A High-level Data Manipulation Language for Hierarchical Data Structures

Barron C. Housel
Nan C. Shu

IBM Research Laboratory
5600 Cottle Rd.
San Jose, Ca. 95193

ABSTRACT: In this paper we assert that the hierarchical view of data will continue to be popular for a broad class of applications and users. In particular, some of these applications require complex data manipulation which, heretofore, has been dealt with procedurally. In this light, a nonprocedural language, CONVERT, is proposed as a high-level DBMS interface. CONVERT is meant to provide users with a tool for performing complex data manipulation and query of hierarchical data abstractions, called "Forms". Included in the paper are a description of the Form data abstraction and the CONVERT language, as well as a complete illustrative sample application.

1. Introduction

In a previous paper [1] we presented a high-level nonprocedural language, CONVERT, which was designed for specifying translation and restructuring for the purpose of data conversion. This paper explores the use of an enhanced version of CONVERT as a high-level user interface for data base management systems (DBMS's) for query and data manipulation. In our studies on data translation, we found hierarchical data structures to be the most prevalent in today's environment. Therefore, CONVERT was based on a hierarchical data abstraction referred to as a Form [1]. A Form is a two dimensional tabular representation of a hierarchical data structure and is described in depth in section 2. The CONVERT language, then, consists of a few conceptually simple operations which produce an output Form from one or more input Forms.

Recently, there has been a great deal of interest in high-level, nonprocedural data base languages because they enable users to state their requests in a natural way without regard to low level details (e.g. access paths, etc.) of the particular DBMS. In addition, they offer a basis for achieving greater data independence, global optimization, integrity checking, and security. However, most of these languages, for example, SEQUEL [2], SQUARE [3], DSL-ALPHA [4], and QUEL [5] to name a few, are based on the relational data model [6].

The motivation for exploring the use of CONVERT as a DBMS user interface lies in our belief that hierarchical data models will continue to be a popular user (application) view for the indefinite future. Also, we feel that existing hierarchical nonprocedural languages lack the capability required for a number of significant applications, particularly those involving complex data manipulation for report generation.

Why Another Language?

CONVERT is certainly not the first nonprocedural language developed for hierarchical DBMS's. For example, HQL [7] is a highly user oriented query language for hierarchical structures. Similarly, MRI's System 2000 [8] offers a powerful nonprocedural query and data manipulation language. However, these and other similar languages are designed to operate within a given hierarchical data base structure which has been predefined in the system by the data base administrator.

The uniqueness of CONVERT lies in its power to specify operations over a family of hierarchies and in its ability to specify that a new hierarchical structure is to be generated from existing ones. This concept is also central to the relational model, in which the relational operators (join, projection, etc.) may be applied to a family of base relations in order to produce a new relation which complies with a particular user view.

CONVERT and the Different Data Models

Hierarchical Systems. One of the main difficulties in hierarchical systems is in developing applications which require a view of the data base (application view) which radically departs from the schemas developed during data base design (native views). For example, such an application view may imply a reorganization of the data base requiring combinations of data from several hierarchies. Currently, such applications must be designed to procedurally "navigate" via access pointers through the various native data base structures in order to collect the required information. As an alternative, we propose using CONVERT as a means of prescribing how the particular application view is to be derived from the native views. One may consider this prescription as resulting in a temporary file for use by an application program (e.g. report formatter), assuming no updating. Alternatively, CONVERT could be viewed as a more interactive interface in which instances of the application view are generated dynamically from the source views.

Network Systems. Most of the previous discussion also applies here. The main difference is that the predefined data base schemas in the data base are not restricted to hierarchical structures; thus, CONVERT cannot be used directly.

It should be mentioned that a nonprocedural interface for a network DBMS has yet to be demonstrated, although McGee's work [9] has taken a nice step in that direction. The problem, of course, is due to the complexity of the network model in that the potentially large number of access paths within a network structure makes the formalization required for nonprocedural query and data manipulation difficult. One approach for overcoming this problem is to use a subschema definition facility for logically partitioning the network into a set of "logical views" which reflect simpler data models (i.e. hierarchical or relational). The facility of IBM's Information Management System [10] (IMS) for defining logical hierarchical structures in terms of several inter-connected physical hierarchies is an example of this approach. Given such a facility, the network could be viewed as a family of hierarchies, thus permitting the use of CONVERT.

Relational Systems. It is not possible to fully implement CONVERT on a relational system which adheres strictly to the relational model as presented by Codd [6]. This is because (a) relations preclude hierarchical structure, since they must be in at least "first normal form" (i.e. only simple domains); (b) all relations must have unique keys, thus preventing the occurrence of duplicate tuples; and (c) there is no ordering property among tuples of a relation. It is possible to simulate the different hierarchical levels via a set of relations where each relation corresponds to a node in the hierarchy and contains among its attributes the keys of all the ancestor nodes (relations) in the hierarchy. (note: this is a sufficient but not necessary condition). However, in CONVERT, instances of a Form may be ordered and duplicates are allowed (i.e. keys need not exist).

In the remainder of the paper, section 2 describes terms and the underlying concept of the Form data abstraction. Section 3 describes the CONVERT language, and section 4 presents a sample application which illustrates the semantics of the operators and the derivation of a new application view from several native views. Section 5 presents the summary and conclusions.

2. The Form Abstraction

A Form is similar in concept to a COBOL or PL/I structure except that it is viewed as a conventional table. A Form structure (schema) can be defined recursively in terms of the constructs Field and Group.

D1. A Field (schema) describes an atomic unit of data.

D2. A Group (schema), G, consists of a sequence of Components <C1,...,Cn>, where Cj (j=1,...,n) is a Field or Group.

D3. A Form (Form Group), consists of a Group which is not a Component of another Group.

Corresponding to the schema definitions are the instance definitions:

D4. A Field instance, is a single data value.

