G-WHIZ*, a Visual Interface for the Functional Model with Recursion

Sandra Heiler and Arnon Rosenthal
Computer Corporation of America

Abstract

G-WHIZ is a QBE-style interface for the functional data model, with extensions that support recursively defined structures such as part hierarchies. Explicit joins are rarely needed because set-valued and entity-valued functions of the functional model are supported. The recursive facilities are integrated with the rest of the language. G-WHIZ currently is being implemented as the user interface to a CAD/CAM DBMS.

1. Introduction

G-WHIZ is a screen-oriented language for the functional data model [Sh]. It currently is being implemented as the main interface to CCDBMS, a CAD/CAM DBMS that must handle complex interrelationships among the stored data. Its style comes from Query-By-Example (QBE) [Zloof, Date].

---

*Grids With Hierarchies, Imitating Zloof

Authors' addresses:

Sandra Heiler
Computer Corporation of America
1800 Diagonal Road
Alexandria, VA 22314
(703) 836-5200
heiler@cca or decvax!cca!heiler

Arnon Rosenthal
Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142
(617) 492-8860
arnie@cca or decvax!cca!arnie

This paper concentrates on the two main areas of G-WHIZ that significantly extend QBE:

1. Use of the functional model, which simplifies complex queries (explicit joins are rare, and example elements nonexistent)

2. Constructs for defining and querying recursively defined structures

2. The Functional Data Model

The basic constructs of the functional model are the entity and the function, which model conceptual objects and their properties. An entity type corresponds to a base relation, a function to an attribute. A function may be single-valued or set-valued (have zero, one, or many values for each entity), and its range may be simple (a string or numeric type) or another entity type.

Relationships between entities are modeled as entity-valued functions. For example, given two types of entities, PARTs and DRAWINGs, the drawing of a part can be defined as an entity-valued function DRAWING(PART). The inverse function defines the relationship in the reverse direction (i.e., PART_IN_DRAWING(DRAWING) yields the PARTs represented in DRAWING).

Entity-valued functions may be single-valued to represent one-to-one relationships or set-valued to represent one-to-many or many-to-many (with set-valued inverses) relationships. (Entity-valued functions may be implemented by storing their values as entity identifiers or they may be derived through uni-directional outer-joins.)

Functions of related entities can be considered derived functions of the base entity and appear in the same view as the base entity without an explicit join. In the above example, functions of DRAWING (which is a function of PART) are derived functions of PART. They can be represented by function composition (nesting). For example, the location of a drawing is a function of the part represented by the drawing LOCATION(DRAWING(PART)).

The functional model supports entity supertypes and subtypes (i.e., generalization hierarchies [SS])). For example, the PART entity type might be defined as a supertype of MADE_PART and PURCHASED_PART entity types as well as ELECTRICAL_PART and MECHANICAL_PART entity types. Such generalizations imply an associated inheritance of functions.

These constructs result in several important differences between the relational model and the functional model that are reflected in G-WHIZ [Man].

1. Since functions may take on entity values, functions from a related entity may be referenced by function composition without an explicit join. For example, to select PART entities based on the locations of their drawings, an explicit join of PARTs and DRAWINGs need not be specified; LOCATION(DRAWING(PART)) can be referenced directly. Further composition has the effect of further joins.(1)

2. Set-valued functions allow multiple values (including duplicates and null) for an entity. For example, the DRAWING function of PART entities may be defined as set-valued to indicate that several drawings describe the part without repeating the part information. The relational model requires a separate relation for each set-valued function.

3. Entity subtyping allows an entity to be several types at once, with the functions of the supertypes inherited by the subtypes. For example, a pump might be an ELECTRICAL_PART as well as a PURCHASED_PART and also automatically be a PART, indicating that it has all functions of both subtypes ELECTRICAL_PART and PURCHASED_PART and also the functions inherited from the parent type PART. The relational model requires separate entities for each type.

4. An entity-valued function represents an outer-join between entities of the base type and entities of the function type. For example, the DRAWING function of the PART entity can be thought of as a unidirectional outer join between PARTs and DRAWINGs where the value of the function is null for parts that have no drawings representing them. Though outer-joins have been added to the relational model, many relational languages do not support them.

