An Implementation of GEM — supporting a semantic data model on a relational back-end.

Shalom Tsur†
Carlo Zaniolo

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

This paper presents a simple approach for extending the relational system INGRES into one supporting a semantic data model. It describe a DBMS consisting of (i) a user-friendly front-end, supporting the GEM semantic data model and query language under the UNIX time-sharing system, and (ii) a dedicated back-end processor providing efficient support for database transactions, concurrency control and recovery. GEM extends the relational model to support the notions of entities with surrogates, the relationships of aggregation and generalization, null values and set-valued attributes, and provides simple extensions of QUEL to handle these new constructs. In this proposed implementation of GEM, the relational database processor IDM 500 by Britton-Lee is used as the back-end machine.*

1. Introduction

Data Base Management Systems (DBMSs) based on the relational approach have gained wide popularity due to their ability of providing users with a simple tabular view of data and high-level set-oriented Data Manipulation Languages (DMLs) for querying and updating the database. These features enhance the ease of use and data independence of these DBMSs, thus reducing the cost of database-intensive application programming [Codd1]; they are also important for providing a good environment for back-end support and distributed databases (particularly, because they reduce communication overhead).

The main limitation of the relational model is its semantic scantiness, which often prevents relational schemas from modeling completely and expressively the natural relationships and mutual constraints between entities. This shortcoming has motivated the introduction of *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*. Semantic data models provide a good basis for enterprise modeling and conceptual schema design, and for integrating programming languages and database facilities [Taxis, Brod, KiMc, Adplx, Catt].

An obvious approach to semantic data models consists of abandoning relational DBMSs and building new systems based on the new models. Of particular interest to us, because it focuses on high-level DML, is the work on DAPLEX [Ship] that provides a user-friendly DML supporting the notions of aggregation and generalization.

Here instead we explore the evolutionary approach of of adding semantic data model capabilities to relational systems. This approach is attractive from a practical viewpoint, since it ensures compatibility with existing systems and the preservation of the considerable know-how acquired in building DBMSs, database machines and distributed databases using the relational approach. This approach was previously advocated in [Codd2], where *surrogates* and *null values* were found necessary for the task. However, Codd concentrated on conceptual issues, without describing a specific architecture or how current relational systems can be modified to provide the new functionality.

A first objective of this research is to combine the advantages of the relational approach with those of semantic data models; a second is to preserve compatibility with existing relational DBMSs. Therefore, GEM was designed as an extension of INGRES that adds a rich set of semantic constructs to the relational model (such as the notions of entities with surrogates, generalization and aggregation, null values and set-valued attributes) and provides an easy-to-use and general-purpose DML for the specification of high-level set-oriented queries and updates on such a model.

A second objective of this work is to provide for better recovery and performance by building GEM on the top of a back-end system rather than on the current UNIX* file system, thus avoiding the problems described in [Stone]. We also wanted to design a system for experimenting with various front-end and back-end combinations all communicating through a standard interface. Thus we have an architecture based upon multiple layers of data representation along the lines recommended in [ANSI].

† Currently with the IBM Scientific Center, Haifa, Israel.

* UNIX is a trademark of AT&T Bell Laboratories.

Finally, we wanted to assess the suitability of current database machines to support semantic data models. Under the influence of [DeHa], we have selected an evolutionary and inductive solution approach to the problem. Thus, rather than postulating a new architecture and new commands to support GEM's new semantic constructs, we have decided to experiment first with a commercial back-end machine to determine its suitability to the new task, hoping to formulate recommendations for architectural improvements based upon lessons learned from this experience. The IDM 500 by Britton-Lee, used as our primary vehicle of back-end support, has influenced the design of our interface and provided ready-made software very expedient in our endeavor [IDM].

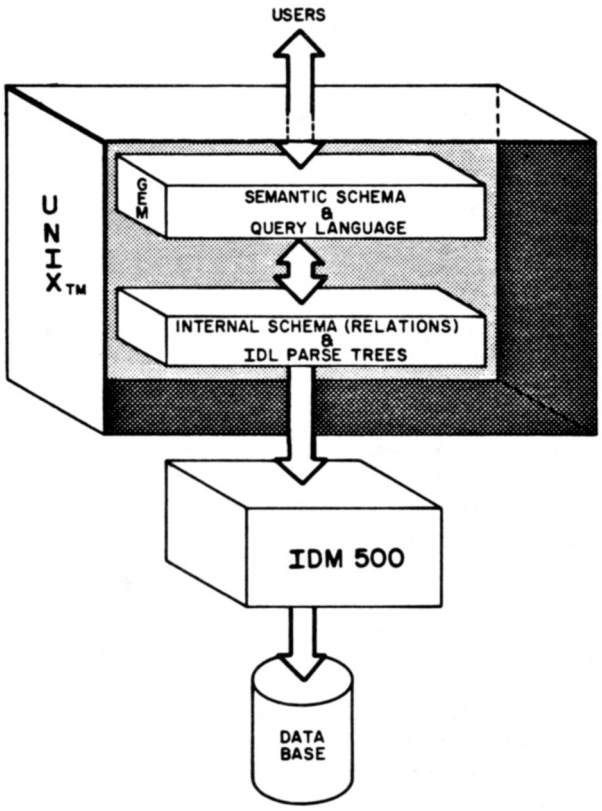

Figure 1. Architecture of the system.

