SQL/NF: A QUERY LANGUAGE FOR ¬1NF RELATIONAL DATABASES

Mark A. Roth, Henry F. Korth and
Don S. Batory

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-19 September 1985

SQL/NF: A Query Language for ¬1NF Relational Databases

Mark A. Roth
Henry F. Korth †
Don S. Batory ‡

Department of Computer Science
University of Texas at Austin

Abstract

There is growing interest in abandoning the first-normal-form assumption on which the relational database model is based. This interest has developed from a desire to extend the applicability of the relational model beyond traditional data-processing applications. In this paper, we extend one of the most widely used relational query languages, SQL [C+] to operate on non-first-normal-form relations as defined in [RKS2]. In this framework, we allow attributes to be relation-valued as well as atomic-valued (e.g., integer or character). A relation which occurs as the value of an attribute in a tuple of another relation is said to be *nested.* Our extended language, called SQL/NF, includes all of the power of standard SQL as well as the ability to define nested relations in the data definition language and query these relations directly in the extended data manipulation language. A variety of improvements are made to SQL; the syntax is simplified and useless constructs and arbitrary restrictions are removed.

1. Introduction

User interfaces to document libraries, catalogs, or other tabular data are constrained to an undesirable extent by requiring values that appear in relations to be atomic. This atomicity is imposed by the assumption that relations are in first normal form (1NF). Commercial relational database systems have deviated only slightly from the assumption of first normal form by allowing string matching operations on character string fields within a relation. In this paper, we take the much larger step of allowing attributes to be relation-valued as well as atomic-valued. A relation which occurs as the value of an attribute in a tuple of another relation is said to be *nested.* In [RKS2], we defined a formal predicate-calculus-based language for dealing with such non-first-normal-form (¬1NF) relations. That language defines a minimal degree of power that we expect from any language designed to operate on nested relations.

For real-world users of a database system, however, a terse predicate-calculus language is too difficult to use. These considerations have lead to the definition of several "syntactically-sugared" query languages such as SQL, Query-By-Example, and QUEL. In this paper, we extend one of the most widely-used of these languages, SQL [C+], to operate on a database of nested relations. Most of our extensions pertain to the data manipulation part of SQL, although we extend also the SQL data definition language to permit the definition of ¬1NF databases. In defining SQL/NF, an important goal was to retain the "spirit" of the existing SQL language so as to reduce the effort required on the part of existing SQL users to learn SQL/NF. Roughly speaking, wherever a constant or scalar-valued variable may appear in SQL, a relation or expression evaluating to a relation may appear in SQL/NF. We introduce new commands to transform a relation to

† Research partially supported by an IBM Faculty Development Award and NSF Grant DCR-8507224

‡ Research supported by NSF grant MCS-83-17353

an equivalent nested one (the *nest* operation) and to transform a nested relation into a less-nested one (the *unnest* operation). We shall assume that the reader is familiar with standard SQL as described in [C+] or in most standard database texts.

Although we retain most constructs of the SQL langauge, we would be remiss if we did not make some obvious improvements in the SQL langauge. Some of these changes are due to the availability of nested relations. For instance, difficult SQL queries involving `GROUP BY` and `HAVING` can be eliminated in favor of rather straightforward queries on properly structured ¬1NF relations. Other changes are simply to correct some mistakes made in the design of SQL. In Date's critique of the SQL language [Dat1], a good case is made for requiring certain modifications to the SQL language. One of the driving forces behind the critique is the language design maxim, the *principle of orthogonality*. This principle requires separate treatment for distinct concepts, and similar treatment for similar concepts [Dat2]. The following definition of the SQL/NF language takes into account this principle, directly incorporating some of the modifications proposed in [Dat1]. We also follow where possible the proposed standard relational database language [X3H2], which already incorporates some the changes suggested here, although relying on a 1NF database model.

Before we present the SQL/NF language itself, let us introduce a sample database, based on the one used in [C+], variations of which we will use throughout this paper. Our example differs from that in [C+] in that we are including nested relations, while [C+] uses first-normal-form relations. We have a corporation relation, *Corp*, with attributes department number (*dno*), name (*dname*), location (*loc*), employees (*Emp*), and parts used (*Usage*). The employee relation in each *Corp* tuple has attributes employee number (*empno*), name (*name*), salary (*sal*), manager's employee number (*mgr*), and children (*Children*). The *Children* relation in each employee tuple has attributes child's name (*name*), and date of birth (*dob*). The usage relation in each *Corp* tuple is a unary relation with attribute part number (*part*). Finally, we have a supply relation with attributes supplier number (*supplier*), and a unary relation supplies (*Supplies*) of part numbers (*part*). An example database is shown in Figure 1-1. Note that for notational convenience we eliminated the set braces which would normally mark off each nested relation.

We could think of this database as a view of an underlying 1NF database which we could obtain by join and nest operations. However, we propose that ¬1NF relations be definable and exist in their own right in the database, thus eliminating costly restructuring operations otherwise needed to give the user a nested view which can be more natural and easier-to-query. The advantages of our sample database design are apparent. We have only two relations rather than the five required in a normalized, atomic-valued database. The one-to-many relationships are more clearly visible to the user, and the complexity of queries will be reduced since fewer joins will be needed, and many useful data groupings already exist, such as employees of each department. If, indeed, the 1NF relations are needed, then a sequence of unnest and projection operations can be applied.