D5. A Group instance, I(G), of Group G, is a sequence of Component instances, <I(C1),...,I(Cn)>, where Cj (j=1,...,n) are Components of G. If Cj is a Field, I(Cj) is a Field instance. If Cj is a Group, I(Cj) is a sequence of homogeneous instances <I1(Cj),...,Im(Cj)>, where m is greater than or equal to 1.

D6. A Section is an instance of a Form Group (D3). A Group instance which is not a section will also be refered to as a subsection.

D7. A Form instance (i.e. file or data base) consists of a sequence of sections of the Form.

In Figure 1, the Form DEPT consists of Components DNO, MGR, EMP, PROJ and DBUDG, where DNO, MGR, and DBUDG are Fields and EMP, PROJ are Groups. In like manner, EMP consists of the Components ENO and JC, and PROJ is comprised of the Components PJNO, CEST, CACT, and EQUIP.

DEPT									
		EMP		PROJ			EQUIP		
DNO	MGR	ENO	JC	PJNO	CEST	CACT	MNO	UF	DBUDG
D1	47	80	8	J4	50	43	M5	.2	180
		19	7	J3	70	80	M8	.3	
		67	9				M15	.5	
		43	2	J2	20	27	M2	.4	
D3	18	72	4	J1	40	20	M4	.7	300
		68	9	J3	63	50	M10	.8	
		96	8				M15	.1	
		53	4	J2	170	177	M10	.2	
							M8	.3	

Figure 1 - Sample "Form" for DEPT.

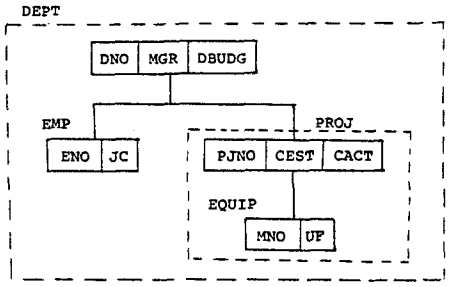

Figure 2 - Hierarchy Graph for Figure 1 (DEPT)

At the instance level, the two DEPT sections are separated by the dash line. An instance (subsection) of the Group PROJ, for example, is:

J3	70	80	M8	.3
			M15	.5

However, the instance of the Component PROJ in a DEPT section consists of all the PROJ subsections within the section.

As pointed out in [1], the hierarchical structure of a Form (Group) can also be represented by a "hierarchy graph" (HG) as shown in Figure 2 for Figure 1. The Field Components of the Group are collected to form the parent node, and the Group Components define hierarchical subgraphs which connect to the parent. A Group name (e.g. PROJ, DEPT) refers to an entire HG as denoted by the dash boxes. Note that in the HG representation of a Form, the positional information of the columns of the Form is lost.

Two or more Fields which are Components of the same Group will be referred to as twins. A Component C is contained in a Group G if it occurs in the HG defined by G. Conversely, G contains C. It is important to distinguish the difference between a Component, C, being a Component of a Group G, and being contained in a Group G. A component C, can be a component of only one Group, as defined in D2; however, C may be contained in any number of Groups because of the recursive defintion of Group. Thus, with respect to Figure 1, ENO, for example, is a Component of EMP and contained in DEPT, but is not a Component of DEPT.

Form Properties. From the previous definitions we see that instances within a Form (i.e. (sub)sections) occur in a specific sequence. Thus, it is possible for Forms to be sorted. This property is important from a user's viewpoint, since frequently he desires his output ordered in a particular way.

In defining a Form, specification of keys is optional; that is to say, duplicate instances are permitted. In some cases duplicates may be semantically meaningful, even in an integrated data base. For example, suppose in a banking application, a customer writes two checks against the same account for the same amount on the same day. If the data base designer presumed that ACCT#, DATE, AMT was sufficient to describe "checks received," then duplicates would occur. In CONVERT the user must explicitly state that duplicates are to be eliminated. This brings out a basic philosophy regarding CONVERT, namely, that except for maintaining consistent hierarchical relationships, no a priori semantic restrictions are imposed on the manipulation of Forms. We agree with Schmid and Swenson [11], that the formal specification of the semantic properties of data are not well understood. In designing CONVERT, we provide powerful operators for data restructuring assuming users posses sufficient knowledge of the data semantics to produce meaningful results. In this regard, we do not view CONVERT as a "casual user" language. Flexibility in data restructuring is particularly important where a user wishes to produce an application view which may not have been anticipated during data base design.

3. The CONVERT Language

The CONVERT language presented below differs slightly from that described in [1]. Here, we concentrate on describing the semantics of the operations. The reader is referred to [1] for further discussions about the various operations. The basis of CONVERT is a set of Form Operations consisting of SELECT, CASE-select, SLICE, CONSOLIDATE, GRAFT, SORT, MERGE, ELIM_DUP, and a set of aggregate functions (COUNT, SUM, AVG, MAX, and MIN). These operations require one or more Forms as input operands and produce an output Form as the result. The general format for all Form operations is

Operator (Operands)

The input Form operands can be defined in terms of Form operations allowing operations to be nested. To save the results of a Form operation, an Assignment statement is provided having the format

Target-Form <- Form-operation ;

Target-Form may consist of a simple name which is not defined in a FORM definition (e.g. F2 <- ...), in which case all Component names are inherited from the schema defined by the Form-operation. Alternatively, a Form definition may be specified in order to assign new names. This can be specified in two ways as shown below (see Figure 1).

```
(1)  FORM DEMPS(DNUM,(EMPL(ENO,JCODE)));
     DEMPS <- SELECT(DNO,EMP FROM DEPT);

(2)  DEMPS(DNUM,(EMPL(ENO,JCODE))) <-
                SELECT(DNO,EMP FROM DEPT);
```

The FORM statement is provided in CONVERT for declaring Form definitions (schemas), and is the only data definition statement. Field definitions are provided elsewhere [12]. Components can be grouped under one name for referencing convenience, e.g. DATE(MO,DA,YR). In defining Forms, the repeating Components such as EMPL are enclosed by parentheses. A repeating Field, f, is viewed as a Group with one Field, and is denoted as (f). It should be mentioned that the schema of the Target-Form, if specified explicitly, must be consistent with that of the resulting Form operation (except for CONSOLIDATE, discussed later).