Figure 1 shows the general architecture of our system. There are basically two levels of data definition and manipulation. The external level, consisting of GEM semantic schemas and data manipulation commands, supplies the interface to users [1]. The internal level consisting of a relational storage schema and of IDM's commands already in their IDL parse-tree form directly processable by the back-end.[2] The translation between the upper and the lower level is handled by the host, and users communicate with the back-end machine through the host-resident UNIX system. There, data definitions and queries are parsed and translated into corresponding parse-trees, which are then passed to the back-end machine. Finally the processed results are returned to the user via the host system.

The main focus of this paper is on the translation of GEM semantic schemas and data manipulation commands into equivalent constructs processable by the relational back-end. We begin with a brief discussion of GEM's DDL and DML and then proceed with the specifics of the translation.

2. The GEM Data Model

Figure 2 gives the GEM schema for an example adapted from [LaPi].

```
SUPPLIER (Company: c, Address: c)  key(Company);
DEPT (Dname: c, Floor: i2)  key(Dname) ;
ITEM (Name: c, Type: c null: "-", Colors: {c})  key(Name);
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 2. The GEM schema describing the following database:

SUPPLIER:	the names and addresses of supplier companies.
DEPT:	for each department its name and the floor where it is located.
ITEM:	for each item, its name, its type, and a set of colors
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.

Figure 3 gives the syntax of the GEM DDL. (GEM's data types — not defined in Figure 3 — 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.) A GEM schema consists of a set of uniquely named entities, and one or more keys specified for each entity. Thus, if we let relations correspond to entities, the relational model and GEM basically share Productions 1 and 2. The big difference is in the declaration of attributes. Whereas a relational system would be limited to the pattern used to generate SUPPLIER and DEPT,

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

GEM provides various extensions. The first is the option of adding a <null spec> to specify that an attribute can be null, by either designating a value from the attribute's domain to serve in this role, or by asking the system to handle this problem (at the cost of additional storage). For instance in ITEM the user let a hyphen denote a null value for the attribute Type.

Also, GEM supports set-valued attributes (Productions 4 and 6); e.g., "Colors: {c}" in ITEM defines an attribute having as value a set of (zero or more) data-items of type c (character string).

The relationship of aggregation is supported in GEM via reference attributes, which have entity occurrences as their values. Thus the attribute Dept in SALES has its value an occurrence of the entity DEPT, and Item has an occurrence of

1. A third possible representation level, consisting of DML-embedded views, such as those of INGRES, is not considered here.
2. IDL is the QUEL-like DML supported by the IDM 500.

ITEM as its value. Null values can also be allowed for reference attributes (Production 7).

1. <Schema>: { <Entity> ; }
2. <Entity>: <EntName>
   (<AttrSpec> { , <AttrSpec> }) { Key }
3. <AttrSpec>: <SimpleAttr> | <SetAttr>
   | <RefAttr> | <Generalization sublist>
4. <SimpleAttr>: <DataAttr> [<null spec>]
5. <DataAttr>: <AttrName> ':' <DataType>
6. <null spec>: **null**':' <datavalue> | **null**':' **system**
7. <RefAttr>: <AttrName>':' <EntName> **[null allowed]**
8. <SetAttr>: <AttrName> ':' '{' <DataType> '}'
9. <Generalization sublist>:
   '[' <Entity> {, <Entity>} , <Entity> ']' |
   '[' <Entity> {, <Entity> } , **others** ']'

Figure 3. The syntax of GEM DDL. (Braces denote zero or more occurrences of the enclosed substring while brackets denote zero or one occurrence of the same; braces, brackets and colons enclosed in semiquotes denote themselves [IDM].)

Finally (Production 9), an <AttrSpec> can be a generalization sublist that specifies 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. For instance in EMP we find two generalization sublists. The first 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 describes the marital status of an employee where one can either be an employee-married (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, producing a generalization hierarchy called an *entity family*. Within an entity family, attribute names must be unique

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. Name and Spouse are the two keys for the EMP family. However, the uniqueness constraint is waived for key values that are partially or totally null; thus the effect is the same as if Spouse were declared a key for the subentity EMARRIED.

DBMS users' prevailing view of schemas and data is graphical (e.g., tabular, hierarchical or network-like), rather than syntactic. Therefore, we want a graphical — preferably a tabular — representation for our schemas. A simple solution to this problem is shown in Figure 4.

There is an obvious correspondence between the in-line schema in Figure 2 and its pictorial representation in Figure 4; 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.[3] We found this representation most useful for formulating queries and update requests; also it defines the skeletons and headings of the row-columns tables used to present query results to users.

3. One could also give this representation a more pronounced E-R or functional flavor by representing entities references by pointed arrows.

3. The GEM Query Language

GEM is designed to be a generalization of QUEL [INGR]. Whenever the underlying schema is strictly relational (i.e., all attributes are <DataAttr>) GEM reduces to QUEL with which we assume that our readers are already familiar. However, GEM allows entity names to be used as range variables without explicit declarations. 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.

In the syntactic context of the **retrieve** and **where** clauses, the previously undeclared identifier DEPT is interpreted by default as a range variable over the entity DEPT.

This option of omitting explicit 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.

3.1 Aggregation

A *reference* attribute, as seen by a GEM user, has an entity occurrence as its value. 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 occurrence of type DEPT and SALES.Item is one of type ITEM. No entity occurrence can be printed as such. Thus, the statement

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

Example 3. A syntactically incorrect query.

is incorrect in GEM, as it would be in QUEL. Since S.Dept denotes an entity occurrence (of type DEPT), the following statement is also incorrect:

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

Example 4. Another incorrect query.

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

retrieve (SALES.Vol)

Example 5. 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 appending ".Floor" to it. Thus,

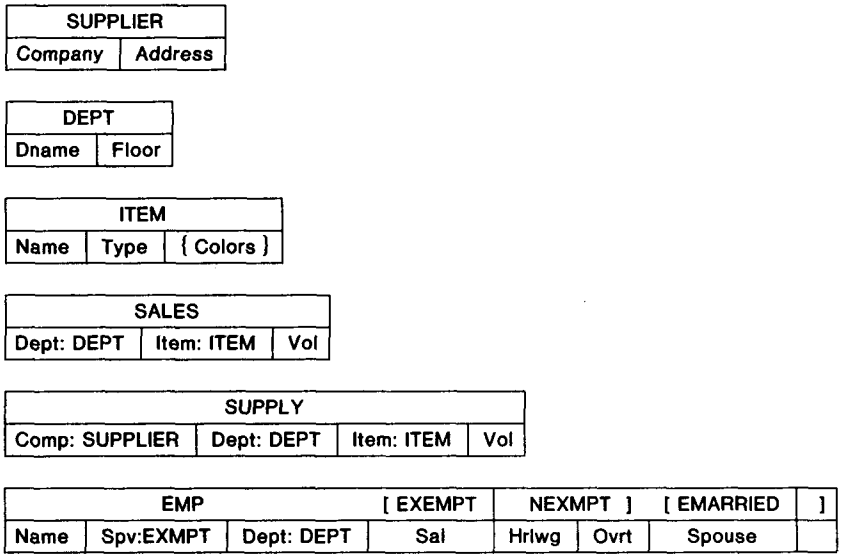

Figure 4. A graphical representation of the GEM schema of Figure 2.

retrieve (SALES.Dept.Floor)
where SALES.Item.Type="sport"

Example 6. Find all the floors where departments selling items of type sport are located.

