a set of uniquely named entities.

1. <Schema>: { <Entity> ; }

An *entity* consists of a set of one or more attributes and the specification of zero or more keys.

```
2. <Entity>: <EntName> ( <AttrSpec>
             {, <AttrSpec> } ) { Key }
```

Attributes can either be single-valued, or be a reference (alias a link) attribute, or be set-valued or represent a generalization sublist.

3. <AttrSpec>: <SimpleAttr> | <RefAttr>
| <SetAttr> | <Generalization sublist>

4. <SimpleAttr>: <DataAttr> [<null spec>]

5. <DataAttr>: <AttrName> ':' <DataType>

GEM's data types include 1-, 2- and 4-byte integers (respectively denoted by i1, i2 and i4), character strings (denoted by c) and all the remaining IDM's types [IDM].

The user can allow the value of a data attribute to be null either by supplying a regular value to serve in this role, or by asking the system to supply a special value for this purpose (additional storage may be associated with this solution).

6. <null spec>: **null**':' <datavalue> | **null**':' **system**

The option **null allowed** must be entered to allow a null link in a reference attribute.

7. <RefAttr>:
<AttrName>':' <EntName> [**null allowed**]

Set-valued attributes are denoted by enclosing the type definition in braces.

8. <SetAttr>: <AttrName>':' '{' <DataType>'}'

A generalization sublist defines a choice between two or more disjoint alternatives enclosed in brackets. The keyword **others** is used to denote that the entity need not belong to one the subentities in the list.

9. <Generalization sublist>:
'[' <Entity> {, <Entity>} , <Entity> ']'
| '[' <Entity> {, <Entity> } , **others** ']'

Repeated applications of this rule produce a hierarchy of entities called an *entity family*. We have the following conventions regarding the names of a schema.

Names: All entity names must be unique within a schema. Attribute names must be unique within an entity-family (i.e., a top level entity and its subentities). Attributes and entities can be identically named.

Any subset of the attributes from the various entities in a family can be specified to be a key; no two occurrences of entities in the family can have the same non-null key value.

The DDL above illustrates the difference between the relational model and the GEM model. Productions 1 and 2 basically apply to GEM as well as to the relational model, with relations corresponding to entities. In the declaration of attributes, however, a relational system would be limited to the pattern:

<AttrSpec>: <SimpleAttr>

<SimpleAttr>: <DataAttr>

Instead GEM's data model is significantly richer than the relational one. However, we will show that it is possible to deal with this richer semantics via simple extensions to the QUEL language and also to retain the simple tabular view of data on which the congeniality of relational interfaces is built.

3. A GRAPHICAL VIEW of GEM SCHEMAS

DBMS users' prevailing view of schemas is graphical, rather than syntactic. IMS users, for instance, perceive their schemas as hierarchies; Codasyl users view them as networks. Relational users view their database schema and content as row-column tables; this view is always present in a user's mind, and often drawn on a piece of paper as an aid in query formulation. Moreover, relational systems also use the tabular format to present query answers to users. For analogous reasons, it would be very useful to have a graphical — preferably tabular — representation for GEM schemas. A simple solution to this problem is shown in Figure 2.

There is an obvious correspondence between the in-line schema in Figure 1 and its pictorial representation in Figure 2; all entity names appear in the top line, where the nesting of brackets defines the generalization hierarchy. A blank entry represents the option "others". Under each entity-name we find the various attributes applicable to this entity. For reasons of simplicity we have omitted type declarations for all but reference attributes. However, it should be clear that these can be added, along with various graphical devices to represent keys and the option "null allowed" to ensure a complete correspondence between the graphical representation and the in-line definition such as that of Figure 1. Such a representation is all a user needs to realize which queries are meaningful and which updates are correct, and which are not[2].

2. A network-like representation can be derived from this by displaying the reference attributes as arrows pointing from one entity to another. The result is a graph similar to a DBTG data structure diagram with the direction of the arrows reversed. More alluring representations (e.g., using double arrows and lozenges) may be useful for further visualizing the logical structure of data (e.g., to represent the generalization hierarchies); but they do not help a user in formulating GEM queries.