(1) Other approaches to a join-free interface have been suggested. For example, in the Universal Relation approach [Mai] the system must decide what real-world objects are being referenced, and find a join path among them. Object identification and path selection is based on names and dependencies, rather than on explicit declarations of entities and functions.

3. Background

Like QBE, G-WHIZ is a two-dimensional interface, designed for simple terminals such as IBM 3270s. Operations and parameters are specified in a grid that looks like the rows and columns of a table. Table rows are equivalent to relational tuples or entities and columns are equivalent to attributes or functions of the entities. The grid removes much of the syntax burden from the user, allowing different parts of a complex query to be generated in whatever order is convenient. The facilities described in this section, except conditional operations and views, are identical to QBE.

Specifying Operations

Queries are specified by entering selection criteria (qualifications) in the columns of the functions they qualify. Projections are performed by deleting columns from the table. Operations I.(insert), U.(update), D.(delete), and P.(print or display) are provided to operate on entities (rows) or functions (columns).

To operate on entities or their functions, the user specifies the entity type name in the upper left corner of the grid; the system fills in the function names, and the user specifies the required operations in the columns of the grid. For example, suppose the user wants to print the values of all functions of PART entities where NAME(PART) is "wing".

```
user:
     PART  |
   --------+--------------------------------
           |

system:
     PART  | PART-NBR | NAME | COLOR | . . .
   --------+----------+------+-------+-------
           |          |      |       |

user:
     PART  | PART-NBR | NAME | COLOR |
  ---------+----------+------+-------+-------
      P.   |          | wing |       |
```

Combining Qualifiers

Qualifiers in a column are ORed and the resulting column specifications are ANDed. For example,

PART	PART_NBR	NAME	COLOR	...	COST
P.		wheel	grey		< 10
			blue		

displays values of all functions of PART entities where ((NAME is "wheel") AND (COLOR is "grey" OR "blue") AND (COST < 10)). Comparators =, <,>, <=, >=, and ~(not) may be specified in qualifiers. The = sign is understood if no comparator is specified. Complex qualifiers not fitting the pattern may be specified within a column by a Boolean expression using names of other functions of the entity. (On the rare occasions where alphanumeric literals conflict with function names in the current view, the literals are placed in quotes.)

Application of Operators

First, entities that satisfy the grid's qualification are selected. Then operators are applied. Operators specified in the entity name column affect all functions of the selected entities; operators specified in other columns affect the specified functions of the selected entities. For example,

PART	PART-NBR	NAME	COLOR	...
		rim	red U. blue	

selects entities where NAME is "rim" and COLOR is "red", and updates COLOR to "blue".

We extend QBE to allow multiple operations in a column. If an operator is specified by itself (i.e., without a qualifier), it applies to all values of that function in the selected entities. If it is specified next to a qualifier, a subselection is performed on the entities that satisfy the union of the qualifiers in that column and the operation is performed only on those that satisfy the associated qualifier. The "otherwise" qualifier is specified as "?". For example,

PART	PART_NBR	NAME	WIDTH
		wing	10 U. 10.1
			20 U. 20.1
			? U. WIDTH * 1.5

specifies that PART entities in which (NAME = "wing") AND (WIDTH = 10 OR 20 OR anything else) are to be selected. Then those in which WIDTH = 10 have that value changed to 10.1 and those in which WIDTH = 20 have that value changed to 20.1; all others have the WIDTH value changed to WIDTH * 1.5.

To insert a new row (entity), the user specifies the I. operator in the entity name column and the values of its functions as equalities in the function columns. The idea and syntax resemble equality qualifiers on any operation.

When display, update, and delete operations are specified, the system responds with the number of entities that were selected (shown in parentheses in the view name column). The user can then display values of the functions of the selected entities (before performing specified updates or deletions), confirm update or delete operations, or cancel the request.

Views

All access to data is through views. Each stored entity type has a view (with the same name) defined over it. The user first defines some stored entity types and their functions, much as tables are defined in QBE. New views are created by selecting and projecting on existing views, or extending them with derived or computed functions (including entity-valued functions to produce the equivalent of join views). A single entity type underlies each view -- the entity type that underlies the view from which it was derived.