will print all the floors where departments that sell sport items are located. The convenience and naturalness of this extension of the dot notation, also used in [Taxis], cannot be overemphasized; as illustrated by numerous examples [Zani2] it supplies a very convenient and natural construct that eliminates the need for complex join statements in most queries (much in the same way as DAPLEX does [Ship]). For instance, consider the classic example:

retrieve (EMP.Name)
where EMP.Sal > EMP.Spv.Sal

Example 7. Find all employees that make more than their supervisor.

Since the dot notation can be viewed as denoting a functional composition, 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**.[4] For instance to find all the employees working in the same department as J. Black we can write:

range of E **is** EMP
retrieve (EMP.Name)
where EMP.Dept **is** E.Dept **and** E.Name = "J. Black"

Example 8. Using an entity join to find all persons working in the same department as J. Black.

4. The operator **isnot** is used to test that two objects are not identical. Valued-based comparison operators, such as =, !=, >, >=, <, and <=, are not applicable to entity occurrences.

3.2 Generalization

Subentities' names can be used in two basic ways. The first is as default 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 9. Find the name and salary of each married employee.

or simply,

retrieve (EMARRIED.Name, EMARRIED.Sal)

Example 10. Same as in example 9.

Thus any attribute within an entity family can be applied to an occurrence ranging over any subtype in the family (without ambiguity since each name is unique within the family).

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 11. Same as Examples 9 and 10.

(Retrieve the name and salary of each employee who is an employee-married.) For each employee who is married but non-exempt the last three queries return his or her name and a null salary. Thus, they are different from

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

Example 12. 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 13. Find all employees that are exempt or married.

retrieves the names of all employees that are exempt or married.

4. Mapping the GEM Schemas into Internal Relations

An important decision, affecting the complexity of query translation mechanisms and the efficiency of query execution and storage utilization, is the mapping of an entity family into internal relations. For instance in [SmSm], a family is partitioned vertically and the fragments are linked together using the Codasyl set-coupling mechanism, while the Local Database Manager described in [ChDa] supports various options including vertical and horizontal partitioning, and physical clustering of fragments. However, additional joins are needed for supporting queries in the presence of vertical fragmentation and set differences and intersections are needed with horizontal fragmentation (direct control of pointers and storage structures can minimize the cost of these operations [ChDa], but this capability is not at hand here since we build on top of a relational system). Also we want to keep the query mapping as simple as possible, and take direct advantage of the IDM's indexing and clustering capabilities. All these considerations lead to an implementation where a whole entity family is mapped into one internal relation. The main drawback of this solution is the additional storage often required, although this is minimized with a technique described later.

Given a family with E as its top entity, the internal relation storing all family entities will be denoted by !E. Thus !EMP is the internal relation corresponding to EMP and all its subentities (namely EXMPT, NEXMPT and EMARRIED).

4.1 Surrogates and Reference Attributes

Each entity occurrence is uniquely identified by the value of its *surrogate*, that will be denoted by the symbol "#". Reference attributes are implemented by linking entity occurrences via their surrogate values. Thus in Figure 4, the reference attributes Dept and Item in SALES are respectively implemented by the integer attributes Dept# and Item# that hold surrogate values of DEPT and ITEM.

Surrogates are inaccessible to users, stored as integers in the internal relations, and maintained by the system by means of the following dictionary table:

SURRGT (Entity: c, Count: i4)

Counts for an entity are incremented when tuples are initially loaded, or when new tuples are appended to relations. Thus, upon the addition of tuple t to entity E, t.# is assigned the value of SURRGT.Count where SURRGT.Entity = "!E".

4.2 Null Values

Every simple attribute or a reference attribute can be set to null if the user allows this option in the schema. Null values for reference attributes are always internally represented as zeros. For simple attributes however, the user has the option of either specifying an internal representation for nulls — e.g., zero could be used to represent a numeric null — or using a **system** specification. Therefore, for each attribute E.A which is null with the **system** option, a flag-attribute, A%, is added, such that when E.A = **null** then E.A% = 1 and the value of E.A is ignored. The value of E.A is read if E.A% = 0.

4.3 Set-valued attributes

Since only normalized relations are supported by the back-end, we store set-valued attributes in separate user-invisible relations, where each set member is linked to its set-owner tuple, by pointing to its surrogate value. In our sample schema the Colors attribute of entity ITEM is stored as

!ITEM_Colors (Ref#: i4, Value: c)

where the column Value holds the color information, and Ref# holds values of surrogates of !ITEM. The internal schema for the example of Figure 2 is given in Figure 5.

4.4 Representation of families and subentities

For each generalization sublist a discriminant field Dj%, invisible to users, is included in the underlying relation. For instance, for a tuple t of EMP, t.D1% = 1 denotes an EXMPT and t.D1% = 2 denotes NEXMPT. Moreover, t.D2% = 1 denotes an EMARRIED and t.D2% = 0 corresponds to **others**. If an employee is not married, then the value Spouse entered in !EMP becomes immaterial, and so we set it to save storage. Since this is a character string attribute, we set it to a blank and store it as a compressed data type so that all leading blanks are eliminated [IDM]. Numeric attributes in inapplicable subentities are instead set to the value zero and stored as compressed data types to remove all leading zeros [IDM].

5. Mapping of GEM into QUEL

We will now describe the mapping of default variables, entity joins, and implicit joins into equivalent expressions of QUEL. Here we use QUEL for clarity of exposition, but in reality we map them into IDL query trees [Tsur].

The translation of Example 2 illustrates the treatment of default variables.

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

Example 15. The translation of the query in Example 2.

5.1 Entity Joins

Identity tests on entity occurrences, specified by **is** or **isnot**, are respectively translated into equality and inequality tests on their surrogates. Thus the query of Example 8 is translated as follows:

range of E **is** !EMP
range of EMP **is** !EMP
retrieve (EMP.Name)
where EMP.Dept# = E.Dept# **and** E.Name = "J. Black"

Example 16. The translation of Example 8.

5.2 Variables ranging over subentities

Variables ranging over subentities are translated into range variables over the corresponding internal relation and a condition on the pertinent discriminant field. For instance, since D2% =1 in !EMP denotes EMARRIED, Example 9 is translated into

range of e **is** !EMP
retrieve (e.Name, e.Sal)
where e.D2% = 1

Example 17. Translation of Example 9.

Default range variables are treated in the same way. For instance Example 10 is translated into