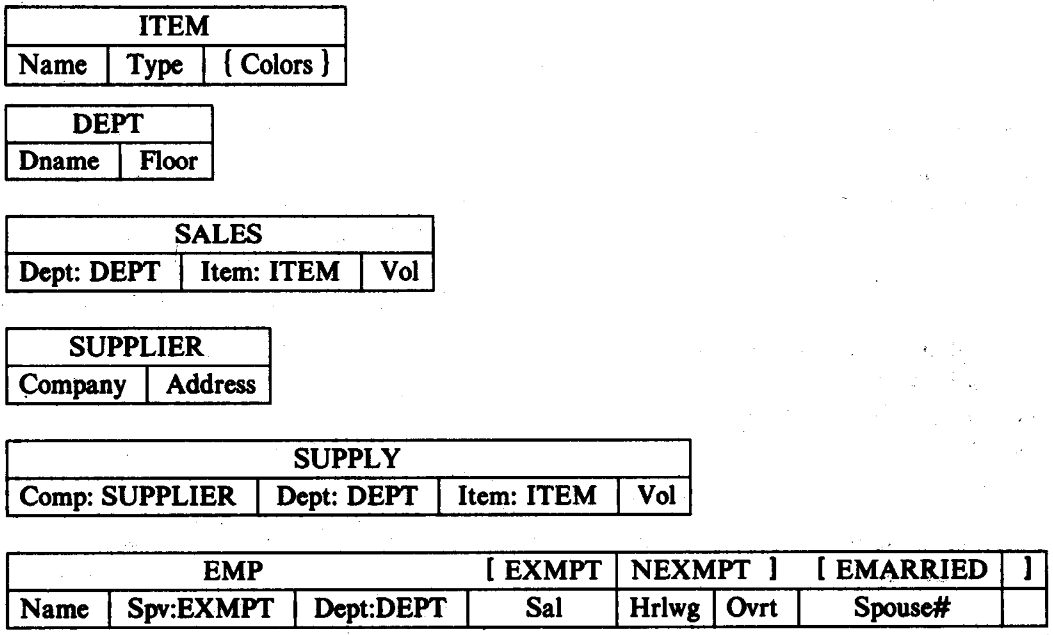

Figure 2. A graphical representation of the GEM schema of Figure 1.

Query results of GEM are also presented in output as row-column tables derived from these tables.

4. THE QUERY LANGUAGE

GEM is designed to be a generalization of QUEL [INGR]; both QUEL and IDM's IDL are upward-compatible with GEM. Whenever the underlying schema is strictly relational (i.e., entities only have data attributes):

<AttrSpec> → <SimpleAttr> → <DataAttr>,

GEM is basically identical to QUEL, with which we expect our readers to be already familiar. However, GEM allows entity names to be used as range variables without an explicit declaration. Thus the query, "Find the names of the departments located on the third floor," that in QUEL can be expressed as

range of dep **is** DEPT
retrieve (dep.Dname)
where dep.Floor=3

Example 1. List each department on the 3rd floor.

in GEM can also be expressed as:

retrieve (DEPT.Dname) **where** DEPT.Floor = 3

Example 2. Same as Example 1.

The option of omitting range declarations improves the conciseness and expressivity of many queries, particularly the simple ones; nor does any loss of generality occur since range declarations can always be included when needed.

The query of Example 2 is therefore interpreted by GEM as if it were as follows:

range of DEPT **is** DEPT
retrieve (DEPT.Dname)
where DEPT.Floor=3

Example 3. Same as examples 1 and 2.

Thus in the syntactic context of the **retrieve** and **where** clauses, DEPT is interpreted as a range variable (ranging over the entity DEPT).

Besides this syntactic sweetening, GEM contains new constructs introduced to handle the richer semantics of its data model; these are discussed in the next sections.

5. AGGREGATION and GENERALIZATION

A *reference* (alias *link*) attribute in GEM has as value an entity occurrence. For instance in the entity SALES, the attribute Dept has an entity of type DEPT as value, and Item an entity of type ITEM, much in the same way as the attribute Vol has an integer as value. Thus, while SALES.Vol is an integer, SALES.Dept is an entity of type DEPT and SALES.Item is an entity of type ITEM. An entity occurrence cannot be printed as such. Thus, the statement,

range of S **is** SALES
retrieve (S)

Example 4. A syntactically incorrect query.

is incorrect in GEM (as it would be in QUEL). Therefore, these statements are also incorrect:

range of S **is** SALES
retrieve (S.Dept)

Example 5. An incorrect query.

and

retrieve (SALES.Item)

Example 6. A second incorrect query.