Now we shall describe the Form operations, appealing to the definitions of section 2 and also to the examples of the next section (reference to "Step n" pertains to the examples in the next section).

The following conventions are used in the descriptions given below. Brackets [..] denote an optional clause; alternative constructs are enclosed in braces {..} and separated by the vertical bar (e.g. {a|b|..}). The default alternative, if any, will be underlined. The capital letter, F, (and F1, F2, ...) indicate a Form specification (i.e. a Form name or Form operation); keywords are capitalized and other syntactical constructs are in lower case.

A. SELECT

```
SELECT([ALL EXCEPT][e1,..,en] FROM F
     [WHERE selection-criteria ] )
```

The SELECT operation can be viewed as two basic processes: 1) pruning out all (sub)sections which fail to meet the "WHERE clause" selection-criteria, and 2) mapping the (sub)sections which remain from 1) to target instances according to the Component selection list e1,...,en. If the WHERE clause is omitted, all (sub)sections are selected. If no Component selection list is specified, all Components (columns) are selected.

If the ALL EXCEPT clause is absent, e1,...,en, specifies the Components to be included in the result and the order of their occurrence. An ej may be an arithmetic expression consisting of the operators +, -, /, and *, and operands which may include Fields, aggregate functions over repeating fields, and literals. All Fields used in arithmetic operations in ej must be twins. If an aggregate function and Field are referenced in the same expression, the result of the aggregation and the referenced Field(s) must be in one to one correspondence.

Examples of ej with respect to DEPT in Figure 1:

```
1)   (CEST-CACT)/2                    (valid)
2)   DBUDG-SUM(CACT FOR EACH DEPT)   (valid)
3)   DBUDG-CEST                       (invalid)
4)   DBUDG-SUM(UF FOR EACH PROJ)      (invalid)
```

An ej which may also consist of a Field assignment may be used to rename a selected Component or give a name to an expression (e.g. CMARG <- CEST-CACT).

The ALL EXCEPT clause indicates all columns are to be mapped to the result with some exceptions. An ej may be a simple component name which indicates that the component is to be omitted from the result (see Steps 2,9,11,12). Alternatively, an ej may consist of a Field assignment which defines a result Field as an expression of (0 or more) Fields from the selected file (see Step 9).

The selection-criteria consists of a conjunction of predicates of the form, p1 AND p2 ... AND pn. A predicate pj may specify a "Field-criteria" which compares single values or "set-criteria," which tests sets of values. Field-criteria predicates have the form, "f1 comp_op f2," where comp_op is one of the comparison operators (<, =, ¬=, etc.), and f1 and f2 are arithmetic expressions (as described for Component selection) or character strings.

Set-criteria have the form "s1 [NOT] set_comp_op s2" where set_comp_op may be IN (s1 is a subset of s2), INTERSECT, or SAME AS (s1 = s2). The set operands s1 and s2 can be specified as a list of constants, a one column Form (i.e. a Form name or operation which returns a one column Form), or a "SET-clause". The SET-clause is provided for referencing sets of values within a given section of a Form; its syntax is given as

```
SET ( f [ FOR EACH gp-name ] )
```

f is a Field name which identifies the column containing the values of the set, and the FOR EACH clause limits the scope of reference to instances of the Group, gp-name. If the FOR EACH clause is absent, all values of f in a given section are referenced. With respect to Figure 1, the operation

```
SELECT(PJNO,MNO FROM DEPT WHERE
          DNO = 'D1' AND
          SET(MNO FOR EACH PROJ) INTERSECT LIST('M5','M15'))
```

would yield:

PJNO	(MNO)
J4	M5
J3	M8
	M15

In this example, SET(MNO FOR EACH PROJ) references threes sets in the first section, {M5}, {M8,M15}, and {M2}. If the FOR EACH clause were omitted, only one set containing all values of MNO (in the section) would have been referenced.

Now we focus on how instances which fail to meet the selection-criteria are identified. For sake of brevity we will treat only the case where each predicate, pj, contains only one Component from the selected Form, F (although Fields from other Forms may occur). These semantics can be simply stated in terms of the definitions of section 2. That is, consider any Group, G(C1,...,Cn), whose Group instance, I(G), is defined by the sequence <I(C1),...,I(Cn)>, and each I(Cj) (j=1,...,n) is given by the sequence of instances <I1(Cj),...,Im(Cj)> (m at least 1). Then the selection algorithm is given as follows:

1) Mark all Component instances (fields or sets) which fail to satisfy the selection criteria.

2) If all instances, <I1(Cj),...,Im(Cj)>, are marked, then mark I(Cj).

3) If any Component instance, I(Cj), of I(G) is marked, then mark I(G).

4) Recurse on 2), 3) until no more instances can be marked and select the unmarked instances.

Although not presented here, a limited form of the disjunctive "OR" is permitted in the selection-criteria. Some of the conceptual difficulties of using the "OR" in an

unrestricted way, with respect to hierarchical structures, have been demonstrated by Hardgrave [13]. The desired effect of OR (union) can be accomplished with combinations of the operators SELECT, MERGE, and ELIM_DUP. In general, the selection-criteria may span multiple Forms as long as there are sufficient equal (=) field-criteria to "tie" one Form to another (illustrated in Step 2).

B. CASE-select

```
CASE( FROM F WHEN case-expr comp_op
      c1 : case-select(1),
      c2 : case-select(2),
      ...
      cn : case-select(n)
  [,OTHERS : case-select(n+1)] );
```

In the above construct, F is the source Form of the CASE selection. "case-expr" is an arithmetic expression containing one or more Fields of F, and comp_op is a comparison operation as given previously. Each cj (j=1,...,n) is a constant or a list of constants, and case-select(j) is a selection operation with the same format as the SELECT operation in A, except the "FROM F" specification is not given.