```
!SUPPLIER (#: i4, Company: c, Address: c)
!DEPT (#: i4, Dname: c, Floor: i1)
!ITEM (#: i4,  Name: c, Type: c)                    !ITEM_Colors (Ref#: i4, Value: c)
!SALES (#: i4,  Dept#: i4, Item#: i4, Vol: i2)
!SUPPLY (#: i4, Comp#: i4, Dept#: i4, Item#: i4, Vol: i2) ;
!EMP (#: i4, Name: c, Spv#: i4, Dept: i4, D1%: i1, Sal: i4, Hrlwg: i4, Ovrt: i4, D2%: i1, Spouse: i4);
```

Figure 5. The internal schema corresponding to the GEM schema of Figure 2.

```
range of EMARRIED is !EMP
retrieve (EMARRIED.Name, EMARRIED.Sal)
where EMARRIED.D2% = 1
```

Example 18. Translation of Example 10.

Consider now the query in Example 11 involving an entity join of EMP with EMARRIED. The translation

```
range of  EMP is !EMP
range of EMARRIED is !EMP
retrieve (EMP.Name, EMP.Sal)
where EXMPT.D2%=1 and
EMP.# = EMARRIED.#
```

Example 19. A translation for query of Example 11.

is correct, but not very efficient considering that both entities EMP and EMARRIED are implemented by the same relation !EMP, and therefore both variables in Example 19 denote the same tuple. Therefore, when confronted with an entity join, our translator checks whether this involves subenties from the same family. When so, one variable name is simply substituted for the other in all occurrences, and so unnecessary range declarations and surrogate equality tests are eliminated. Thus the previous query is replaced by