Thus, reference attributes cannot be printed. However both single-valued and set-valued attributes can be obtained by using QUEL's usual dot-notation; thus

retrieve (SALES.Vol)

Example 7. Find the volumes of all SALES.

will get us the volumes of all SALES. Moreover, since SALES.Dept denotes an entity of type DEPT, we can obtain the value of Floor by simply applying the notation ".Floor" to it. Thus,

retrieve (SALES.Dept.Floor)
where SALES.Item.Name="SPORT"

Example 8. Floors where departments selling items of type SPORT are located.

will print all the floors where departments that sell sport items are located. The importance of this natural extension of the dot notation cannot be overemphasized; as illustrated by the sixty-six queries in Appendix II of [Zani3], it supplies a very convenient and natural construct that eliminates the need for complex join statements in most queries. For instance, the previous query implicitly specifies two joins: one of SALES with DEPT, the other of SALES with ITEM. To express the same query, QUEL would require three range variables and two join conditions in the **where** clause.

A comparison to functional query languages may be useful here. Reference attributes can be viewed as functions from an entity to another, and GEM's dot-notation can be interpreted as the usual dot-notation of functional composition. Thus GEM has a functional flavor; in particular it shares with languages such as DAPLEX [Ship] the convenience of providing a functional composition notation to relieve users of the burden of explicitly specifying joins. Yet the functions used by GEM are strictly single-valued non-redundant functions; multivalued and inverse functions and other involved constructs are not part of GEM, which is based on the bedrock of the relational theory and largely retains the "Spartan simplicity" of the relational model.

Joins implicitly specified through the use of the dot notation will be called *functional joins*. An alternative way to specify joins is by using *explicit entity joins*, where entity occurrences are directly compared, to verify that they are the same, using the *identity test operator*, **is**[3]. For instance the previous query can also be expressed as follows:[4]

range of S **is** SALES
range of I **is** ITEM
range of D **is** DEPT
retrieve (D.Floor)
where D **is** S.Dept **and** S.Item **is** I
and I.Type="SPORT"

Example 9. Same query as in example 8.

Comparison operators such as =, !=, >, >=, <, and <= are not applicable to entity occurrences.

The names of entities and their subentities — all in the top row of our templates — are unique and can be used in two basic ways. Their first use is in defining range variables. Thus, to request the name and the salary of each married employee one can write:

range of e **is** EMARRIED
retrieve (e.Name, e.Sal)

Example 10. Find the name and salary of each married employee.

or simply,

retrieve (EMARRIED.Name, EMARRIED.Sal)

Example 11. Same as in example 10.

Thus all attributes within an entity can be applied to any of its subtypes (without ambiguity since their names are unique within the family).

3. The operator **isnot** is used to test that two objects are not identical.

4. queries in Examples 8 and 9 are equivalent only under the assumption that the Dept attribute in SALES cannot be null. If some SALES occurrences have a null Dept link, then the results of the two queries are not the same, as discussed in detail in the section on null values.

Subentity names can also be used in the qualification conditions of a **where** clause. For instance, an equivalent restatement of the last query is

retrieve (EMP.Name, EMP.Sal)
where EMP **is** EMARRIED

Example 12. Same as example 11.

(Retrieve the name and salary of each employee who is an employee-married.) For all those employees who are married but non-exempt this query returns their names and a null salary. Thus, it is different from

retrieve (EXMPT.Name, EXMPT.Sal)
where EXMPT **is** EMARRIED

Example 13. Find all exempt employees that are married.

that excludes all non-exempt employees at once. The query,

retrieve (EMP.Name) **where**
EMP **is** EXMPT **or** EMP **is** EMARRIED

Example 14. Find all employees that are exempt or married.

will retrieve the names of all employees that are exempt or married.

In conformity to QUEL, GEM also allows the use of the keyword **all** in the role of a target attribute. Thus to print the whole table ITEM one need only specify,

retrieve (ITEM.**all**)

Example 15. Use of **all**.