The definition of a view includes:

1. The entity type and its functions
2. Selections
3. Projections
4. Formatting instructions, such as function display widths
5. Functions of entities referenced through entity-valued functions
6. Definitions of computed functions
7. In a hierarchical view, the successor function for the traversal and the beginning node(s) (described in section 9.1)

The name of a view can act as a qualifier in the entity name column of another view or of an entity-valued function. It represents the set of entities defined by the view.

4. G-WHIZ Screen Format

The G-WHIZ screen format is similar to that of QBE. An entity type (or view name) is specified in the upper left corner, the names of functions of the entity type are specified across the top row, and operations and selection qualifiers are specified in the rows and columns. G-WHIZ uses an asterisk to identify functions that participate in the primary key of the entity.

The functional model interface benefits from some minor enhancements to the QBE screen format. Set-valued functions are identified by a double underline. Entity-valued functions (which also may be set-valued) are marked by filler lines that precede and follow the function name in the grid segment, double filler lines if the function is both entity valued and set-valued. For example:

```
PART |*PART_NBR| NAME |COLOR|...|=DRAWING=
-----+---------+------+=====+---+=========
     |         |      |     |   |
```

indicates that PART_NBR is an identifying (key) function of PART, COLOR is set-valued, and DRAWING is both entity-valued and set-valued.

G-WHIZ displays a pop-up menu of commands and programmed function (PF) key meanings, to help the user remember which commands are relevant in the current context.

5. Set-Valued Functions

The relational model's simplicity is partly due to the fact that attributes are atomic. An unfortunate consequence is that to associate a set of values with a single entity, a join is necessary. The functional model avoids these joins by allowing a function value to be a set. (Many proposals have been made to add set-valued attributes to the relational model (e.g., [AB], [RKS]).)

This section shows how G-WHIZ extends the QBE-style interface to set-valued functions. The extension is consistent with constructs like conditional update from the basic interface.

When a set-valued function is qualified, entities are selected if any value of the set-valued function satisfies the qualifier. The qualifiers "-" (null) or "~-" (not null) are used to test whether the set is empty.

The insert (I.) column operator inserts a value into the set of values of the function for each selected entity. The display (P.), update (U.), and delete (D.) column operators apply to all values of the function, for each selected entity, unless further qualified.

If operators are specified in conjunction with qualifiers, the qualifier(s) are first used to select a set of entities. Then each operator is applied to the subset of those entities and the particular values of the set-valued function that satisfy its associated qualifier. (Results are indeterminate if qualifiers overlap.) For example, suppose that COLOR has been defined as set-valued:

```
 PART | NAME | . . . |   COLOR
------+------+-------+============
      | tail |       |red U. rouge
      |      |       |blue U. bleu
```

In this example, entities in which ((NAME = "tail") AND (any value of COLOR = "red" OR "blue")) are selected. In the selected entities, COLOR values "red" are changed to "rouge" and "blue" to "bleu".

When a new entity is inserted (I. row operator), multiple values may be listed in the column for each set-valued function.

The display operator (P.), displays each entity instance as a single row. If some function of that entity has a set with more than one value, the count of the set is displayed (in parentheses). To display the values of the entity's set-valued functions, the user moves the cursor to the appropriate row and presses the ZOOM key.

6. Entity-Valued Functions

Entity-valued functions eliminate the need for explicitly specifying joins. G-WHIZ incorporates these functions through the addition of a single operator, EXPAND.

In the functional model, a relationship between entities is represented by an entity-valued function of one entity; the inverse relationship is represented by a function of the other entity. For example, the relationship between drawings and parts is represented by the DRAWING function of PART and the inverse by a PART_SHOWN function of DRAWING.

When a view contains an entity-valued function, the user can include derived functions (i.e., functions of the related entities) in the view by positioning the cursor on the entity-valued function and pressing the EXPAND key. For example,

```
                                 |       expand
                                 V        key
PART | NAME | COLOR | ... |=DRAWING=
-----+------+=======+-----+=========
     |      |       |     |
```

which results in

```
                          |=====DRAWING========
PART |NAME| COLOR |...|  |DNBR|-LOCN-|PAGES
-----+----+=======+---+==+----+------+-----
     |    |       |   |  |    |      |
```

Now the user can select PART entities based on values of functions of their related drawings, and display functions of both PART and DRAWING, as shown below:

```
                              |=====DRAWING========
PART | NAME | COLOR |...|   |DNBR|-LOCN-|PAGES
-----+------+=======+---+==+----+------+-----
     |P.wing|       |   |  | P. |      | >4
```

The above example selects PARTs where NAME=wing and any drawing of the part has PAGES(DRAWING)>4 and displays the values of NAME and associated DNBRs of the drawings of the selected PARTs.