The remainder of this paper is organized as follows. Section 2 presents an overview of the ¬1NF model. We indicate previous work in this area and present the basic concepts involved. Sections 3–5 contain our definition and description of the SQL/NF language. We define the query facilities, the data manipulation language and the data definition language. Host language support is beyond the scope of this paper. Section 6 compares our language with some previous attempts at defining a high-level query language for ¬1NF models. We also provide a BNF definition of our language in the Appendix.

2. The ¬1NF Relational Model

Various researchers have studied the effect of dropping the assumption that all relations in a relational

Corp

dno	dname	loc	Emp						Usage
			empno	name	sal	mgr	Children		part
							name	dob	
10	Mfg	Austin	5	Jones	5000	10	Sam	2/2/80	7
							Sue	3/5/81	8
			6	Smith	4000	5	Bill	9/3/70	9
			10	Cane	2000				
			12	Niven	3000	5	Bill	8/8/72	
							Mike	8/2/73	
			20	Roth	9000	10	David	5/4/84	
20	Pers	Dallas	16	Adams	2000	21			8
			21	Garza	9000		Joe	5/7/83	10
			23	Beal	3000	21	Pat	5/8/60	20
							Ann	9/1/65	32
							Beth	8/3/66	34
									38
									41
30	Retail	Austin	4	Davis	5000	17	Bill	2/6/75	3
			17	Dale	6000				5
			24	Soo	1000	4	Bill	6/8/80	9
							Will	6/8/80	10
							Jill	6/8/80	32

Supply

supplier	Supplies
	part
42	7
	8
	9
	10
	18
	20
	21
45	8
	10
	32
	34
	38
56	3
	5
	10
	41

Figure 1-1. Sample database.

database must be in first-normal-form (1NF). Early work was done by Makinouchi [Mak] and led to the concept of nesting. This concept was later studied by Jaeschke and Schek [JS] for one level nesting over single attributes and by Thomas and Fischer [TF] in a more general setting. Utilizing ¬1NF relations for structuring database outputs was discussed by Kambayashi, et al. [KTT], while Fischer and Van Gucht [FV1, FV2] looked at dependencies which characterize ¬1NF relations.

Özsoyoğlu and Özsoyoğlu [OO] consider operations similar to that of [JS], and extend the basic algebra for relations by aggregate operators. Özsoyoğlu and Yuan [OY] introduce *nested normal form* for ¬1NF relations.

Our previous work [RKS2] defines a relational calculus and relational algebra for ¬1NF relations and proves their equivalence. We also introduced *partitioned normal form* for ¬1NF relations (described later) which is equivalent to *scheme trees* of [OY] and *formats* of [AB]. Abiteboul and Bidoit [AB] also define some extended operators which are refined in [RKS2]. Others [Jae2, Jae3, PHH, Sch2, ScS1, ScS2] have been developing languages and implementations for ¬1NF relational databases.

2.1 Basic Concepts

A ¬1NF relation may have attributes which are relation-valued as well as atomic-valued. Thus we expand our notation for defining schemes as follows. A *database scheme* S is a collection of rules of the form $R_j = (R_{j_1}, R_{j_2}, \ldots, R_{j_n})$. The objects R_j, R_{j_i} are attributes. R_j is a *higher order attribute* if it appears on the left hand side of some rule; otherwise it is *zero order*. The names on the right hand side of rule R_j form a set denoted E_{R_j}, the elements of R_j. As with any set, attributes on the right hand side of the same rule are unique, and to avoid ambiguity, we require that no two rules can have the same name on the left hand side.

Employee

ename	Children		Skills		
	name	dob	type	Exams	
				year	city
Smith	Sam	2/10/84	typing	1984	Atlanta
	Sue	1/20/85		1985	Dallas
			dictation	1984	Atlanta
Watson	Sam	3/12/78	filing	1984	Atlanta
				1975	Austin
				1971	Austin
			typing	1962	Waco

Figure 2-1. A sample relation on the Emp scheme.

Example 2.1: Consider a somewhat simpler version of the corporation example used in the introduction. The scheme is

$$\begin{aligned} \text{Emp} &= (\text{ename, Children, Skills}), \\ \text{Children} &= (\text{name, dob}), \\ \text{Skills} &= (\text{type, Exams}), \\ \text{Exams} &= (\text{year, city}). \end{aligned}$$

In this scheme each employee has a set of children each with a name and birthdate, and a set of skills, each with a skill type and a set of exam years and cities, when and where the employee retested his proficiency at the skill. A sample relation is shown in the relation in Figure 2-1. □

In this example, the higher order attributes are *Emp*, *Children*, *Skills* and *Exams*. All others are zero order attributes. We will generally use capitalized names for higher order attributes and uncapitalized names for zero order attributes.

A 1NF database scheme is a collection of rules of the form $R_j = (R_{j_1}, R_{j_2}, \ldots, R_{j_n})$ where all the R_{j_i} are zero order. ¬1NF schemes may contain any combination of zero or higher order attributes on the right hand side of the rules as long as the scheme remains nonrecursive. Note, a nested relation is represented simply as a higher order attribute on the right hand side of a rule.

In the relation of Figure 2-1, we would not expect two tuples with employee = 'Smith' since all of Smith's children and skills should be grouped into one tuple. That is, there is no significance in separate groups of children or skills. This led us to restrict the set of ¬1NF relations to those that are in *partitioned normal form* (PNF).

A relation is in PNF if the zero order attributes in the relation form a key for the relation, and each nested relation is also in PNF, that is, the zero order attributes of each nested relation form a key for that relation. The employee relation in Figure 2-1 is in PNF. Note that *ename* is a key for *Employee*, *type* is a key for each *Skills* relation, and (*Year*, *City*) is a key for each *Exams* relation.