In the presence of generalization and aggregation the **all** construct can be extended as follows. Say that t ranges over an entity or a subentity E. Then "t.**all**" specifies *all simple and set-valued attributes in E and its subentities*. Thus,

retrieve (EMP.**all**)
where EMP.Sal > EMP.Spv.Sal

Example 16. Extended use of **all**.

returns the name, the salary, the hourly wage and overtime rate, and the spouse's social security number of every employee earning more than his or her supervisor (the values of some of these attributes being null, of course); while

retrieve (EXMPT.**all**)
where EXMPT.Sal > EXMPT.Spv.Sal

Example 17. Use of **all** *with subentities.*

returns only the salaries of those employees.

The following query gives another example of the use of entity joins and the use of subentity names as default range variables (EMP and EMARRIED are the two variables of our query).

retrieve (EMARRIED.Name)
where EMP.Name="J.Black"
and EMP.Dept **is** EMARRIED.Dept

Example 18. Find all married employees in the same department as J.Black.

6. NULL VALUES

A important advantage of GEM over other DBMSs is that it provides for a complete and consistent treatment of null values. The theory underlying our approach was developed in [Zani1, Zani2], where a rigorous justification is given for the practical conclusions summarized next.

GEM conveniently provides several representations of null values in storage and in output tables; at the logical level, however, all occurrences of nulls are treated according to the no-information interpretation discussed in [Zani1].

A three-valued logic is required to handle qualification expressions involving negation. Thus, a condition such as,

ITEM.Type= "SPORT"

evaluates to **null** for an ITEM occurrence where the Type attribute is null. Boolean expressions of such terms are evaluated according to the three-valued logic tables of Figure 3. Qualified tuples are only those that yield a TRUE value; tuples that yield FALSE or the logical **null** are discarded.

It was suggested in [Codd2] that a **null** version of a query should also be provided to retrieve those tuples where the qualification, although not yielding TRUE, does not yield FALSE either. By contrast it was shown in [Zani2] that the TRUE version suffices once the expression "t.A **is null**" and its negation "t.B **isnot null**" are allowed in the qualification expression. Therefore, we have included these clauses in GEM. Thus, rather than requesting a null version answer for the query in Example 2, a user will instead enter this query:

OR	T	F	**null**
T	T	T	T
F	T	F	**null**
null	T	**null**	**null**

AND	T	F	**null**
T	T	F	**null**
F	F	F	F
null	**null**	F	**null**

NOT	
T	F
F	T
null	**null**

Figure 3. Three-valued logic tables.

range of dep **is** DEPT
retrieve (dep.Name)
where dep.Floor **is null**

Example 19. Find the departments whose floors are unspecified.

Indeed, this query returns the names of those departments that neither meet nor fail the qualification of Example 2 (dep.Floor =3).

In [Zani2] it is shown that a query language featuring the three-valued logic with the extension described above is complete — relational calculus and relational algebra are equivalent in power, as query languages. GEM, which is also complete, consists of a mixture of relational calculus and algebra, just like QUEL. In particular, both languages draw from the relational algebra inasmuch as they use set-theoretic notions to eliminate the need for universal quantifiers in queries. The treatment of set and aggregate operations in the presence of null values will be discussed in the next section.

R	
#	A
1	a1
2	a2
3	a3

S			
#	B	Ref1:R	Ref2:R
6	b1	1	2
7	b2	2	**null**

Figure 4. A database.

Null values make possible a precise definition of the notion of implicit join defined by the dot-notation. For concreteness consider the database of Figure 4. On this database, the query

retrieve (S.B, S.Ref1.A)

Example 21. Implicit join without nulls.

produces the following table. (For clarity we show the reference columns although they are never included in the output presented to a user.)

#	B	Ref1	A
6	b1	1	a1
7	b2	2	a2

Figure 5. The result of example 21.

However, the query

retrieve (S.B, S.Ref2.A)

Example 22. Implicit join with nulls.

generates the table,

#	B	Ref2	A
1	b1	2	a2
2	b2	**null**	**null**

Figure 6. Result of example 22.