Multiple levels of entity-valued functions can be EXPANDed. For example, the location of a drawing (LOCN), which is shown as entity-valued, could be expanded to show its functions as derived functions of PART.

The EXPAND operation circumvents an awkward feature of the basic functional model. When referencing several functions of a related entity, it is awkward to repeatedly express the function composition. For example:

```
Retrieve (PAGES(DRAWING(PART)), DNBR(DRAWING(PART))
                where PAGES(DRAWING(PART)) <16 )
```

Updating through Views

G-WHIZ has simple (though limited) semantics for view update. Only entities of the type underlying the view can be inserted, deleted, or updated and only functions of the entity underlying the view can be inserted, deleted, or updated. Computed and derived (nested) functions are not updatable.

When insert, delete, or update operations are specified on entity-valued functions, they operate on the references to the related entity type in the entities of the primary type that underlies the view. They cannot insert or delete entities of the related type, or update functions of that type. For example,

```
                          |========DRAWING=======
PART |NAME| COLOR |...|    |DNBR|-LOCN-|PAGES
-----+----+=======+---+====+----+------+-----
     |pump|       |   | I. |845A|      |
```

inserts a reference to the DRAWING entity whose DNBR is 845A into the DRAWING function of the PART entity whose NAME is pump. (If no such DRAWING entity exists, the insert operation is rejected.) I., D., and U. operators cannot be specified in the expanded columns of the related entity type.

7. Computed Functions

7.1 Defining Simple Computed Functions

Computed functions may be included in a G-WHIZ view by inserting a column, naming it, and specifying its value as an equality in the inserted column, similar to the way values are specified when a new row is inserted. The equality may be a constant, an expression, or null (-). For example,

```
                                   |    insert
                                   V      key
 PART |NAME|...|HEIGHT |WIDTH|   AREA
------+----+---+-------+-----+--------------
      |    |   |       |     |HEIGHT * WIDTH
```

Other qualifiers may be combined with the equality in a boolean expression to specify conditional values. The notation below uses & to denote "where <condition>". The usage is consistent with notations for selecting the proper value for the added function, and for specifying values to be inserted. For example:

```
                      |       insert
                      V        key
...|              COLOR_CODE             |...
---+-------------------------------------+---
   | bright & (COLOR = red | yellow)     |
   | dark & (COLOR = blue|black|brown)   |
```

The newly-defined function normally is single valued. It will be set-valued if multiple equalities are specified or if the expression evaluates to a set. A computed function can be defined to contain a subset of values of another set-valued function. For example, the following specification defines RED_BLUE, which contains the subset of values of COLOR that are equal to red or blue. It is null for entities in which no value of COLOR is red or blue.

```
                              |     insert
                              V      key
PART |NAME| COLOR | ... |  RED_BLUE
-----+----+=======+-----+===========
     |    |       |     |COLOR & red
     |    |       |     |COLOR & blue
```

Arithmetic expressions and conditions in function definitions or qualifiers may be continued on the next line by ending a line with an arithmetic or logical operator or an open parenthesis.

7.2 Defining Entity-Valued Computed Functions

Entity-valued computed functions are defined by identifying the range (entity type or view name) of the new function and specifying the condition that determines the values of the new function. New entity-valued functions may be computed to capture a value-vased join condition, to define unions, or to subset an existing entity-valued function based on some qualification.

For example, suppose the user wants to define a new entity-valued function of PART, whose values will be the set of VENDORs that make that PART. He associates appropriate vendor information with PART entities by adding an entity-valued function, which he calls SOURCE in this case (he could as well call it VENDOR), to the PART view, defining its range as VENDOR, and specifying its value to be the set of VENDORs satisfying the join condition MADE_BY(PART) = COMPANY(VENDOR).

```
                             |    insert
                             V     key
PART|NAME|...|MADE_BY|...|=SOURCE=
----+----+---+-------+---+========
    |    |   |       |   | VENDOR
```

```
                                  |    expand
                                  V     key
                          |======SOURCE======
PART|NAME|...|MADE_BY|...|  |COMPANY|ADDRESS
----+----+---+-------+---+==+-------+-------
    |    |   |       |   |  |MADE_BY|
```

Each PART entity will be associated with SOURCEs (VENDOR entities) in which COMPANY(SOURCE) = MADE_BY(PART).