2.2 Operators

In this section, we discuss operators for ¬1NF relations. The traditional relational algebra operators can be used with ¬1NF relations with only minor modifications [RKS2]. With the addition of *unnest* and *nest* operators, this algebra is as powerful as the standard relational algebra working on 1NF relations. Formal

Employee-U

ename	name	dob	Skills		
			type	Exams	
				year	city
Smith	Sam	2/10/84	typing	1984	Atlanta
				1985	Dallas
			dictation	1984	Atlanta
Smith	Sue	1/20/85	typing	1984	Atlanta
				1985	Dallas
			dictation	1984	Atlanta
Watson	Sam	3/12/78	filing	1984	Atlanta
				1975	Austin
				1971	Austin
			typing	1962	Waco

Figure 2-2. The Employee relation unnested along Children.

definitions of *nest* and *unnest* can be found in [RKS2, TF]. Intuitively, the *unnest* operator (μ) transforms a relation into one which is less deeply nested by concatenating each tuple in the relation being unnested to the remaining attributes in the relation. Figure 2-2 shows the result of unnesting the relation in Figure 2-1 along the *Children* attribute. This corresponds to the algebra expression

$$\text{Employee-U} = \mu_{\text{Children}}(\text{Employee})$$

The *nest* operator (ν) partitions relations and creates nested relations for each partition formed. To return to the Employee-U relation back to the original relation we would use

$$\text{Employee} = \nu_{\text{Children}=(\text{name,dob})}(\text{Employee-U})$$

3. Query Facilities

In SQL, a basic query conforms to the structure

```
SELECT attribute-list
FROM   relation-list
WHERE  predicate
```

This SFW-expression can be conceptually executed by forming the cartesian product of all relations in the *relation-list*, choosing only tuples in this product that satisfy the *predicate*, and then choosing only those attributes in the *attribute-list*. If no qualification of tuples is needed then the WHERE clause can be omitted. If all attributes of the relations in the *from-list* are desired then SQL allows the use of "*" instead of actually specifying all attributes in the *attribute-list*. In the proposed standard relational database language (hereafter called RDL), the keyword ALL is used instead of "*" [X3H2].

The obvious way to access the entire contents of a relation would be to simply state the relation name. However, in SQL, we have to use

```
SELECT *
FROM   relation-name
```

In SQL/NF, we allow free substitution of *relation-name* for "SELECT * FROM *relation-name*" and adopt the RDL change substituting ALL for "*". Consider the 1NF database in Figure 3-1. The department (*Dept*) relation has three attributes, department number (*dno*), department name (*dname*), and location (*loc*). The

Dept

dno	dname	loc
10	Manufacturing	Austin
20	Personnel	Dallas
30	Retail	Austin
⋮	⋮	⋮

Emp

eno	ename	dno	sal
13	Smith	10	20000
33	Jones	30	14000
34	Adams	10	15000
48	Miller	10	40000
⋮	⋮	⋮	⋮

Figure 3-1. A sample 1NF database.

employee (*Emp*) relation has four attributes, employee number (*eno*), employee name (*ename*), department number of department in which employee works (*dno*), and salary (*sal*). The query to get employee data for employees in department 10 is

```
SELECT ALL
FROM   Emp
WHERE  dno = 10
```

or using our simplified notation,

```
Emp WHERE dno = 10
```

To get departments which have at least one employee, the query is

```
SELECT ALL
FROM   Dept
WHERE  EXISTS (SELECT ALL
               FROM   Emp
               WHERE  Dept.dno = Emp.dno)
```

or, in SQL/NF,

```
Dept WHERE EXISTS (Emp WHERE Dept.dno = Emp.dno)
```

This last query easily paraphrases as: Get department tuples where there exists employee tuples where the department numbers are the same. The same can not be said about the strict SQL version. In fact it is not clear why we are selecting any attributes at all in the EXISTS subquery, since our goal is not to actually extract any information from the employee relation but rather to test for its existence.

3.1 Nested Expressions

> "A language should provide, for each class of object it supports, a general, recursively defined syntax for expressions that exploits to the full any closure properties the object class may possess" [Dat1; 12].

The primary objects a relational database language supports are scalar (atomic) values and relations. In 1NF databases, each relation is comprised strictly of scalar values. In ¬1NF databases, each relation may be comprised of other relations as well as scalar values. The principle of orthogonality has been usefully employed in defining the ¬1NF data structure. Wherever a scalar value could occur in a 1NF relation, a relation can now occur. This simple transformation is also employed in our definition of the data sublanguage. SQL has the *closure* property where the result of any query on one or more relations is itself a relation. The principle of orthogonality suggests that we should allow an SFW-expression wherever a relation name could exist. In SQL this means allowing SFW-expressions in the FROM clause.