We can compare these queries with the explicit-join queries:

retrieve (S.B, R.A) **where** S.Ref1 **is** R

Example 23. Explicit join without nulls.

and

retrieve (S.B, R.A) **where** S.Ref2 **is** R

Example 24. Explicit join with null.

Since two entities are identical if and only if their surrogate values are equal, these queries are equivalent (in a system that, unlike GEM, allows direct access to surrogate values) to:

retrieve (S.B, R.A) **where** S.Ref1=R.#

Example 25. Implementing Example 23 by joining on surrogates.

and,

retrieve (S.B, R.A) **where** S.Ref2=R.#

Example 26. Implementing Example 24 by joining on surrogates.

Applying the three-valued logic described above, one concludes that the queries of Examples 21 and 25 return the same result; however, the query of Example 22 produces the table of Figure 6, while that of Example 26 produces the same table but without the last row.

It can be proved that an implicit functional join, such as the one of Figure 6, corresponds to a semi-union join [Zani1], alias a semi-outer join [Codd2], which is in turn defined as the union of S with the entity join, S ⋈ R. Therefore, implicit functional joins are equivalent to explicit entity joins whenever the reference attributes are not null. Therefore, queries of Examples 8 and 9 are equivalent only under the assumption that SALES.Dept is not allowed to be null. However, nulls in SALES.Item would have no effect, since such tuples are discarded anyway because of the qualification, I.Type="SPORT".

7. SET-VALUED ATTRIBUTES and OPERATORS

The availability of set-valued attributes adds to the conciseness and expressivity of GEM schemas and queries. For instance, in the schema of Figure 2, we find a set of colors for each item:

ITEM (Name, Type, {Colors})

This information could also be modeled without set-valued attributes, as follows,

NewITEM (Name, Type, Color)

However, Name is a key in ITEM, but not in NewITEM, where the key is the pair (Name, Color). Thus the functional dependency of Type (that denotes the general category in which a merchandise ITEM lies) on Name is lost with this second schema.

A better solution, from the modeling viewpoint, is to normalize NewITEM to two relations: an ITEM relation without colors, and a COLOR relation containing item identifiers and colors. But this would produce a more complex schema and also more complex queries.

Thus inclusion of set-valued attributes is desirable also in view of the set and aggregate functions already provided by QUEL. In QUEL, and therefore in GEM, the set-valued primitives are provided through the (grouped) **by** construct. For example, a query such as, "for each item print its name, its type and the number of colors in which it comes" can be formulated as follows:

```
range of I is NewITEM
retrieve (I.Name, I.Type,
      Tot=count( I.Color by I.Name, I.Type))
```

Example 27. Use of **by**.

(Since Type is functionally dependent on Name, I.Type can actually be excluded from the **by** variables without changing the result of the query above.)

Using the set-valued Colors in ITEM, the same query can be formulated as follows:

```
range of I is ITEM
retrieve (I.Name, I.Type, Tot=count(I.Colors))
```

Example 28. Example 27 with a set-valued attribute.

Thus, ITEM basically corresponds to NewITEM grouped by Name, Type. Therefore, we claim that we now have a more complete and consistent user interface, since GEM explicitly supports as data types those aggregate and set functions that QUEL requires and supports as query constructs.

In order to provide users with the convenience of manipulating aggregates GEM supports the set-comparison primitives included in the original QUEL [HeSW]. Thus, in addition to the set-membership test operator, **in**, GEM supports the following operators:

=	(set) equals
!=	(set) does not equal
>	properly contains
>=	contains
<	is properly contained in
<=	is contained in

These constructs were omitted in recent commercial releases of QUEL [QUEL]. This is unfortunate, since many useful queries cannot be formulated easily without them — as demonstrated by the sixty-six queries in Appendix II of [Zani3].

Unfortunately, set operators are also very expensive to support in standard relational systems. Our approach to this problem is two-fold. First we plan to map subset relationships into equivalent aggregate expressions that are more efficient to support. Then we plan to exploit the fact that set-valued attributes can only be used in this capacity, so that substantial improvements in performance can be achieved by specialized storage organizations. Performance improvements obtained by declaring set-valued attributes may alone justify their addition to thè relational interface.