Function names among related entities may be qualified by their entity type to distinguish duplicates.

8. Recursively Defined Views

Current systems for bill-of-materials and other applications over recursively-defined "path" structures use applications code to navigate the database. G_WHIZ integrates facilities for processing such structures into the query language of the DBMS. The integrated architecture uses DBMS facilities to handle query language commands, query optimization, query execution (e.g., handling temporaries), and the user interface.

In G-WHIZ, a hierarchical view is a view specified recursively, as a rooted tree. We use the terms "hierarchical view," "hierarchy," and "recursively-defined view" synonymously. A hierarchical view H is defined over an existing view (or entity type) V by specifying:

1. The entities in H. Each entity in H corresponds to a node of the tree. The entities in H include the functions of the entities in the underlying type V, plus some recursively-defined functions. Each entity instance in H corresponds to some entity instance of V. The entities present and the values of the new functions are computed recursively, as described later.

2. A successor function. One of the functions of V is selected as the successor function (succ()) of the hierarchy H. succ() must be entity-valued, ranging over entities of the same type as V. It must be acyclic; that is, repeated applications of succ() should not return to the starting point. A hierarchy can be defined over any view that includes a successor function.

3. Root node(s). Some instance(s) of V must be selected as the beginning of the recursive traversal. For simplicity, our examples assume the traversal begins at a single entity, E0 in the underlying view V.

Beginning at E0, the hierarchy's nodes contains a node corresponding to E0, plus the hierarchies rooted at successors(E0). For a Part hierarchy for an airplane, H might consist of the airplane, and the hierarchies of each immediate successor of airplane. (Immediate successors of airplane might be left wing, right wing, fuselage, and tail). If entity E in V is the successor of two different entities that appear in the hierarchy, E (and the hierarchy below E) will appear below each of them. For example, a pump (and its decomposition) may appear several times in the PART hierarchy of an airplane. If multiple beginning nodes have been selected, the hierarchical view has a tree below each.

The full hierarchy can be enormous. Therefore G-WHIZ provides facilities for the user to form subsets of the hierarchy in several ways:

- Begin the traversal deeper in the hierarchy (e.g., form the hierarchy rooted at cockpit, not at airplane)

- Restrict the successor function so entire subtrees are skipped (e.g., consider only subtrees whose root entity is manufactured by XYZ Corp.)

- After the hierarchic view has been computed, restrict it using ordinary G-WHIZ entity selection.

A query language extension was necessary to handle recursive hierarchies because the path length in the hierarchy depends on the stored entity instances, not on the schema. No fixed number of expansions of the successor function can be guaranteed to produce all levels of the tree. Furthermore, each expansion would create new functions, while the hierarchy should have the same functions in all the nodes, aligned in columns to permit further selections.(2)

8.1 Defining a Recursive Hierarchy

To define a recursive hierarchy, the user displays a view and chooses a successor function by placing an H. in the function column. (The system may check whether the relationship really is acyclic.) Qualifiers preceded by B. are applied to select beginning node(s) to be used as the root(s) of traversals. If no beginning qualifiers are specified, entities that are not referenced by any successor function are used as the beginning nodes.

The grid below defines a hierarchical view over PART, using the SUBPART function as the successor function and beginning at the PART named "wing".

```
PART | NAME | COST | ... |=SUBPART=
-----+------+------+-----+=========
     |B.wing|      |     |   H.
```

Hierarchies are defined over views, not merely over stored entity types. Therefore, a successor function can be a computed function or it can be derived by composition. For example, the grids below define a computed function that includes only SUBPARTs whose cost is a significant fraction of the PART's cost. Then they define a hierarchical view beginning at the wing, using the computed successor function MAJOR_SUB.

```
                                      insert key
                                          |
                   |==SUBPART===|         V
PART|NAME|COST|...|  |NAME|COST|====MAJOR_SUB===
----+----+----+---+==+----+----+================
    |    |    |   |  |    |    |SUBPART & COST >
    |    |    |   |  |    |    | .1*COST(PART)
```

```
                    |==SUBPART===|
PART|NAME   |COST|...|  |NAME|COST|=MAJOR_SUB=
----+-------+----+---+==+----+----+===========
    |B.wing|    |   |  |    |    |     H.
```