The first use of such a modification is the building of incremental queries. Using the database of Figure 3-1, consider the query: Get names of employees who work in department 5. The first step a user may

recognize is the the need to join the Emp and Dept relations on dno. So he forms the query

```
SELECT ALL
FROM   Emp, Dept
WHERE  Emp.dno = Dept.dno
```

Then from this relation get names of employees in department 5 producing

```
SELECT ename
FROM   (SELECT ALL
        FROM   Emp, Dept
        WHERE  Emp.dno = Dept.dno)
WHERE  dno = 5
```

An SQL/NF–level query optimizer could then translate this query into the equivalent query

```
SELECT ename
FROM   Emp, Dept
WHERE  Emp.dno = Dept.dno
AND    dno = 5
```

A more useful example involving nested expressions in the FROM clause involves the UNION operator. UNION is an infix operator in SQL and is used in the form

SFW-expression UNION *SFW-expression*

Note that, in SQL, one must use SFW-expression's with UNION and not relations. To illustrate, suppose we have two employee relations (as in Figure 3-1). The *Emp-exec* relation contains executive level employees and the *Emp-other* relation contains all other employees. If we want to get all employees, we write the SQL query

```
SELECT *
FROM   Emp-exec
UNION
SELECT *
FROM   Emp-other
```

In SQL/NF, we can write the simpler query

Emp-exec UNION Emp-other

Now, if we modify our query so that we get all employees who make more than $35,000, then we must add a WHERE clause to each SFW-expression in the SQL query.

```
SELECT *
FROM   Emp-exec
WHERE  sal > 35000
UNION
SELECT *
FROM   Emp-other
WHERE  sal > 35000
```

Using SQL/NF, we can form the union first, place that expression in the FROM clause, and qualify all tuples with one WHERE clause.

```
SELECT ALL
FROM   (Emp-exec UNION Emp-other)
WHERE  sal > 35000
```

Nested expressions are even more applicable when using a ¬1NF database. Since attributes may now be relation valued, relation names may occur in the SELECT clause of a query, and so, under the principle of orthogonality, we allow SFW-expressions in the SELECT clause. For the following examples we will use the ¬1NF database in Figure 3-2, in which we have combined the data of the Dept and Emp relations used in

Company

dno	dname	loc	Emps		
			eno	ename	sal
10	Manufacturing	Austin	13	Smith	20000
			34	Adams	35000
			48	Miller	40000
20	Personnel	Dallas			
30	Retail	Austin	33	Jones	14000
⋮	⋮	⋮	⋮		

Figure 3-2. A sample ¬1NF database.

Figure 3-1. The company (*Company*) relation has four attributes, department number (*dno*), department name (*dname*), location (*loc*), and employees (*Emps*). Each *Emps* relation has three attributes, employee number (*eno*), employee name (*ename*), and salary (*sal*). Recall that for notational simplicity, we eliminate the set braces which normally would occur around each *Emps* relation in Figure 3-2. Consider a query to get department names and the employees in each department making more than $35,000. First consider getting all employees. The query is

```
SELECT dname, Emps
FROM   Company
```

Since Emps is a relation we could, equivalently write

```
SELECT dname, ( SELECT ALL
                FROM   Emps)
FROM   Company
```

Now to limit employees to those making more than $35,000, it is a simple matter of adding a WHERE clause to the nested SFW-expression, giving

```
SELECT dname, ( SELECT ALL
                FROM   Emps
                WHERE  sal > 35000)
FROM   Company
```

or

```
SELECT dname, (Emps WHERE sal > 35000)
FROM   Company
```

Note that a SFW-expression may produce an empty relation. In the last query the "Personnel" tuple will have an empty Emps relation since it was empty to begin with, and the "Retail" tuple will have an empty Emps relation since none of its employees satisfy the "sal > 35000" predicate. To eliminate tuples with empty *Emps* relations in the result we can write

```
SELECT dname, (Emps WHERE sal > 35000)
FROM   Company
WHERE  EXISTS(Emps WHERE sal > 35000)
```

Later, we introduce a technique for referencing the new *Emps* relation the first time it is mentioned, avoiding the duplicate specification of the nested query.

Not only can we select specific tuples from a nested relation we can also select specific attributes. To illustrate, consider the query to get department names and locations and employee names and salaries. We write

```
SELECT dname, loc, (SELECT ename, sal
                    FROM   Emps)
FROM   Company
```

Combining the above techniques, we can write the following query. Get department names and locations and employee names and salaries where the location is 'Austin' and and the employee salary is more than $35,000.

```
SELECT dname, loc, ( SELECT ename, sal
                     FROM   Emps
                     WHERE  sal > 35000)
FROM   Company
WHERE  loc = 'Austin'
```

3.2 Functions

> In SQL, "the argument to a function such as SUM is a column of scalar values and the result is a single scalar value; hence, orthogonality dictates that (a) any column-expression should be permitted as the argument, and (b) the function-reference should be permitted in any context in which a scalar can appear. However, (a) the argument is in fact specified in a most unorthodox manner, which means in turn that (b) function references can actually appear only in a very small set of special-case situations" [Dat1; 20].

Date's arguments are even more valid when we assume a ¬1NF model. Here we have built-in sets of values in the form of nested relations and it would make more sense to apply functions to relations rather than artificially applying them to attributes. Then, by the principle of orthogonality, we should be able to apply functions to any expression that evaluates to a relation.