```
range of  EMP is !EMP
retrieve (EMP.Name, EMP.Sal)
where EMP.D2%=1
```

Example 20. An optimized translation for Example 11.

Likewise, the translation of Example 13 is:

```
range of EMP is !EMP
retrieve (EMP.Name) where
EMP.D1%=1 or EMP.D2%=1
```

Example 21. The translation of Example 13.

5.3 Functional Joins

Implicit functional joins are translated into explicit ones by the introduction of additional range variables. For instance,

```
retrieve (EMP.Dept.Floor)
where EMP.Name = "T. Green"
```

Example 22. At which floor is T. Green's department located?

is translated into the following query (D is a unique name generated by the translator):

```
range of EMP is !EMP
range of D is !DEPT
retrieve ( D.Floor)
where EMP.Dept# = D.# and EMP.Name = "T. Green"
```

Example 23. Translation of Example 22.

The translation of Example 7 is (E is a unique name generated by the translator):

```
range of EMP is !EMP
range of E is !EMP
retrieve (EMP.Name)
where EMP.Sal > E.Sal
and EMP.Spv# = E.#
```

Example 24. The translation of Example 7.

In the general case, a functional join chain of the form

$$\text{retrieve } (E.A_0.A_1.\cdots.A_n)$$

will be translated as:

$$\begin{aligned}
&\text{range of } E_1 \text{ is } \cdots\\
&\quad\vdots\\
&\text{range of } E_n \text{ is } \cdots\\
&\text{retrieve } (E_n.A_n)\\
&\text{where } E.A_0\# = E_1.\#\\
&\quad \text{and } E_1.A_1\# = E_2.\#\\
&\quad\vdots\\
&\quad \text{and } E_{n-1}.A_{n-1}\# = E_n.\#
\end{aligned}$$

The proper range for $E_1, \ldots, E_n$ is derived from the schema description stored in the data dictionary. During this process, additional checks are performed to ensure that, for each $0 \leqslant j < n$, A_j is a reference attribute of E_j, and A_n is either a simple or a set-valued attribute of A_n.

6. Null Values

Our treatment of null values uses three-valued logic as in [Codd2], but it is based on a new interpretation of nulls, the *no-information* interpretation[5], that along with a new treatment of sets and aggregate operations eliminates the logical problems besetting Codd's approach [Zani1]. Three-valued logic is needed to handle expressions such as

```
ITEM.Type != "sport"

not (ITEM.Type = "sport")
```

Figure 6. Two logically equivalent conditions.

For an ITEM occurrence where the Type attribute is null, one may try to evaluate all comparisons to FALSE; then the first condition in Figure 6 would evaluate to FALSE and the second to TRUE, a contradiction. However, consider the three-valued logic approach; all comparisons where one or both operands are null evaluate to a (logical) **null**.

---

5. Under this interpretation the system treats a null as an information vacuum — i.e., as a placeholder for value that perhaps doesn't exist or is otherwise unknown. In this respect our approach differs from [Codd2] where a value is assumed to exist although it is unknow.

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 7. Three-valued logic tables.

Then, Boolean expressions of such terms are evaluated according to the three-valued logic tables of Figure 7.

Therefore, both ITEM.Type != "sport" and ITEM.Type = "sport" evaluate to a logical **null** and, according to Figure 7, so does the negation of the latter. Thus three-valued logic allows a consistent truth-functional evaluation for expressions involving negations and null values. To answer a query, the systems selects all tuples for which the **where** clause evaluates to TRUE; tuples that yield FALSE or the logical **null** are discarded [6].

While three-valued logic would be simple to implement in a system that supports a general-purpose programming language, our IDM 500 only supports the standard two-valued-logic version of QUEL. To overcome this problem, we emulate three-valued logic by two-valued logic via query reformulation. Thus, for each GEM query Q we generate an equivalent QUEL query Q', such that Q' evaluated in two-valued logic produces the same result as Q evaluated in three-valued logic.

To accomplish this, we parse the **where** clause of a query and then transform only its comparison terms. Comparison terms have the form

$$t.A\ \theta\ k$$

or,

$$t.A\ \theta\ r.B$$

where, r and t are range variables, k is a constant, and θ is one of the comparison operators =, !=, <, <=, >, >=. The transformation begins by counting the number of negations between the root of the parse tree to the term; if this number is odd the term is *negative*, otherwise it is *positive*. Then, positive terms in the parse tree are respectively replaced by

(t.A isnot null and t.A θ k)

and

(t.A isnot null and t.B isnot null and t.A θ t.B).

Negative terms are respectively replaced by

(t.A is null or t.A θ k)

and

(t.A is null or t.B is null or t.A θ t.B).

(Naturally, if t.A or r.B are not allowed to be null according to the schema's declarations, these transformations will either be simplified or omitted.) Finally, we transform each condition such as "t.A **is null**" or "t.A **isnot null**" by taking into account the internal representation of nulls for A. Thus if in Production 6, A was declared **null**: <datavalue>, these two map respectively into t.A = <datavalue> and t.A != <datavalue>. Otherwise the **system** option was chosen and these two map respectively into t.A% != 0 and t.A% = 0.

7. Set-Valued Attributes and Set Operations

The availability of set-valued attributes adds to the conciseness and expressivity of GEM schemas and queries [Zani2]. However, set valued attributes pose some non-trivial implementation problems. After investigating various approaches we have chosen to implement each set-valued attribute by an additional user-invisible internal relation where a tuple stores a set member with the surrogate value of the parent tuple. For example, in our sample schema the Colors attribute of entity ITEM and its parent relation will be stored as the pair:

!ITEM (#, Name, Type) !ITEM_Colors (Ref#, Value)

In GEM, 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 It **is** ITEM
retrieve (It.Name, It.Type, Tot=**count**(It.Colors))

Example 25. Using a set-valued attribute.

Then, such a query is mapped into the following (C is a unique name generated by the translator):

range of It **is** !ITEM
range of C **is** !ITEM_Colors
retrieve (It.Name, It.Type, Tot=
count (C.Value **by** It.# **where** C.Ref# = It.#))

Example 26. The translation of the previous query.

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

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

These set-comparison operators and the aggregate operations of **count** and **any** can also be applied to sets of entity occurrences (the other aggregate operators cannot). Thus the query, "Find the items supplied by every department on the 2nd floor," can be expressed in GEM as follows (a set is denoted by the enclosing braces):

retrieve (SALES.Item.Name)
where {DEPT **where** DEPT.Floor = 2} <=
{SALES.Dept **by** SALES.Item}

Example 27. Find the items supplied by every department at the 2nd floor.

To translate this query we map operations on entity occurrences into operations on their surrogate values. Then we translate subset relationships into equivalent aggregate expressions that can be evaluated with reasonable efficiency. For instance the previous query is translated as follows ("It" is a new variable

6. As shown in [Zani3] this approach is also capable of producing the MAYBE-answers (consisting of all objects for which the **where** clause does not evaluate to FALSE) to queries [Codd2], provided that the predicates "t.A **is null**" and "t.B **isnot null**" are allowed in qualification expressions; GEM includes such predicates [Zani2].

generated by the translator):