Another example: suppose Circuits have multiple diagrams and that each diagram can include several component circuits. The specification below defines a hierarchy of circuits. The successor function is the derived function SUB_CIRCUIT(DIAGRAM(CIRCUIT)).

```
                    |============DIAGRAM==========
CIRCUIT|C_NAME|...|  |DNBR|AUTHOR|=SUB_CIRCUIT=
-------+------+---+==+----+------+=============
       |      |   |  |    |      |     H.
```

The Result of a Hierarchic View Definition

The result of a hierarchy definition over viewname is a hierarchical view (called H.viewname). G-WHIZ automatically defines recursively-computed functions LEVEL., PATH., and PREV. For example:

```
H.PART|NAME|COST|...|LEVEL.| PATH.|-/\PREV.-|=\/SUBPART=
------+----+----+---+------+------+---------+==========
      |    |    |   |      |      |         |
```

LEVEL. gives the entity's depth in the tree (starting at 1). PATH. gives the position in the traversal of the tree. For example, the fourth SUBPART of the second SUBPART of the beginning of the first tree has PATH. = 1.2.4.

PREV. is an entity-valued function that gives the hierarchic predecessor (parent) of an entity in the view. The hierarchic predecessor of an entity in the hierarchy is unique, even if the underlying PART is a SUBPART of several different entities. Since PREV. is entity-valued, it can be EXPANDed like any other entity-valued function. PREV. is particularly useful for defining functions in terms of the value of that function in the PREV. node.

---

(2) The problem is shared by all "first order languages," including QUEL, SQL, etc. [AU, Mai]. QBE [Date], Oracle [JS], and a proposal in [Cle] include facilities for defining and manipulating hierarchies, though recursively-defined functions are not discussed. The exact power of these systems is hard to judge, because descriptions in the literature are somewhat sketchy.

PREV. is marked with an up arrow (/\); the successor function (SUBPART) is marked with a down arrow (\/). If no selection on beginning entities is specified, the resulting hierarchy is rooted at entities that have no predecessor (i.e., that are not SUBPARTs). The functions, LEVEL., PATH., and PREV. are subject to all the usual operations on computed functions, except that their names are reserved words.

The content of a hierarchical view is defined by the the algorithm below (though the actual computation strategy may be different). Nodes of the hierarchy are instances of the view against which the hierarchy was defined.

1. Find view entities satisfying the Begin (B.) qualification and begin building a tree from each of these.

2. For each entity in the hierarchy, include it and its children in the hierarchical view via the successor function. Evaluate any computed functions (including recursive functions such as PATH.) all of whose data is available from the underlying view or from the PREV. entity in the hierarchy. Continue by traversing each successor of a chosen entity.

3. Perform additional traversals to compute recursively-defined functions whose arguments became available on the previous traversal.

4. After the entire tree has been traversed and all recursive functions computed (this may require extra traversals), apply the qualification specified by the qualifiers without a prefix. This step is an ordinary qualification on the set of entities seen in the hierarchical view. For example, the grid below begins traversal for H.PART at the wing, and after traversal is complete imposes an ordinary selection on the resulting view, selecting parts whose COST>100 and LEVEL.>3.

```
H.PART| NAME | COST |...|LEVEL.| PATH. |-/\PREV.-|=\/SUBPART=
------+------+------+---+------+-------+---------+===========
      |B.wing| >100 |   |  >3  |       |         |
```

8.2 Recursively-Defined Functions

Application systems that traverse hierarchies often compute functions that summarize information about the hierarchy. LEVEL. and PATH. are two examples. One might also recurse upward, summing the weights of all PARTs descended from a given PART. These computations cannot be expressed in first order queries on the set of PARTs.

The powerful function-definition mechanisms of G-WHIZ can be used for these recursive definitions. Functions derived in such a way can be queried like any other computed function or used in the specification of the beginning node(s) or successor function of another hierarchical view built over the first. For example, LEVEL. is a system-defined recursive function. (If the underlying view already has a function LEVEL., PATH., or PREV., the new functions are denoted LEVEL2., PATH2., PREV2., etc.)