Consider first the 1NF database of Figure 3-1, and a query to find the total amount made by all employees. In SQL, we would write

```
SELECT SUM(sal)
FROM   Emp
```

The argument to SUM is actually the entire *sal* column of *Emp*, whereas a reference to *sal* in a WHERE clause (e.g., sal > 5000), is refering to individual *sal* values. Therefore, we adopt Date's suggestion to apply functions to their actual argument. Thus, our query becomes

```
SUM(SELECT sal
    FROM   Emp)
```

Another example is the query which gets all departments that employ more than 10 people:

```
SELECT dno
FROM   Dept
WHERE  COUNT(SELECT *
             FROM   Emp
             WHERE  Dept.dno = Emp.dno)
       > 10
```

or using our simplified notation which substitutes "Emp" for "SELECT * FROM Emp":

```
SELECT dno
FROM   Dept
WHERE  COUNT(Emp WHERE Dept.dno = Emp.dno) > 10
```

In SQL, the latter query would usually be formulated using GROUP BY and HAVING.

```
SELECT dno
FROM   Dept
WHERE  dno IN
       (SELECT   dno
        FROM     Emp
        GROUP BY dno
        HAVING COUNT(DISTINCT eno) > 10)
```

GROUP BY introduces a new structure into the relational model: partitioned relations. The only attributes which can be selected from a partitioned relation are the "group by" attributes, i.e., those that have the same value for each partition, and single-valued functions of any attribute. Normally, a function operates on the entire relation, but when a relation is partitioned, the function is applied separately to each partition. Thus, we have a new structure which, incidentally, is not in 1NF, with new rules for the execution of SFW-expressions, and a new HAVING clause to test predicates on partitions.

In some cases GROUP BY and HAVING are not necessary. An example of this is when the values of the functions are not being retrieved in a SELECT clause. For example, a legal SQL query to do the last query is

```
SELECT dno
FROM   Dept
WHERE  10 < (SELECT COUNT(*)
             FROM   Emp
             WHERE  Emp.dno = Dept.dno)
```

Furthermore, if we allow nested queries in the SELECT clause then GROUP BY and HAVING are totally unnecessary. For example, to retrieve the counts of employees for each department we could write

```
SELECT dno, COUNT(Emp WHERE Emp.dno = Dept.dno)
FROM   Dept
```

Now let us consider the ¬1NF database in Figure 3-2. Since employees have already been "grouped by" department, our queries are easier to formulate. To get the employee counts, we write

```
SELECT dno, COUNT(Emps)
FROM   Company
```

To get departments where the employee count is more than 10, we write

```
SELECT dno
FROM   Company
WHERE  COUNT(Emps) > 10
```

By structuring relations appropriately, we can turn any GROUP BY/HAVING query into a straightforward SFW-expression. Since these types of queries are some of the hardest to formulate in SQL, and operate under a different set of rules from standard SQL queries, their elimination is a major advantage of the ¬1NF model.

A further advantage of using relations or nested expressions as input to functions is the ability to use multi-attribute relations and have the function apply to several attributes simultaneously. For example, suppose we have a *Sales* relation with employee number (*eno*) and 12 sales attributes (*Jan-sales, Feb-sales, ..., Dec-sales*) showing total sales for each month of the year for the employee. Then to get the total of all sales in each month we can write

```
SUM(SELECT Jan-sales, Feb-sales, Mar-sales, Apr-sales, May-sales, Jun-sales,
           Jul-sales, Aug-sales, Sep-sales, Oct-sales, Nov-sales, Dec-sales
    FROM   Sales)
```

The SUM function is applied to each column of the argument relation. In general, a *column* function, (SUM, AVG, MAX, MIN), reduces a relation to a single tuple with the same number of attributes, by applying the function to each column of the relation. A *table* function (COUNT), reduces a relation to a single tuple with one attribute. Thus, the result of applying a function is always a single tuple relation.

3.3 Null Values and Operations Dealing with Nulls

One question that usually arises when dealing with functions concerns the presence of null values. SQL makes the decision to ignore null values in all functions except COUNT. An unfortunate consequence of this is that the equality of AVG(*Rel*) * COUNT(*Rel*) and SUM(*Rel*) may be violated. We believe that nulls should not be ignored in any function, rather they should, when appropriate, produce an error. This forces the user to remove the nulls before applying the function and also prevents him from believing he has received a precise answer to a query which is, in fact, based on imprecise data.

A thorough treatment of nulls for ¬1NF databases can be found in [RKS1]. One of the functions which is usually required when dealing with null values is a method for eliminating *subsumed* tuples in a relation. A tuple t is *subsumed* by tuple q if t's non-null attributes have the same values as the corresponding attributes in q. For example, the tuple $t =<$ Smith, NULL, NULL $>$ is subsumed by $<$ Smith, 10, NULL $>$ and by $<$ Smith, 20, 15000 $>$, but not by $<$ Jones, NULL, 15000 $>$. Subsumed tuples are like duplicate tuples in that they do not provide any more information than some other tuple in the relation. When nested relations are attributes the definition is applied recursively, so that relation r subsumes relation s, if every tuple in s is subsumed by some tuple in r.

Although SQL eliminates duplicate tuples via the SELECT DISTINCT construct, it does not eliminate subsumed tuples, even though null values are allowed. Therefore, we introduce the SUBSUME function to eliminate subsumed tuples from a relation. Note, that SUBSUME also removes duplicate tuples, since by definition if $t = q$ then t subsumes q and q subsumes t. We also use our standard notation for applying a function for the syntax of DISTINCT and SUBSUME.

To eliminate duplicates from the Company relation we use

```
DISTINCT(Company)
```

To get department names and employees names and salaries, eliminating subsumed employee tuples, we use

```
SELECT dname, SUBSUME(SELECT ename, sal
                      FROM   Emps)
FROM   Company
```

If we want to eliminate duplicates before counting the number of tuples in the Company relation, our query is

```
COUNT(DISTINCT(Company))
```

Another important operation which becomes available when null values are supported is the *outer join*. The outer join is similar to a traditional join, except that tuples which normally would not participate in the join are added to the result. Null values are used for the attributes not in the relation. In [Dat3], a proposal is made for supporting the outer join operation with a PRESERVE clause. All tuples of the relations specified in the PRESERVE clause are included in the resulting relation even if they do not satisfy the predicates of the WHERE clause. The attributes of the resulting relation which are not in the "preserved" relation are set to NULL for those tuples which did not satisfy the WHERE clause.