In the more germane domain of user convenience, set-valued attributes entail a more succinct and expressive formulation of powerful queries. For instance, the query "Find all items for which there exist items of the same type with a better selection of colors," can be expressed as follows:

> **range of** I1 for ITEM
> **range of** I2 for ITEM
> **retrieve** (I1.**all**) **where**
> I1.Type = I2.Type **and** I1.Colors < I2.Colors

Example 29. Items offering an inadequate selection of colors (for their types).

Thus, set-valued attributes can only be operands of set-valued operators and aggregate functions. The latter, however, can also apply to sets of values from single-valued attributes and reference attributes. Thus, to find all the items supplied to all departments one can use the following query:

> **retrieve** (SUPPLY.Item.Name) **where**
> {SUPPLY.Dept **by** SUPPLY.Item} >= {DEPT}

Example 30. Items supplied to all departments.

Observe that sets are denoted by enclosing them in braces. Also, GEM enforces the basic integrity tests on set and aggregate functions (sets must consist of elements of compatible type).

In the presence of null values, the set operators must be properly extended. A comprehensive solution of this complex problem is presented in [Zani1]; for the specific case at hand (sets of values rather than sets of tuples), that reduces to the following simple rule: Null values are excluded from the computation of all aggregate functions or expressions; moreover, they must also be disregarded in the computation of the subset relationship.

8. UPDATES

GEM supports QUEL's standard style of updates, via the three commands *insert*, *delete* and *replace*. Thus,

> **append to** DEPT (Dname="SHOES", Floor= 2)

Example 31. Add the shoe department, 2nd floor.

adds the shoe department to the database.

To insert a soap-dish that comes in brass and bronze finishes, one can write:

> **append to** ITEM (Name="Soap-dis",
> Type="Bath", Colors={brass, bronze})

Example 32. Inserting a new item.

Attributes that do not appear in the target list are set to **null** if single-valued; if set-valued, they are assigned the empty set.

The statement,

> **append to** ITEM (Name= "towel-bar",
> Type= ITEM.Type, Colors= ITEM.Colors)
> **where** ITEM.Name = "Soap-dish"

Example 33. Completing our bathroom set.

allows us to add a towel-bar of the same type and colors as our soap-dish. (According to the syntax of the **append to** statement, the first occurrence of "ITEM" is interpreted as an entity name, while the others are interpreted as range variables declared by default.)

Hiring T. Green, a new single employee in the shoe department under J. Black, with hourly wage of $ 5.40 and overtime multiple of 2.2, can be specified by the statement,

> **append to** EMP (Name="T.Green", Spv=
> EXMPT, Dept = DEPT, Hrlwg=5.40, Ovrt=2.2)
> **where** EXMPT.Name="J.Black"
> **and** DEPT.Dname= "SHOE"

Example 34. Adding a new single non-exempt employee.

In this statement, we can replace EMP by NEXMPT without any change in meaning since the fact that Hrlwg and Ovrt are not null already implies that the employee is non-exempt. Moreover, since no attribute of EMARRIED is mentioned in the target list, the system will set the new EMP to **others**, rather than EMARRIED. (If no attribute of either EXMPT or NEXMPT were in the target list an error message would result, since **others** is not allowed for this generalization sublist.)

If after a while T. Green becomes an exempt employee with a salary of $12000 and a supervisor yet to be assigned, the following update statement can be used:

replace EMP (Spv = **null**, Sal= 12000)
where EMP.Name = "T.Green"

Example 35. Tom Green becomes exempt and loses his supervisor.

Note that the identifier following a **replace** is a range variable, unlike the identifier following a **append to**. The fact that salary is assigned a new value forces an automatic change of type from NEXMPT to EXMPT. Finally, note the assignment of **null** to a reference attribute.

GEM also allows explicit reassignment of entity subtypes. Thus the previous query could, more explicitly, be formulated as follows:

replace NEXMPT
with EXMPT (Spv=**null**, Sal= 12000)
where NEXMPT.Name= "T.Green"

Example 36. Same as Example 35.

Say now that after being married for some time, T. Green divorces; then the following update can be used:

replace EMARRIED **with** EMP
where EMP.Name= "T.Green"

Example 37. T. Green leaves wedlock.

This example illustrates the rule that, when an entity e1 is replaced with an ancestor entity e2, all the entities leading from e1 to e2 are set to **others**. Thus the EMP T. Green will be set to **others** than EMARRIED.