In the SELECT operation (described in A), the mapping of input to output is uniform for all sections in F. The CASE-select operation allows this mapping to vary from section to section depending on the evaluation of case-expr. For each (sub)section, the tests "case-expr comp_op cj" (j=1,,...,n) are performed until "true" is returned for some j, in which case case-select(j) is chosen to map the components of that section to the output. If none of the tests return "true" and the OTHERS option is specified, case-select(n+1) is chosen; otherwise no mapping is performed for the current section. Regardless of the "case" selected, all output sections must conform to a common hierarchical structure.

For example, in Figure 1, suppose it is desired to increase CEST by 10% if PJNO is 'J1', by 15% if PJNO is 'J2', and otherwise leave the current value. This would be stated as

```
CASE(FROM DEPT WHEN PJNO =
  'J1': SELECT(ALL EXCEPT CEST <- CEST*1.1),
  'J2': SELECT(ALL EXCEPT CEST <- CEST*1.15),
OTHERS: SELECT(ALL));
```

Taking a more complex example,

```
CASE(FROM DEPT WHEN PJNO =
  'J2': SELECT(DNO,CEST),
  'J3': SELECT(DNO,CACT WHERE CEST > 60 AND CEST < 70));
```

Result:

DNO	(x)	Source of x
D1	20	CEST
D3	50	CACT
	170	CEST

The Fields contained in case-expr determine the Components which can be included in the varying mappings and the WHERE selection criteria. Specifically, if case-expr contains Fields of Group G, then only those Groups and Fields contained in G can participate. Other mappings must be identical in all the case-select's. In the previous example, the second Field selected varies from case 1 to case 2. This is permitted since CEST and CACT are both contained in the Group PROJ, which includes PJNO as one of its Field Components. The mapping of DNO is (and must be) invariant in both cases. By using the ALL EXCEPT option in the case-select's, only the varying mappings (ej's) need be specified explicitly.

C. SLICE

```
SLICE(f1,f2,...,fn FROM F)
```

The objective of SLICE is to provide the facility for generating "flat" tables from hierarchical Form structures. In the above construct, f1,...,fn are Fields contained in F such that for any two fields fi and fj, the following condition holds. If fi (fj) is a Component of Group G, then fj (fi) must be contained in G.

Examples:

1) SLICE(DNO,MNO, DBUDG FROM DEPT) (see Figure 1)

Results:

DNO	MNO	DBUDG
D1	M5	180
D1	M8	180
D1	M15	180
D1	M2	180
D3	M4	300
D3	M10	300
D3	M15	300
D3	M10	300
D3	M8	300

2) SLICE(DNO,ENO,PJNO FROM DEPT)

This is invalid because ENO is a Component of EMP, but PJNO is not contained in EMP; similarly, ENO is not contained in PROJ.

D. CONSOLIDATE

(a) Target-Form <- CONSOLIDATE(F) ;

(b) CONSOLIDATE(F1 AS Target-Form)

The function of CONSOLIDATE is opposite of SLICE in that it produces an output Form with more hierarchical structure than the input Form in order to remove data redundancy. The input Form, however, does not have to be a relational table. CONSOLIDATE varies from the other operations in that the schema of the output must be specified explicitly in order to define how the consolidation is to be carried out. Format (b) may be used if it is desired to nest a CONSOLIDATE within another operation. The semantics is illustrated by example. In Figure 3a, SPT2 is derived from SPT1 by

SPT2(S#,SL,(PT(P#,QTY))) <- CONSOLIDATE(SPT1);

Note that 'S#,SL' are unique in SPT2, but redundant in SPT1. The Group PT is established for mapping the unique Components associated with 'S#,SL' in SPT1. Thus, the target schema serves the function of determining the "uniqueness criteria" and defining the additional Groups required in order to factor out the unique Fields.

The "consolidation" can propagate to any depth. For example, in Figure 3b, F8 can be consolidated to F8A by specifying

FORM F8A(X,(YZ(Y,(Z))),(W));
CONSOLIDATE(F8 AS F8A)

The definition of F8A establishes that X is unique within the file F8A, and Y is unique for all subsections of YZ.

SPT1

S#	SL	P#	QTY
S1	SJ	P1	3
S1	SJ	P2	2
S1	SJ	P3	4
S1	LA	P4	2
S1	SJ	P5	1
S1	LA	P6	1
S2	SF	P1	4
S2	SF	P2	4
S3	NY	P3	4
S3	LA	P5	2

SPT2

S#	SL	PT	
		P#	QTY
S1	LA	P4	2
		P6	1
S1	SJ	P1	3
		P2	2
		P3	4
		P5	1
S2	SF	P1	3
		P2	4
S3	LA	P5	2
	NY	P3	4

(Figure 3a)

F8

X	YZ		(W)
	Y	Z	
A	P1	10	WA1
	P2	20	WA2
	P3	30	
B	P1	15	WB1
	P4	25	
	P5	35	
A	P1	2	WA3
	P3	4	WA1
	P5	6	
B	P4	30	WB1
C	P5	20	WC1

F8A

X	YZ		(W)
	Y	(Z)	
A	P1	10	WA1
		2	WA2
	P3	30	WA3
		4	WA1
	P5	6	
B	P1	15	WB1
	P4	25	WB1
		30	
	P5	35	
C	P5	20	WC1

(Figure 3b)

E. GRAFT(F1,...,Fn ONTO Fm [AT f] WHERE match-conditions)

GRAFT is similar to the "join" operation in the relational model. Its function is to combine two or more Forms based on the specified "match-conditions". Each Fj (j=1,...,n), referred to as the branch Forms, is combined with the root form, Fm. In terms of trees, one can view the operation as "grafting" the HG defined by the branch Forms onto the HG defined by the root Form.

First, we will describe GRAFT with only one branch Form (n=1). The match-conditions can be stated in terms of either "Equal-conditions" or as "Prevail-conditions".