For example, to join the *Dept* and *Emp* relations in our 1NF database, without losing the department data for departments that do not have any employees, we would use the PRESERVE clause as follows.

```
SELECT   *
FROM     Dept, Emp
WHERE    Dept.dno = Emp.dno
PRESERVE Dept
```

Finally, a clarification of the relationship between empty relations and null values is in order. For reasons discussed in [RKS1], we note that the empty relation is equivalent to any relation in which all attributes of all tuples have null values for the atomic attributes and, recursively, empty relations for the nested relations. Under subsumption, all of these relations are equivalent to a single tuple relation, where the value of each attribute is null or empty. Since we have a single type of null in SQL/NF, we assume the most general interpretation, that is, the *no-information* interpretation. This means we do not know whether or not an actual value exists which could replace this null. Since empty relations are equivalent to a relation with null-tuples, we assign the no-information interpretation to empty sets as well. We specifically forbid the *non-existent* interpretation for empty relations (see [RKS1]).

3.4 Miscellaneous Features

3.4.1 Unnesting after a Function

When a column or table function is applied to a nested relation it doesn't make sense to retain the relation structure for a single tuple. Therefore, our functions will cause the relation in which it occurs to be unnested one level. For example, instead of the result of our query to get department numbers and the number of employees in each department having tuples $\{< 10, \{3\} >, < 20, \{0\} >, < 30, \{1\} >, \ldots\}$, we would have $\{< 10, 3 >, < 20, 0 >, < 30, 1 >, \ldots\}$. This feature also allows easier application of multiple functions. For instance, to get the total number of employees in the company from our ¬1NF database we would write

```
SUM(SELECT COUNT(Emps)
    FROM   Company)
```

Without the COUNT function unnesting its sets the SUM function would get sets of counts as arguments and would not work properly.

3.4.2 Attribute Lists

Sometimes it is easier to list the attributes you do not want to deal with. For this, SQL/NF allows the construct "ALL BUT *attribute-list*". Recall a previous example in which we were interested in getting the total sales in each month from a *Sales* relation with employee number and 12 sales attributes, one for each month. In that query we had to list all 12 sales attributes, when it would be much easier to list the one attribute we were not interested in, *eno*. Our query then becomes

```
SUM(SELECT ALL BUT eno
    FROM   Sales)
```

3.4.3 Don't Care Values

When comparing constant values with attributes values in a "don't care" value is useful for making wild card comparisons. Our "don't care" value is the question mark (?). To illustrate its use, consider the query to get *Company* tuples where one of the employees has name "Smith" and salary $20,000. One way to write this query is to look for an employee tuple with "ename = 'Smith', "sal = 20000", and any value for *eno*. We can use our "don't care" value as follows

```
Company WHERE <?, "Smith", 20000> IN Emps
```

This is certainly more straightforward than the alternative

```
Company WHERE EXISTS
        (Emps WHERE ename = "Smith"
              AND sal = 20000)
```

Various other text matching facilities could be incorporated. RDL has a proposed text matching facility based on the SQL "LIKE" predicate, and much of Schek's work (cf. [Sch1]) has been involved with text retrieval in a database system.

3.5 Data and Relation Restructuring Operations

Two operations, NEST and UNNEST are provided for restructuring relations into either more or less nested forms. One operation, ORDER, rearranges the tuples of a relation.

The restructuring operations correspond to the nest and unnest operators of the ¬1NF relational algebra. The syntax of these operators is

```
NEST (query)                                    UNNEST (query)
ON   attribute-list [AS name]                   ON     attribute-list
```

In the following, let *Rel* be the relation formed by (*query*). The NEST operation partitions *Rel* on the attributes not specified in the *attribute-list*. For each partition, a new tuple is created with the values of the attributes in the *attribute-list* collected into a new nested relation. The nested relation is given an optional *name* but, if not named, it cannot be referenced any place else in the query. Note that if any nested relation formed consists solely of tuples in which every attribute has a value which is null or the empty relation, then the value of this nested relation is the empty relation.

Let us go through a step by step building of a query to convert the 1NF database of Figure 3-1 to the ¬1NF database of Figure 3-2.

First we will nest the employee relation by collecting *eno*, *ename*, and *sal* into a nested relation called *Emps*.

```
NEST  Emp
ON    eno, ename, sal AS Emps
```

Next, we join this relation with the *Dept* relation on *dno* and eliminate one of the duplicate *dno* columns.

```
SELECT ALL BUT Emp.dno
FROM   Dept, (NEST Emp
                ON   eno, ename, sal AS Emps)
WHERE  Dept.dno = Emp.dno
```

This query produces all tuples in the *Company* relation where departments have employees. If we want to also include the departments which do not have employees, assigning a null tuple to the nested *Emps* relation, we need to *preserve* the *Dept* relation. The final query is

```
SELECT   ALL BUT Emp.dno
FROM     Dept, (NEST  Emp
                  ON    eno, ename, sal AS Emps)
WHERE    Dept.dno = Emp.dno
PRESERVE Dept
```

The UNNEST operation creates several tuples for each tuple in *Rel*, by concatenating the attributes not specified in the *attribute-list* with a tuple from each of the attributes that is specified. The attributes of the unnested relations now become attributes of *Rel*. Note that an empty relation unnests to a single tuple with null values for each atomic attribute and an empty relation for each nested relation.