```
range of SALES is !SALES
range of DEPT is !DEPT
range of It is !ITEM
retrieve (It.Name)
where count ( DEPT.# where DEPT.Floor=2) =
      count ( DEPT.# by SALES.Item# where
            DEPT.Floor = 2 and DEPT.# = SALES.Dept#)
        and SALES.Item# = It.#
```

Example 28. The translation of the previous query.

These two queries are equivalent since R is a subset of S if and only if the cardinality of R ∩ S is equal to the cardinality of R. More generally the pattern:

$$\{x_1 \text{ by } y_1 \text{ where } z_1\} <= \{x_2 \text{ by } y_2 \text{ where } z_2\}$$

is implemented as follows:

$$\text{count } (x_1 \text{ by } y_1 \text{ where } z_1) =$$
$$\text{count } (x_1 \text{ by } y_1, y_2 \text{ where } z_1 \text{ and } z_2 \text{ and } x_1 = x_2 \text{ and } y_1 = y_2).$$

The remaining set relationships are implemented in a similar fashion, except for the membership test **in**. A membership test, say x **in** S, is implemented by checking that the intersection {x} ∩ S is not empty. This test can then be performed efficiently using the aggregate operator **any** that returns zero if a set is empty and one otherwise.

In the presence of null values, the set operators must be properly extended. A comprehensive solution of this interesting problem is presented in [Zani1]; for the specific case at hand (sets of values rather than sets of tuples), it 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 subset relationships.

8. Updates

GEM supports QUEL's standard style of updates, via the three commands, **append to**, **replace**, and **delete**, which generalize the corresponding relational operators in a natural fashion.

8.1 Append

The first example involves inserting an occurrence in an entity that contains only simple attributes:

```
append to DEPT (Dname = "toys", Floor =2)
```

Example 29. Adding the "toys" department at the second floor.

This is translated into (the IDL parse-tre representation of):

```
append to !DEPT(#=NEXT(DEPT), Dname="toys", Floor=2)
```

Example 30. The translation of Example 29.

The function NEXT generates a new surrogate value by looking up in SURRGT the current counter value (for DEPT), then incrementing it by one. The relationships of aggregation and generalization are handled in a natural fashion, as illustrated by the following example.

```
append to NEXMPT ( Name = "T.Jones", Spv = EXMPT,
        Dept = DEPT, Hrlwg = 5.0, Ovrt= 1.8)
where EXMPT.Name = "F.Green" and DEPT.Dname = "toys"
```

Example 31. T. Jones is hired in the toys department, under F. Green.

This request is translated into