Equal-conditions are stated as

(i) f1 = g1 AND f2 = g2 ... AND fn = gn

where fi (i=1,...,n) are twin Field Components of F1 (i.e. Fields in the root node of the HG of F1), and gi (i=1,...,n) are twin Field Components of Fm or of some Group contained in Fm. With Equal-conditions, each equality predicate in the conjunct must be satisfied before an output can result. If the predicates evaluate true, then the F1 section is concatenated with the (sub)section of Fm and output; however, the common Fields (those used in the predicates) are recorded only once. The match Fields in the root Form determine the Group to be expanded. For example, the result of

```
F3 <- GRAFT(F1 ONTO F2 WHERE F1.A = F2.A);
```

is given in Figure 4. In this example, the Group Y is expanded to include all the Components of F1.

F1

A	B		E
	C	D	
a1	c1	d1	e1
	c2	d2	
a2	c3	d3	e2

F2

X	Y	
	A	Z
x1	a1	z1
	a3	z2
x2	a2	z3

F3

X	Y		B		E
	A	Z	C	D	
x1	a1	z1	c1	d1	e1
			c2	d2	
x2	a2	z3	c3	d3	e2

Figure 4 - Graft Example

To describe the case for n > 1, let "ec(j)" denote the Equal-conditions for n=1 in order to GRAFT the branch Form Fj onto the root Form Fm. The Equal-conditions for n > 1, can be stated in terms of the sequence C1, C2,...,Cq, where each Cj is some ec(k) or a conjunction:

```
ec(j1) AND ec(j2) AND ...
```

An output is produced for each section of Fm if at least one Cj (j=1,...,q) evaluates true. Instances from a branch Form, Fa, are included in each output section if ec(a) is a term in some Cj which evaluates true.

The Prevail-conditions is specified when it is desired to produce an output for input instances of one or more of F1,...,Fn, Fm, even if there are no satisfying match conditions. For example, in the previous example, if

```
F3 <- GRAFT(F1 ONTO F2 WHERE F2.A PREVAIL F1.A);
```

were specified, the Y Component instance for the first section would consist of

```
a1   z1   c1   d1   e1
          c2   d2
a3   z2   -    -    -
```

For further discussion of PREVAIL, see [1].

The "AT f" specification is used to alter the column position of Fm where the "grafted" information is to be inserted. More examples of GRAFT are shown in steps 11 and 12.

F. SORT(F BY f1 [{ASC|DES}],...,fn [{ASC|DES}])

fi (i=1,...,n) are Fields contained in the Form F, and specify the sort keys. If some fi is a Field Component of some Component Group, then only the subsections within the Group instances are sorted. For example, if SORT(DEPT BY DNO,ENO DES) were specified for DEPT of Figure 1, then the result would be sorted by DNO, and within each DEPT section the EMP subsections would be sorted by ENO in descending order (e.g. within the section where DNO= 'D1', then the EMP subsections would be <80,8> <67,9> <43,2> <19,7>).

G. MERGE(F1,...,Fn)

Sections from the homogeneous Forms are combined into one output Form having the same structure. If MERGE is nested within another operation, the Component names of the result are inherited from the schema of the first Form in the list (i.e. F1). In general, the order of the output is unpredictable.

H. ELIM DUP(F)

This operation copies the sections of F to the output with duplicates eliminated.

I. Aggregate Functions

function (f [FROM F] [FOR EACH gp-name])

"function" is one of the functions SUM, MAX, MIN, COUNT or AVG. "f" is the Field to be aggregated within the Form F, and gp-name is a Group name (possibly qualified) which defines the scope of aggregation.

As shown in previous examples and in section 4, these functions can be used in arithmetic expressions within SELECT lists and also in the selection criteria. In that context, the FROM clause may be omitted, since the Form name is specified in the FROM clause of the SELECT. However, aggregate functions can also be used as self-contained Form operations. If the FOR EACH clause is missing, f is applied to all instances of f in F, returning a scalar (i.e. a one column, one row Form). For example, in Figure 1, SUM(UF FROM DEPT) would result in 3.5. If the FOR EACH clause is specified, a vector is returned. The operation SUM(UF FROM DEPT FOR EACH DEPT.PROJ) would return a vector <.2,.8,.4,.7,.9,.5>, while SUM(UF IN DEPT FOR EACH DEPT) would yield <1.4, 2.1>.

More CONVERT Statements

Up to this point we have really discussed only the CONVERT statements for the purpose of data restructuring. Additionally, in the context of a DBMS interface, we need statements for updating and outputting existing Forms.

J. ADD (F1 TO gp-name [IN F2 WHERE add-crit]) ;

The function of this statement is to add sections to F2 or subsections to some Group "gp-name" contained in F2. The Form F1 contains the (sub)sections to be added. If gp-name equals F2 then sections are to be added to F2 from F1 and the "IN-WHERE" clause is omitted (e.g. ADD(NEWDEPTS TO DEPTS);). The "add-crit" is a conjunction of equal predicates. The schema of F1 must be a subschema of F2. Specifically, F1 must contain gp-name as one of its Components. The subsections of gp-name in F1 will be added to Component instances of gp-name in F2 depending on the add-crit. Components other than gp-name in F1 are used in the add-crit to locate the specific component instance of gp-name in F2 being updated. For example, suppose we want to add employees to departments 'D1' and 'D3' (Figure 1), where F1 is defined as

F1		
	EMP	
DNO	ENO	JC
D1	85	7
	89	8
D3	94	9
	99	4

The update would be specified as

ADD(F1 TO EMP IN DEPT WHERE F1.DNO=DEPT.DNO);

There must be sufficient Fields in F1 and terms in the "add-crit" to isolate the specific

(sub)section which contains the Component instance of the Group (gp-name) being updated.

K. DELETE (gp-name [IN F] [WHERE delete-crit]) ;

The function of this statement is to cause deletion of sections in F or subsections of a Group "gp-name" contained in F subject to certain criteria, "delete-crit". If gp-name equals F (the Form name), then the IN phrase may be omitted. If the WHERE clause is omitted, all the instances of gp-name are deleted. The delete-crit has the same form as field-crit for SELECT. We illustrate with several examples with reference to Figure 1.