To unnest the *Company* relation we write

```
UNNEST (Company)
ON     Emps
```

To convert the ¬1NF database to the 1NF database we issue a query for each relation. To get the *Emp* relation we write

```
SELECT eno, ename, dno, sal
FROM   (UNNEST (Company)
         ON     Emps)
WHERE  eno IS NOT NULL
```

and, to get the *Dept* relation we write

```
SELECT dno, dname, loc
FROM   Company
```

In SQL, the ORDER BY clause is added to a query if tuples are to be sorted in some particular order before being output to the user. We retain this function, but modify its syntax to match the other functions in our language. The new syntax is

```
ORDER (query)
   BY name [ASC | DESC] {, name [ASC | DESC]}
```

To sort the *Company* relation into ascending order by location, we write

```
ORDER Company
   BY loc ASC
```

3.6 Name Inheritance and Aliasing

The attributes of the relations formed in the FROM clause of a SFW-expression may be used in several places in a query. They may be referenced directly in (1) the SELECT clause, (2) the FROM clause of a nested SFW-expression, or (3) the WHERE clause. Attributes may be referenced also in the WHERE clause of any nested SFW-expression. A problem occurs when attribute names are not unique. This can be due to the need to use multiple copies of a relation in a single query or to the presence of identical names in different relations, or nested relations.

In the first case, when we need to use multiple copies of a relation, the solution is to introduce *reference names* for the relations. For example, the query to get all pairs of department names that exist at the same location requires reference name for the *Company* relation. A reference name is specified by including the key word AS and the new name.

```
SELECT First.dname, Second.dname
FROM   Company AS First, Company AS Second
WHERE  First.loc = Second.loc AND First.dno < Second.dno
```

If necessary, or desired, reference names can also be used for nested SFW-expressions and for attribute names. If a single relation X is specified in the FROM clause of a SFW-expression the name of the resulting relation defaults to X. However, if there is more than one relation in the FROM clause, then there is no default, and a reference name is required if the resulting relation is going to be referenced elsewhere in the query. Consider the last query to get pairs of department names. Let us use the result of this query to get all triples of department names at the same location, and rename the attributes to *dname1*, *dname2*, and *dname3*. We will use the reference name *Pairs* for the last query and use *Pairs2* for a new reference name for *Pairs*.

```
SELECT Pairs.First.dname AS dname1, Pairs.Second.dname AS dname2, Pairs2.Second.dname AS dname3
FROM   (SELECT First.dname, Second.dname
        FROM   Company AS First, Company AS Second
        WHERE  First.loc = Second.loc AND First.dno < Second.dno) AS Pairs,
       Pairs AS Pairs2
WHERE  dname1 = Pairs2.First.dname
```

Note how reference names are cascaded when necessary to distinguish attribute names. This same cascading may be necessary when dealing with nested relations. In the Corp relation introduced in the

first section, the attribute *name* occurs more than once. If it is necessary to distinguish them, each *name* attribute can be prefixed by the relation name of the nested relation in which it occurs. Unnesting a relation via the UNNEST operator may require that the unnested relation's name be attached to any names which would otherwise be identical in the resulting relation. For example, if we unnested the *Corp* relation via the query:

```
UNNEST Corp ON Emp, Children
```

then the resulting relation would have attributes *name*, from the unnest of *Emp*, and *Children.name*, from the unnest of *Children*.

Reference names are useful for simplifying queries in which a nested query expression is used in several places in the query. Recall from section 3.1 the query to get department names and employees making more than $35,000, eliminating departments with no employees meeting the salary requirement. Our solution then was

```
SELECT dname, (Emps WHERE sal > 35000)
FROM   Company
WHERE  EXISTS(Emps WHERE sal > 35000)
```

By using a reference name for the nested query on *Emps*, the duplication can be eliminated, as follows:

```
SELECT dname, (Emps WHERE sal > 35000) AS Emps-rich
FROM   Company
WHERE  EXISTS(Emps-rich)
```

4. Data Manipulation Language

In this section we discuss commands to store, modify, and erase data from relations in the database. These commands can be thought of as functions which transform relations into other relations by adding, changing, or deleting data from them. Just as an SFW-expression produces relations from relations, the DML commands perform similarly with the additional effect that the new relations replace the old relations in the database. Thinking of DML commands as functions is critical if we want to apply them to ¬1NF relations. In a ¬1NF model we will need to manipulate nested relations as easily as we manipulate traditional relations.

To get the feel for our syntax (adapted from the RDL standard [X3H2]), let us start with some examples on the 1NF database of Figure 3-1. The STORE statement can be used to add user specified tuples to a relation or to add tuples retrieved via a query specification to a relation. To add two new departments to the *Dept* relation, we write

```
STORE Dept
  VALUES <50, Training, Waco>
         <60, Sales, Austin>
```

Suppose we had a relation *New-Dept* with two attributes, *deptno* and *deptname*, which contained information on some new departments. If we want to store this data in the *Dept* relation we write

```
STORE Dept(dno, dname)
  SELECT ALL
  FROM   New-Dept
```

For each of the *New-Dept* tuples stored in *Dept*, the *loc* attribute will be set to the default value defined for *loc* in the schema definition, or NULL if no default value was specified. In general, an arbitrary SFW-expression

can be used in the STORE command to specify the tuples to be stored. Of course, the relation created must be compatible (number of attributes and domain types) with the relation being stored into.