The deletion of an entity occurrence will set to **null** all references pointing to it. Thus the resignation of T. Green's supervisor,

delete EMP.Spv
where EMP.Name = "T.Green"

Example 38. T. Green's supervisor quits.

causes the Spv field in T. Green's record, and in the records of those under the same supervisor, to be set to **null** (if null were not allowed for Spv, then the update would abort and an error message be generated), and then the supervisor record is deleted[5].

5. Of course, according to standard management practices T. Green's people may instead be reassigned to another supervisor, e.g. Green's boss; this policy can be implemented by preceding the deletion of Green's record with an update reassigning his people.

A request such as,

delete EXMPT
where EXMPT.Name="T.Green"

Example 39. T. Green goes too.

is evaluated as the following:

delete EMP
where EMP.Name="T.Green"
and EMP **is** EXMPT

Example 40. Same as above.

Thus, since T. Green is exempt, his record is eliminated; otherwise it would not be.

9. CONCLUSION

A main conclusion of this work is that relational query languages and interfaces are very robust. We have shown that with suitable extensions the relational model provides a degree of modeling power that matches or surpasses those of the various conceptual and semantic models proposed in the literature. Furthermore, with simple extensions, the relational language QUEL supplies a congenial query language for such a model. The result is a friendly and powerful semantic user interface that retains, and in many ways surpasses, the ease of use and power of a strictly relational one. Because of these qualities, GEM provides an attractive interface for end-users; moreover, as shown in [Andr], it supplies a good basis on which to build database interfaces for programming languages.

The approach of extending the relational model is preferable to adopting a new semantic model for many reasons. These include compatibility and graceful evolution, since users that do not want the extra semantic features need not learn nor use them; for these users GEM reduces to QUEL. Other advantages concern definition and ease of implementation. As indicated in this paper and shown in [Tsur, TsZa], all GEM queries can be mapped into equivalent QUEL expressions. In this way a precise semantic definition and also a notion of query completeness for GEM can be derived from those of QUEL, which in turn maps into the relational calculus [Ullm]. This is a noticeable improvement with respect to many semantic data models that lack formal, precise definitions. Finally, the mapping of GEM into standard QUEL supplies an expeditious and, for most queries, efficient means of implementation; such an implementation, planned for the commercial database machine IDM 500, is described in [Tsur, TsZa].

Acknowledgments

The author is grateful to J. Andrade and S. Tsur for helpful discussions and recommendations on the design of GEM. Thanks are due to D. Fishman, M. S. Hecht, E. Y. Lien, E. Wolman and the referees for their comments and suggested improvements.

References

[Andr] Andrade J. M. "Genus: a programming language for the design of database applications," Internal Memorandum, Bell Laboratories, 1982.

[Brod] Brodie, M.L., "On Modelling Behavioural Semantics of Databases," *7th Int. Conf. Very Large Data Bases,* pp. 32-42, 1981.

[Codd1] Codd, E.F., "Relational Database: A Practical Foundation for Productivity" *Comm. ACM,* 25,2, pp. 109-118, 1982.

[Codd2] Codd, E.F., "Extending Database Relations to Capture More Meaning," *ACM Trans. Data Base Syst.,* 4,4, pp. 397-434, 1979.

[Chen] Chen, P.P., "The Entity-Relationship Model — Toward an Unified View of Data," *ACM Trans. Database Syst.,* 1, 1, pp. 9-36, 1976.

[DSIS] Lien, Y.E., J.E. Shopiro and S. Tsur. "DSIS — A Database System with Interrelational Semantics," *7th Int. Conf. Very Large Data Bases,* pp. 465-477, 1981.

[HeSW] Held, G.D, M.R. Stonebraker and E. Wong, "INGRES: a Relational Data Base System," *AFIPS Nat. Computer Conf.,* Vol. 44, pp. 409-416, 1975.

[KiMc] King, R. and D. McLeod, "The Event Database Specification Model," *2nd Int. Conf. Databases — Improving Usability and Responsiveness,* Jerusalem, June 22-24, 1982.

[IDM] IDM 500 Software Reference Manual. Ver. 1.3, Sept 1981. Britton-Lee Inc., 90 Albright Way, Los Gatos, CA, 95030.