1) Delete machines (EQUIP) in projects if the utilization factor (UF) is less than .3.

```
DELETE(EQUIP IN DEPT WHERE UF < .3);
```

2) The same as in 1) except only for DNO = 'D1'.

```
DELETE(EQUIP IN DEPT WHERE DNO = 'D1' AND UF < .3);
```

3) The same as in 1) except only if PJNO is either 'J2' or 'J3'.

```
DELETE(EQUIP IN DEPT WHERE UF < 0.3
     AND SET(PJNO FOR EACH PROJ) IN LIST('J2','J3'));
```

L. Updating an Existing Form "in Place"

Update-in-place can be performed on an existing Form by utilizing the SELECT and CASE operations following the keyword "UPDATE". In updating, however, the ALL EXCEPT version of the SELECT's must be used, since Component selection has no meaning. For example, the request: "increase DBUDG for DEPT for department D1" could be stated as

```
UPDATE SELECT(ALL EXCEPT DBUDG <- DBUDG*1.1 FROM DEPT
              WHERE DNO = 'D1');
```

M. Output Statements

```
{DISPLAY|PRINT} F ;
```

The DISPLAY and PRINT operators designate on-line and remote printer, respectively. A predefined format for the Forms is assumed. In the above statement, F may be either a Form name or a Form operation.

This concludes the description of the CONVERT language.

4. A Sample Application

In this section we illustrate the salient features and usage of CONVERT by programming a sample application. The source data bases for the application are depicted by the Forms in Figure 5, which describe various departments (DEPT), projects (PROJECTS), and employees (EMPLOYEE) within a corporation. In this company it is known that a) more than one department can work on a project; b) a department may be responsible for projects in different locations, although a project has only one location; c) more than one project may use a machine (EQUIP).

Application Description

The president of the company is concerned about the cost overrun on those projects located in San Jose and wants a report on them. Specifically, he wants a breakdown of the departments which work on each project and the equipment used by each project. The final report (given in Figure 6) is to be ordered on project number (PJNO) and department number (DNO), and includes the total project budget (BUDG) and the total actual cost (TCOST) that all the participating departments have spent on each project. For each department working on a given project, the report is to contain DNO, the manager's name (NAME) and phone (PHONE), and the difference between the estimated and actual cost (CMARG=CEST-CACT) the department has spent on each project. In addition, the president wants all departments which are over budget to be boldly marked (he's near sighted) to indicate the budget status (BUDSTAT). For each machine used for a given project, he wants the machine number (MNO) and the fraction (TUF) that each machine has been used for each project.

This example was intentionally constructed to be rather involved in order to illustrate that many commercial applications cannot be handled easily by a simple query and also to show how the problem can be decomposed into subproblems using CONVERT.

The Solution

Now we show a series of simple CONVERT statements which will produce the result in Figure 6 from the source Forms in Figure 5. The output Forms resulting from each operation are given in the Appendix.

DEPT

		EMP		PROJ			EQUIP		
DNO	MGR	ENO	JC	PJNO	CEST	CACT	MNO	UF	DBUDG
D1	47	83	8	J4	50	43	M5	.2	180
		19	7	J3	70	80	M8	.3	
		67	9				M15	.5	
		43	2	J2	20	27	M2	.4	
D3	18	72	4	J1	40	20	M4	.7	300
		68	9	J3	63	50	M10	.8	
		96	8				M15	.1	
		53	4	J2	170	177	M10	.2	
							M8	.3	
D2	98	10	2	J6	83	74	M12	.6	250
		15	7				M11	.2	
		93	8	J5	57	63	M7	.6	
		88	8				M9	1.0	
		86	4	J3	40	70	M8	.2	
		91	7				M15	.1	
D4	20	73	7	J4	90	160	M5	.7	175
		62	3				M6	1.0	
		87	9	J2	40	37	M2	.5	
		63	6				M3	1.0	
		17	5						
D5	25	57	3	J5	10	12	M11	.2	50
		52	8				M12	.1	
		45	7	J6	15	14	M7	.3	

PROJECTS

PJNO	BUDG	LOC
J1	50	SF
J3	190	SJ
J5	75	LA
J2	250	SJ
J6	110	SD
J4	160	SJ

EMPLOYEE

ENO	DNO	NAME	PHONE	JC
10	D2	JONES	7536	2
18	D3	LEWIS	6673	7
20	D4	CHU	3348	10
25	D5	SMITH	7777	10
47	D1	KING	6397	10
98	D2	PARKS	6967	10
..	..	..	..	.

Figure 5 - Source Forms for Sample Application

REPORT

			DEPT				EQUIP	
PJNO	BUDG	TCOST	DNO	NAME	PHONE	CMARG BUDSTAT	MNO	TUF
J2	250	241	D1	KING	6397	-7 * OB *	M2	.9
			D3	LEWIS	6673	-7 * OB *	M3	1.0
			D4	CHU	3348	3 * OB *	M8	.3
							M10	.2
J3	190	200	D1	KING	6397	-10 * OB *	M8	.5
			D2	PARKS	6967	-30 * OB *	M10	.8
			D3	LEWIS	6673	13	M15	.7
J4	160	203	D1	KING	6397	7	M5	.9
			D4	CHU	3348	-70 * OB *	M6	1.0

Figure 6 - Result of Sample Application

Step 1. Define the source Form schemas.

```
FORM DEPT(DNO,MGR,(EMP(ENO,JC)),
         (PROJ(PJNO,CEST,CACT,(EQUIP(MNO,UF)))),DBUDG)
     KEY IS (DNO WITHIN DEPT),
     KEY IS (ENO WITHIN EMP),
     KEY IS (PJNO WITHIN PROJ FOR EACH DEPT),
     KEY IS (MNO WITHIN EQUIP FOR EACH PROJ);

FORM PROJECTS(PJNO,BUDG,LOC) KEY IS (PJNO);

FORM EMPLOYEE(ENO,DNO,NAME,PHONE,JC) KEY IS (ENO);
```

The WITHIN clause identifies the Group in which the key Field(s) is a Component.