```
range of DEPT is !DEPT
range of EXMPT is !EMP
append to !EMP(# =NEXT(!EMP), Name = "T. Jones",
      Spv# = EXMPT.#, Dept# = DEPT.#, D1% = 2,
      Hrlwg = 5.0, Ovrt =1.8, D2% = 0)
where EXMPT.Name = "F. Green"
and DEPT.Dname = "toys" and EXMPT.D1% =1
```

Example 32. The translation of Example 31.

In parsing Example 31, EXMPT and DEPT were interpreted as default variables and given the respective ranges !EMP and !DEPT; since EXMPT is a subentity of EMP the condition EXMPT.D1%=1 was also added to the **where** clause. However NEXMPT, since it follows **append to**, was syntactically interpreted as an entity name, not as a range variable; thus it was translated into the relation name !EMP and the assignment D1% = 2.

Example 31 involves an explicit assignment to a subentity; implicit assignments are also supported in GEM. For instance the result of the previous query does not change if we replace "NEXMPT" by "EMP" since the assignment of a value to Hrlwg and Ovrt would cause the entity to be assigned to the NEXMPT subtype, anyway — a case of redundant but consistent assignment. Symmetrically the absence of an assignment for Spouse results in the entity being classified as **others** than EMARRIED (thus D2% = 0). The translator deduces implicit assignments and checks for consistency by consulting the dictionary tables containing the schema description.

An interesting problem arises when several tuples are added at once. Take for example the following request:

```
append to SALES ( Dept= DEPT, Item= ITEM, Vol = 100)
where ITEM.Name = "catalog"
```

Example 33. Send 100 catalogs to each department.

Conceptually, this can be translated as follows:

```
range of  DEPT is !DEPT
range of ITEM is !ITEM
append to !SALES(#=NEXT(!SALES), Dept#= DEPT.#,
                    Item#=ITEM.#, Vol=100)
where ITEM.Name = "catalog"
```

Example 34. The translation of Example 33.

Here we want NEXT to generate a new number for each new tuple appended to !SALES. The simplest way to accomplish this is to increase a counter by one, for each tuple appended. Unfortunately even this simple operation cannot be requested from the IDM machine (as it exceeds the power of the so-called complete relational calculus or algebra). Instead, we have to bring each target tuple into the host, assign a unique value to its surrogate, and finally return it to the back-end and append it to !SALES.

8.2 Replace

Updating simple and reference attributes is easy in GEM. For instance, to transfer a certain type of item from one department to another, one only needs to say:

```
replace  SALES (Dept = DEPT)
where SALES.Item.Name = "sport-clothes"
and DEPT.Dname = "sport"
```

Example 35. Reassigning sport clothes to the sport department.

The translation of this request proceeds along the lines of query translations previously discussed, yielding ("It" is a unique name generated by the translator):

```
range of SALES is !SALES
range of DEPT is !DEPT
range of It is !!TEM
replace SALES (Dept# = DEPT.#)
where SALES.Item# = It.# and It.Name ="sport-clothes"
and DEPT.Dname="sport"
```

Example 36. The translation of Example 35.

The ease of use of GEM is well illustrated by a comparison of Example 35 with Example 36 that is basically what a user of INGRES, using a relational schema, would have to write to express an equivalent request.

8.3 Delete

Referential integrity constraints demand that all the reference to an entity occurrence must be set to null (if this is allowed) before the occurrence can be eliminated. For instance, since null is allowed for the Spv attribute in EMP, the request:

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

Example 37. Remove employee T. Green.

will result in the removal of T. Green's tuple from the database and in the setting to null of the Spv references for employees working for T. Green. However, since the reference attributes Item in SALES and Item in SUPPLY point to ITEM and null is dissallowed for both, the update

```
delete ITEM
where ITEM.Name = "soap-dish"
```

Example 38. Dropping the item soap-dish from the stock.

will be executed only if no SALES or SUPPLY record refers to this "soap-dish" ITEM; otherwise the update will be rejected and an error-message generated.

Among the alternative solutions considered for maintaining the referential integrity constraint, one consists of linking together the referred tuple with the referring ones in the style of Codasyl's owner-coupled set implementations. A second consists in keeping a reference count in each tuple referred by others [ChDa]. Both approaches require some additional operations, not only upon deletions, but also upon executions of **replace** and **append**. In the end, we opted for a solution that keeps the query mapping simpler and confines all integrity maintenance operations within deletions. This solution employs the temporary relation !TEMP(No: i4) and the statements **begin transaction** and **end transaction**. Thus the query of Example 37 is translated as follows:

```
begin transaction
   range of EMP is !EMP
   retrieve into !TEMP(No = EMP.#)
   where EMP.Name = "T.Green"
   range of TEMP is !TEMP
   delete EMP where EMP.# = TEMP.No
   replace EMP (Spv# = 0) where EMP.# = TEMP.No
end transaction
```

Example 39. Translation of Example 37.

The translation of Example 38 is:

```
begin transaction
   range of ITEM is !!TEM
   retrieve into !TEMP (No = ITEM.#)
   where ITEM.Name = "soap-dish"
   range of SALES is !SALES
   range of SUPPLY is !SUPPLY
   range of TEMP is !TEMP
   delete ITEM where ITEM.# = TEMP.No
   and any(SALES.Item# = TEMP.No) = 0
   and any(SUPPLY.Item# = TEMP.No) = 0
end transaction
```

Example 40. Translation of Example 38.

Only when there is no reference pointing to !!TEM, the **any** functions in Example 40 evaluate to zero and the tuple is deleted. The IDM returns to the host a tally for the number of tuples inserted in !TEMP and those deleted from !!TEM. This is sufficient to determine whether the request was correct or it did not execute because the existence constraints were violated, in which case an error message is returned to the user.

9. Conclusion

In an attempt to demonstrate the feasibility of extending a relational DBMS into one supporting a semantic data model, we have presented a system that consists of a UNIX-based front-end that maps the GEM semantic data model and query language to an underlying IDM 500 relational database machine. By means of representative examples we have illustrated the nature of the mapping, underscoring both the feasibility and the limitations of our approach. Here, we would like to summarize the positive and negative lessons learned in the course of this research

A main positive conclusion is that relational query languages and interfaces are more robust than they are generally given credit for. GEM shows that it is possible to extend the relational approach to achieve (i) a modeling power that matches that of other semantic models, and (ii) a high-level and set-oriented DML that matches and in many ways surpasses (e.g., by allowing functional joins) the ease of use and power of relational languages. In a way, GEM adds a distinct Entity-Relationship flavor [Chen] to relational schemas, and the expressivity of functional languages to relational queries. It is also suitable for embedding database facilities in programming languages [Andr]. These conclusions, reinforced by the basic simplicity of the mapping from GEM to QUEL, demonstrate the feasibility and desirability of the evolutionary approach to semantic data models. This approach preserves compatibility with existing relational systems, since users who don't want the extra semantic features need not learn nor use them; for these users GEM reduces to QUEL.

It is too soon to evaluate the effectiveness of the specific architecture chosen here to implement this evolutionary approach, since this work is currently at an early implementation stage. But a few interesting lessons have already been learned. One is that it is easier to support entities, aggregation, generalization and null values than sets or set-valued attributes. Another is the pros and cons of using a commercial database machine. Our work has benefited a great deal from our decision of basing our internal representation on an IDM_500-compatible interface. The availability of ready-made software is the most obvious advantage. Moreover, the availability of a high-level well-documented interface allowed us to concentrate on conceptual issues, rather than on implementation details, and to complete and document our design with less effort. The

increased productivity claimed in [Codd1] became real to us.

On the negative side, we found the IDM 500 interface lacking in functionality and flexibility since all it offers is a relational DDL and DML. With such an interface, complex database operators are expressed easily, but even the simplest procedural function becomes impossible to compute. For instance, it is impossible to count the tuples of a relation assigning a sequence number to each, without streaming them through the host; also, conditional statements or procedure calls are not provided. These limitations forced us to move much computation to the host, thus reducing the benefits of off-loading, and creating unnecessary communication costs. We also found that certain improvements are desirable with respect to the storage organization. Thus, binary data types and bit operations would be desirable (at the present, bytes are the smallest unit of data and no boolean operator is supported); also, internal support for null values would be very useful. Finally, one needs better control on the placement of stored data (e.g., to allow clustering, or value-dependent placement). All these features are important and apt to influence the performance of the system; however, the problem of providing a measure of extensibility, e.g. via some procedural capabilities, is even more critical since it constitutes the *sine qua non* for relational machines to be used as general-purpose back-end systems, as demanded by many applications. These enhancements should provide a focus for further work.

Acknowledgments

The authors are grateful to J. E. Andrade for helpful discussions and comments.

References

[Adplx] Smith, J.M., S. Fox, T. Landers, "ADAPLEX: The integration of the DAPLEX Language with the Ada Programming Language," Technical Report, Computer Corporation of America, 1982.

[Andr] Andrade, J. M. "Genus: A Programming Language for the Design of Database Applications," Internal Memorandum, Bell Laboratories, 1982.

[ANSI] Tsichritzis, D.C. and A. Klug (eds.) "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems," *Information Systems,* Vol. 3, pp. 173-191, 1978.

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

[Catt] Cattell, R.G.G., "Relationship-Entity-Datum Data Model," Technical Report CSL 83-4, Xerox Palo Alto Research Center, 1983.

[ChDa] Chan, A., Danberg, S., Fox, S., Lin, W.K., Nori, A., Ries, D., "Storage and Access Structures to Support a Semantic Data Model," *Proc. Very Large Data Base Conference,* Mexico City, 1982, pp. 122-130.

[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.

[DeHa] DeWitt D.J. and P. Hawthorn, "A Performance Evaluation of Database Machine Architectures," *7th Int. Conf. Very Large Data Bases,* pp. 199-215, 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 Data Language DAPLEX," *ACM Trans. Data Base Syst.,* 6,1, pp. 140-173, 1982.

[SiKe] Sibley, E.H., and Kerschberg, L. "Data Architecture and Data Model Considerations," *Proc. AFIPS Nat. Computer Conf.,* Dallas, Tex, June 1977, pp. 85-86.

[Stone] Stonebraker, M., "Operating System Support for Database Management," *Comm ACM,* 24:7, 1981, pp. 412-417.

[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.

[Taxis] Mylopoulos, J., P.A. Bernstein and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications," *ACM Trans. Database Systems,* 5, 2, pp. 185-207, June 1980.

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

[Zani1] Zaniolo, C., "Database Relations with Null Values," *Journal of Computer and System Sciences,* 28, Feb. 1984 (abstract in the *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,* 1982).

[Zani2] Zaniolo, C., "The Database Language GEM," *ACM SIGMOD Conference,* May 1983.

[Zani3] Zaniolo, C., "A Formal Treatment of Nonexistent Values in Database Relations," manuscript submitted for publication, 1983.