The following grids show a hierarchy defined over an earlier hierarchical view. In the example, the first hierarchy uses the successor function SUBPART and begins at the wing. The second hierarchy is built over the first and begins at the 4th level of the first hierarchy.

```
PART | NAME | COST |...|=SUBPART=
-----+------+------+---+==========
     |B.wing|      |   |  H.

H.PART|NAME|COST|...|LEVEL.|PATH.|-/\PREV.-|=\/SUBPART=
------+----+----+---+------+-----+---------+===========
      |    |    |   | B.4  |     |         |
```

The user need not know how to define hierarchies in order to define recursive functions. Given a hierarchical view that already is defined, the user simply inserts a new function and provides a defining expression by using functions of PREV. for downward computations, or by using the successor function (e.g. SUBPART) for upward computations. (Downward and upward recursions cannot be in the same function definition).

The next example computes Cumulative Value Added (CUM_VAL_ADD) for each PART in the H.PART hierarchy by summing Value Added (VAL_ADD) for all PARTs below it in the hierarchy. The computation proceeds recursively from SUBPARTs to PARTs. The SUM. function returns 0 when summing over an empty set so we need not specify an initial value in this case.

Defining a Function by
<u>Aggregating Over Successors</u>

```
                              |   insert
                              V    key
H.PART|NAME|...|VAL_ADD|   CUM_VAL_ADD                 |=\/SUBPART=
------+----+---+-------+-------------------------------+===========
      |    |   |       |VAL_ADD +                      |
      |    |   |       | SUM.(CUM_VAL_ADD(SUBPART))    |
```

The next example shows a function recursively computed from a PART's hierarchic predecessor.

Assume that the hierarchy consists of physical parts where each PART appears only once as a SUBPART. Suppose OFFSET(PART) contains the x-distance between the leftmost edge of PART and the leftmost edge of PREV.(PART) (its immediate parent). The cumulative offset (relative to the entity at the top of a traversal) is the sum of the offsets along the path. We qualify the definition to set CUM_OFFSET to 0 at the top of the current traversal.

Defining a Function by
<u>Aggregating Over Predecessors</u>

```
H.PART|NAME|...|/\PREV.|OFFSET|     CUM_OFFSET        |=\/SUBPART=
------+-----+---+-------+------+----------------------+===========
      |     |   |       |      |0 & PREV. -           |
      |     |   |       |      |                      |
      |     |   |       |      |(OFFSET +             |
      |     |   |       |      | CUM_OFFSET(PREV.)) & |
      |     |   |       |      |  PREV. ~-            |
```

CUM_OFFSET of the top of this view is 0, because PREV.(PART) at the top of the hierarchy is - (null).

8.3 <u>Monotonicity and Geometry</u>

When a hierarchy is defined, the system asks the user which functions always increase or decrease between an entity and its successors. The user can specify, for example, that WEIGHT =< WEIGHT(PREV.).

This monotonicity declaration is used for conventional query optimization and for improving the user interface. For example, given a query:

```
PART | NAME  | WEIGHT |...|=SUBPART=
-----+-------+--------+---+=========
     |B.wing|  >10   |   |   H.
```

it is unnecessary to traverse SUBPARTs of PARTs weighing =< 10. Monotonicity also is used to reduce the amount of data presented to the user. If we return the information that a seat weighs more than 10 pounds, the interface may suppress superparts of the seat (e.g., cockpit, fuselage, and airplane). See [RHM84] for a full treatment of monotonicity.

<u>Hierarchical Views for Geometric Data</u>

The CAD/CAM data in CCDBMS requires a datatype to approximate geometric objects. GEOM_OBJ(PART) is an entity-valued function that stores the PART's shape, and also the position and location relative to each superpart. A recursive function POSITION (generalizing the OFFSET example) is defined to give the 3-dimensional offset and orientation of the PART relative to the beginning of a hierarchy.

GEOM_OBJ has several predefined functions (e.g., DISTANCE., EXTEND.) and predicates "contains", "contained in", "properly intersects", etc. The use of functional notation made it easy to include the abstract data type and specialized built-in functions and predicates. The monotonic behavior of these functions and predicates is predeclared to the system. For example, if a region contains a PART, it contains all SUBPART(PART).