If the WITHIN clause is omitted, the key Field(s) is assumed to be a Component of the Form Group. The FOR EACH clause limits the scope of uniqueness of the key Field(s). If the FOR EACH clause is omitted, the key Field(s) values are assumed to be unique for all the Field instances in the Form. For example, if projects could only be worked on by one department, then the "FOR EACH DEPT" phrase would have been omitted. An ordering clause is also permitted if the source is known to be ordered (e.g. ORDERED ON (DNO ASC)).

Step 2. Select only those sections from DEPT which have projects located in San Jose and omit the project information (PROJ) for those projects located elsewhere, and also the department's employees (EMP), and the department budget (DBUDG).

```
DEPTSJ <- SELECT(ALL EXCEPT EMP,DBUDG FROM DEPT WHERE
                 DEPT.PJNO = PROJECTS.PJNO AND
                 PROJECTS.LOC = 'SJ');
```

Step 3. Normalize DEPTSJ to create a degenerate hierarchy (i.e. a relation).

```
T1 <- SLICE(DNO,MGR,PJNO,CEST,CACT,MNO,UF FROM DEPTSJ);
```

Step 4. From T1, group the project cost data with the department data and subordinate the department data to project number.

```
PRDEPT(PJNO,(DEPT(DNO,MGR,CEST,CACT))) <- ELIM_DUP(
     CONSOLIDATE(SELECT(PJNO,DNO,MGR,CEST,CACT FROM T1)));
```

Step 5. From T1 group equipment data by project number.

```
PREQUIP(PJNO,(EQUIP(MNO,(UF))))<-CONSOLIDATE(SELECT(PJNO,MNO,UF FROM T1));
```

Step 6. Compute the total utilization factor for each machine according to project.

```
PREQTUF(PJNO,(EQUIP(MNO,TUF))) <-
     SELECT(PJNO,MNO,SUM(UF FOR EACH EQUIP) FROM PREQUIP);
```

Step 7. Compute the total cost spent by all departments on a given project.

```
PROJTCOST(PJNO,TCOST) <-
     SELECT(PJNO,SUM(CACT FOR EACH PRDEPT) FROM PRDEPT);
```

Step 8. Generate a new Form where a new Field, CMARG = CEST-CACT, replaces CEST and CACT.

```
PRDEPT2(PJNO,(DEPT(DNO,MGR,CMARG))) <-
     SELECT(PJNO,DNO,MGR,CEST-CACT FROM PRDEPT);
```

In the above statement, the arithmetic expression CEST-CACT is assigned to CMARG since CMARG is the fourth Field in the target specification, and CEST-CACT is the fourth expression in the select list.

Step 9. Create a new Form from PRDEPT2 in which a "budget status" field is added which flags those departments which are over budget.

```
PRDEPT3(PJNO,(DEPT(DNO,MGR,CMARG,BUDSTAT))) <-
     CASE(FROM PRDEPT2 WHEN DEPT.CMARG <
       0: SELECT(ALL EXCEPT BUDSTAT <- '* OB *'),
  OTHERS: SELECT(ALL EXCEPT BUDSTAT <- ' '));
```

Step 10. Find out more information about the department managers from the EMPLOYEE Form.

```
MGREMPS <- SELECT(ENO,NAME,PHONE FROM EMPLOYEE
                  WHERE ENO = DEPTSJ.MGR);
```

Step 11. Add the information found about the department managers to the department data in PRDEPT3.

```
PRDEPT4 <-
  SELECT(ALL EXCEPT MGR FROM
     GRAFT(MGREMPS ONTO PRDEPT3 AT DNO WHERE
           MGREMPS.ENO = PRDEPT3.MGR));
```

Step 12. Combine the total cost data, the department data, the equipment data, and the project data. Omit the project location field, since it is not required and sort the result on PJNO, DNO.

```
REPORT <-
  SORT(SELECT(ALL EXCEPT LOC FROM
     GRAFT(PROJTCOST,PRDEPT4,PREQTUF ONTO PROJECTS
           WHERE PROJTCOST.PJNO = PROJECTS.PJNO,
                 PRDEPT4.PJNO = PROJECTS.PJNO,
                 PREQTUF.PJNO = PROJECTS.PJNO))
        BY PJNO,DNO);
```

Step 13. Print the report.

```
PRINT REPORT;
```

In this application example, the program was broken into multiple statements for the sake of clarity. However, some statements could have been nested (except FORM). For example, Steps 10 and 11 would combine to give:

```
PRDEPT4 <-
  SELECT(ALL EXCEPT MGR FROM
   GRAFT( MGREMPS: SELECT(ENO,NAME,PHONE FROM EMPLOYEE
                   WHERE EMPLOYEE.ENO = DEPTSJ.MGR),
          ONTO PRDEPT3 AT DNO
          WHERE MGREMPS.ENO = PRDEPT3.MGR));
```

As in SEQUEL [2], a label (e.g. MGREMPS) must be used when a reference to a name in an intermediate operation is ambiguous. In this example, MGREMPS is not required since ENO and MGR are distinct. Label definitions are local to the Form operation which contains them.

5. Summary and Conclusions

In this paper we assert that the hierarchical view of data will continue to be popular for a broad class of applications and users. In particular, some of these applications require complex data manipulation which, heretofore, has been dealt with procedurally. In this light, a nonprocedural language, CONVERT, is proposed as a high-level DBMS interface. CONVERT is meant to provide users with a tool for performing complex data manipulation and query of hierarchical data abstractions, called "Forms".

In section 2, we present the definitions and terms relevant to the Form data model. Section 3 describes the CONVERT language in some detail and gives a number of examples. A previous version of CONVERT was described in the context of data translation [1]. Section 3 gives further semantic detail and describes additional features required due to the change of emphasis, and language revisions resulting from continued developement. Section 4 gives a nontrivial application scenario, and demonstrates the data restructuring facilities of CONVERT by presenting a program which derives a user report from the "native" data base.