[INGR] Stonebraker, M., E. Wong, P. Kreps and G. Held. "The Design and Implementation of INGRES", *ACM Trans on Database Syst.* 1:3, pp. 189-222, 1976.

[LaPi] Lacroix, M. and A. Pirotte, "Example queries in relational languages," MBLE Tech. note 107, 1976 (MBLE, Rue Des Deux Gares 80, 1070 Brussels).

[QUEL] Woodfill, J. et al., "INGRES Version 6.2 Reference Manual," Electronic Research Laboratory, Memo UCB/ERL-M78/43, 1979.

[Ship] Shipman, D.W., "The Functional Model and the Lata Language DAPLEX," *ACM Trans. Data Base Syst.,* 6,1, pp. 140-173, 1982.

[SiKe] Sibley, E.H. and L. Kershberg, "Data Architecture and Data Model Considerations," *AFIPS Nat. Computer Conf.,* pp. 85-96,1977.

[SmSm] Smith, J.M. and C.P. Smith, "Database Abstractions: Aggregation and Generalization," *ACM Trans. Database Syst.,* 2, 2, pp. 105-133, 1977.

[Tsur] Tsur, S., "Mapping of GEM into IDL," internal memorandum, Bell Laboratories, 1982.

[TsZa] Tsur, S. and C. Zaniolo, "The Implementation of GEM — Supporting a Semantic Data Model on a Relational Backend", submitted for publication.

[Ullm] Ullman, J., "Principles of Database Systems," Computer Science Press, 1980.

[Zani1] Zaniolo, C., "Database Relations with Null Values," *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,* Los Angeles, California, March 1982.

[Zani2] Zaniolo, C., "A Formal Treatment of Nonexistent Values in Database Relations," Internal Memorandum, Bell Laboratories, 1983.

[Zani3] Zaniolo, C., "The Database language GEM," Internal Memorandum, Bell Laboratories, 1982.

Appendix I: GEM SYNTAX

This syntax is an extension of, and use the same metanotation as, that of [IDM]. Thus {...} denote a set of zero or more occurrences, while [..] denotes one or zero occurrences. Symbols enclosed in semiquotes denote themselves.

```
retrieve [ unique ] ( <query target list> ) [where <qualification> ]

<query target list>      : <query target element> '{' , <query target element> '}'
<query target element>   : <attribute> /* a simple or set-valued attribute*/
                         | <name> = <expression>
                         | <name> = <set>
<attribute>              : <variable> . <name>
<variable>               : <variable> { . <name> } /* every <name> must denote a reference attribute/*
                         | <range variable>   /* declared in the range statement/*
                         | <entity name>      /* range variable by default/*
<expression>             : <aggregate>  /* count(), average(), etc. /*
                         | <attribute>  /* a simple attribute/*
                         | <constant>
                         | - <expression>
                         | ( <expression> )
                         | <function> /* see [IDM] for a definition of functions /*
<set>                    : <attribute> /* a set-valued attribute/*
                         | '{' <extended expr> [ by <extended expr> { , <extended expr> } ]
                           [ where <qualification> ] '}'
                         | <constants set>
<extended expr>          : <expression> | <variable>
<constants set>          : '{' '}' | '{' <constant> {, <constant>} '}'
<qualification>          : ( <qualification> )
                         | not <qualification>
                         | <qualification> and <qualification>
                         | <qualification> or <qualification>
                         | <clause>
<clause>                 : <expression> <relop> <expression>
                         | < extended expr> in <set>
                         | <set> <relop> <set>
                         | <attribute> <identity test> null
                         | <variable> <identity test> <variable>
<relop>                  : = | != | < | <= | > | >=
<identity test>          : is |  isnot

append [ to] <entity name> ( < update target list>)

delete <variable> [ where <qualification> ]

replace <variable> [with <entity name>] (<update target list>)

<update target list> : <update target element> '{' , <update target element> '}'

<update target element>        : <name> = <expression> /* <name> of a simple attribute */
                               | <name> = <variable> /* <name> of a reference attribute*/
                               | <name> = <set> /* <name> of a set attribute */
                               | <name> = null  /* <name> of a simple or reference attribute/*
```