Note that only <u>relative</u> position within the immediate superpart is physically stored. Subpart positions within an item are stored only once, regardless of how many times the item is used in the top-level product. Also, when the item is moved within its superpart, the relative position of the item's subparts remain fixed, and there are no stored absolute positions to be updated.

CCDBMS geometric facilities are not intended to perfectly represent shapes of three-dimensional objects. Solid modelling was too costly for our goal, which was to permit database queries that would limit the number of objects that would need careful inspection. Approximations using extents boxes were sufficient.

8.4 <u>Further Questions about Recursive Hierarchies</u>

We are currently investigating several issues:

1. Aggregation facilities from multiple parents

   In part hierarchies, each path to a part type (e.g., bolt) represents a different physical object. In some other structures (e.g., task scheduling networks) the object reached is the same, regardless of the path. The two behaviors must be distinguished, and facilities provided to aggregate information obtained along all the paths.

2. Query optimization

   We will investigate optimization strategies for various types of queries. [Me] investigated queries that touch nearly all the stored entities. However, a very different kind of query processing strategy is needed for an interactive system where most queries touch only a small subset of the entities.

   Architectural issues also will be investigated. In particular, how can optimization routines for hierarchies be integrated with the rest of a query optimizer?

3. Additional kinds of predicates

   For example, "Between" predicates, as in Find assemblies within the tail that include bolt type B123.

4. Facilities for defining a new hierarchy from a given one

We also want to make it easier to define a new hierarchy based on an existing one using its recursively-computed functions to specify the beginning nodes or the successor function.

5. Update

For hierarchies where no underlying entity appears more than once, update should be possible.

References

[AB] S. Abiteboul, N. Bidoit, "Non-First Normal Form Relations to Represent Hierarchically Organized Data", ACM Symposium on Principles of Database Systems, R. Fagin (ed.), 1984, 191-203

[AU] A. Aho and J. Ullman, "Universality of Data Retrieval Languages", 6th ACM Symposium on Principles of Programming Languages, 1979.

[Cle] E. Clemons, "Design of an External Schema Facility to Define and Process Recursive Structures", ACM Trans. Database Syst., Vol. 6, No. 2, June 1981, 295-311.

[Date] C.J. Date, An Introduction to Database Systems, Addison Wesley, 1977, pp 137-152.

[JS] G. James, W. Stoeller, "Operations on Tree-Structured Tables", X3H2-26-15 Standards Committee Working Paper, 1982, pp 81-92.

[Mai] D. Maier, The Theory of Relational Databases, Computer Science Press, Rockville, MD 1983.

[Man] F. Manola, "A Comparison of the Daplex and Relational Data Models in the CCDBMS Preliminary Design," CCA Report, July, 1984.

[Me] T. Merrett, Relational Information Systems, Reston Publishing, Reston VA, 1984.

[RHM] A. Rosenthal, S. Heiler, F. Manola, "An Example of Knowledge-Based Query Processing in a CAD/CAM DBMS", VLDB Conference, 1984, Singapore, 363-370.

[RKS] M Roth, H. Korth, A. Silberschatz, "Theory of Non-FIrst-Normal-Form Relational Databases", University of Texas Computer Science TR-84-36.

[Sh] D. Shipman, 'The Functional Data Model and the Data Language DAPLEX', ACM Trans. Database Syst., Vol. 6, No. 1, March, 1981.

[SS] Smith, J.M. and Smith, D.C.P., Database abstractions: Aggregation and generalization, ACM Trans. Database Syst. 2,2, June 1977, 105-133.

[Zloof] M. Zloof, "Query By Example", Proc. NCC44, May, 1975.

Acknowledgements

The language presented in this paper was originally sponsored by General Dynamics Data Systems Division. Further research on hierarchic facilities was sponsored by CCA and DARPA.

Bill Harrelson, Mort Goldman, Celia Shapiro, Jan Dreisbach, Deborah Hamill, and Lisa Haflin have participated in design or implementation of the G-WHIZ language and interactive interface. Bill Holmes (of General Dynamics), Richard Meier, Peter Gutterman, Frank Manola, and Umesh Dayal have made useful suggestions about the presentation.

The name G-WHIZ can be blamed on Mort Goldman and Celia Shapiro.