The "Form" data model presents a general but conceptually simple view of hierarchical data structures. Similarly, we believe the basic Form-operations while providing powerful manipulation capabilities, are at the same time, conceptually easy to understand. Herein, lies the attractiveness of the language. Currently, we are implementing CONVERT as a data translator and later hope to test the language on real users.

ACKNOWLEDGEMENTS

The authors wish to thank Vincent Lum for his encouragement in pursuing the extended role of the CONVERT language. We also thank Robert W. Taylor, Paul McJones, and Jim Gray for their many helpful suggestions and comments.

REFERENCES

[1] Shu, N. C., Housel, B. C., Lum, V. Y., "CONVERT: A High Level Translation Definition Language for Data Conversion," CACM, October, 1975.

[2] Chamberlin, D. D. and Boyce, R. F., "SEQUEL: A Structured English Query Language." Proc. of 1974 ACM SIGFIDET Workshop, Ann Arbor, Michigan, April 1974.

[3] Boyce, R. F., et. al., "Specifying Queries as Relational Expressions," Proc. of ACM SIGPLAN/SIGIR Interface Meeting, Gaithersburg, Md., Nov., 1973.

[4] Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif., November, 1971.

[5] Held, G., et. al., "INGRES - A Relational Data Base System," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975.

[6] Codd, E. F., "A Relational Model of Data For Large Shared Data Banks," Comm. ACM 13,6 (June, 1970) 377-387.

[7] Fehder, P. F., "HQL: A Set-Oriented Translaction Language for Hierarchically-structured Data Bases," Proc. ACM National Conference, San Diego, November, 1974.

[8] System 2000 Reference Manual, MRI Systems Corporation, Austin Texas, August 1974.

[9] McGee, W. C., "File-level Operations on Network Data Structures," Proc. ACM SIGMOD Conference, San Jose, Ca., May 1975.

[10] IMS/360, Version 2, System/Application Design Guide, IBM SH20-0910-3, 1972.

[11] Schmid H. A., Swenson, J. R., "On the Semantics of the Relational Data Model," Proc. of ACM SIGMOD Conference, San Jose, Ca., May 1975.

[12] Housel, B. C., et. al, "DEFINE - A Nonprocedural Data Description Language for Defining Information Easily," Proc. ACM Pacific 75 Symposium, San Francisco, Ca., April 1975.

[13] Hardgrave, W. T., "BOLTS: A Retrieval Language for Tree Structured Data Base Systems," Systems Information COINS IV, pp. 137-158, Plenum Press, New York, 1974.

APPENDIX - Intermediate Results for Steps in Sample Application in Section 4.

(Step 2)

DEPTSJ						
		PROJ				
					EQUIP	
DNO	MGR	PJNO	CEST	CACT	MNO	UF
D1	47	J4	50	43	M5	.2
		J3	70	80	M8	.3
					M15	.5
		J2	20	27	M2	.4
D3	18	J3	63	50	M10	.8
					M15	.1
		J2	170	177	M10	.2
					M8	.3
D2	98	J3	40	70	M8	.2
					M15	.1
D4	20	J4	90	160	M5	.7
					M6	1.0
		J2	40	37	M2	.5
					M3	1.0

(Step 3)

T1						
DNO	MGR	PJNO	CEST	CACT	MNO	UF
D1	47	J4	50	43	M5	.2
D1	47	J3	70	80	M8	.3
D1	47	J3	70	80	M15	.5
D1	47	J2	20	27	M2	.4
D3	18	J3	63	50	M10	.8
D3	18	J3	63	50	M15	.1
D3	18	J2	170	177	M10	.2
D3	18	J2	170	177	M8	.3
D2	98	J3	40	70	M8	.2
D2	98	J3	40	70	M15	.1
D4	20	J4	90	160	M5	.7
D4	20	J4	90	160	M6	1.0
D4	20	J2	40	37	M2	.5
D4	20	J2	40	37	M3	1.0

(Step 4)

PRDEPT				
	DEPT			
PJNO	DNO	MGR	CEST	CACT
J2	D1	47	20	27
	D3	18	170	177
	D4	20	40	37
J3	D1	47	70	80
	D2	98	40	70
	D3	18	63	50
J4	D1	47	50	43
	D4	20	90	160

(Step 5)

PREQUIP		
	EQUIP	
PJNO	MNO	(UF)
J2	M2	.4
		.5
	M3	1.0
	M8	.3
	M10	.2
J3	M8	.3
		.2
	M10	.8
	M15	.5
		.1
		.1
J4	M5	.2
		.7
	M6	1.0

(Step 6)

PREQTUF		
	EQUIP	
PJNO	MNO	TUF
J2	M2	.9
	M3	1.0
	M8	.3
	M10	.2
J3	M8	.5
	M10	.8
	M15	.7
J4	M5	.9
	M6	1.0

(Step 7)

PROJTCOST	
PJNO	TCOST
J2	241
J3	200
J4	203

(Steps 8 and 9)

PRDEPT3				
	DEPT			
PJNO	DNO	MGR	CMARG	BUDSTAT
J2	D1	47	-7	* OB *
	D3	18	-7	* OB *
	D4	20	3	
J3	D1	47	-10	* OB *
	D2	98	-30	* OB *
	D3	18	13	
J4	D1	47	7	
	D4	20	-70	* OB *

(Step 10)

MGREMPS			
ENO	DNO	NAME	PHONE
18	D3	LEWIS	6673
20	D4	CHU	3348
47	D1	KING	6397
98	D2	PARKS	6967

(Step 11)

PRDEPT4					
	DEPT				
PJNO	DNO	NAME	PHONE	CMARG	BUDSTAT
J2	D1	KING	6397	-7	* OB *
	D3	LEWIS	6673	-7	* OB *
	D4	CHU	3348	3	
J3	D1	KING	6397	-10	* OB *
	D2	PARKS	6967	-30	* OB *
	D3	LEWIS	6673	13	
J4	D1	KING	6379	7	
	D4	CHU	3348	-70	* OB *