The MODIFY command is used to replace values with others in the database. Suppose we want to give every employee in the *Emp* relation a 10% raise. We would write

```
MODIFY Emp
  SET sal = sal * 1.1
```

If we want to limit the raise to those employees in department 10, we add a WHERE clause to the query as follows:

```
MODIFY Emp
  SET sal = sal * 1.1
  WHERE dno = 10
```

In general, we can specify more than one replacement in the SET clause, and qualify the tuples to be modified via an arbitrary predicate in the WHERE clause.

The ERASE command is used to delete tuples from relations. An optional WHERE clause is used to identify the tuples to be deleted. Let us delete all departments with no employees.

```
ERASE Dept
WHERE NOT EXISTS (Emp WHERE Dept.dno = Emp.dno)
```

All three DML commands operate by first computing all changes, and then making all changes to the relation in one atomic action. This way the relation being changed may be referenced in a nested SFW-expression without fear of it changing while the command is being executed. For example, suppose we want to delete all employees whose salary is greater than the current average salary. The appropriate ERASE command is

```
ERASE Emp
WHERE sal > AVG(SELECT sal FROM Emp)
```

If, instead of the above rule, we recalculated the average salary as we checked and perhaps deleted each tuple in *Emp*, it is possible to wind up deleting all tuples in *Emp*!

Now let us focus on the particular problem that ¬1NF relations pose for our DML commands. We need a way of performing the three DML commands on individual nested relations. No matter which operation we perform on a nested relation, we are changing only the relation in which the updated relation is nested. Therefore, all changes to nested relations are done with a MODIFY command on the database relation. For the next set of examples we use the ¬1NF database of Figure 3-2. Suppose we want to insert a new employee, <32, Samuels, 49000>, working in department 10. An outline of the required command is

```
MODIFY Company
  SET Emps = X
  WHERE dno = 10
```

Since *Emps* is a nested relation, what should we use for X in this operation? Since we allow any atomic-valued expression to be used when the attribute being changed is atomic, we allow any relation valued expression to be used when the attribute being changed is a relation. Thus, one legitimate solution is to replace X with

```
(Emps UNION <32, Samuels, 49000>
```

Another, more general solution is to replace X with a "nested" STORE command on the *Emps* relation. The total query is then

```
MODIFY Company
  SET Emps = (STORE Emps
                VALUES <32, Samuels, 49000>)
  WHERE dno = 10
```

In general, we can use UNION instead of STORE and DIFFERENCE instead of ERASE, however, there is usually not a good way to simulate MODIFY using a query expression. The command to give each employee in department 10 that makes more than $30,000, a 10% raise, is written

```
MODIFY Company
  SET Emps = (MODIFY Emps
                SET sal = sal * 1.1
                WHERE sal > 30000)
  WHERE dno = 10
```

The alternative query expression for the nested MODIFY is the much more complex expression:

```
SELECT eno, ename, sal * 1.1
FROM   Emps
WHERE  sal > 30000
UNION
Emps WHERE sal <= 30000
```

In summary, when tuples are to be stored, modified, or erased from a nested relation, either a query expression can be constructed to perform the modification or the appropriate DML command can be used. In either case, any operation on a nested relation is done within a MODIFY command on the relation containing the nested relation.

5. The SQL/NF Data-Definition Language

In standard SQL, it is possible to define relations using the CREATE TABLE command, and views using the DEFINE VIEW command. As part of the CREATE TABLE command, the user specifies the attribute names and the domain (e.g., integer, character) to be associated with each attribute of the relation. In the proposed RDL standard, base tables and views are defined in a SCHEMA command, which includes a TABLE command for each base table being defined, and a VIEW command for each view being defined. In addition to specifying the attribute names and their domains, a variety of integrity constraints can also be specified (UNIQUE, NOT NULL, REFERENCES ..., CHECK ...)[1].

We shall adopt the RDL framework for the SQL/NF data-definition language. However, we shall make need to make appropriate modifications to allow for definition of ¬1NF relations. Let us first show the definitions for the 1NF database in Figure 3-1.

```
SCHEMA
  TABLE Dept
    ITEM dno INTEGER UNIQUE NOT NULL
    ITEM dname CHARACTER 10
    ITEM loc CHARACTER 10
  TABLE Emp
    ITEM eno INTEGER UNIQUE NOT NULL
    ITEM ename CHARACTER 10
    ITEM dno INTEGER REFERENCES Dept.dno
    ITEM sal REAL
```

[1] See [X3H2] for details on these constraints.

Each ITEM command defines a column in the relation. The UNIQUE constraint specifies that no duplicates are allowed for the attribute (thus forming a *key* for the relation). The NOT NULL constraint specifies that no null values are allowed for the attribute, and the REFERENCES constraint disallows any value for the attribute that is not a value in the referenced column. Note that these constraints are also allowed as separate clauses in a TABLE definition. This is especially needed when two or more attributes are to be key for a relation and their combination must be specified as UNIQUE (see the Appendix for syntax).

Following the principle of orthogonality, in order to define ¬1NF relations, we must allow TABLE definitions wherever an atomic-valued specification could occur before. The definitions for the ¬1NF database of Figure 3-2 are:

```
SCHEMA
  TABLE Company
    ITEM dno INTEGER UNIQUE NOT NULL
    ITEM dname CHARACTER 10
    ITEM loc CHARACTER 10
    ITEM (TABLE Emps
            ITEM eno INTEGER UNIQUE
            ITEM ename CHARACTER 10
            ITEM sal REAL)
```

In order to simplify the definition of nested schemes, we allow for the definition of *relation schemes* separately from the definition of the relations themselves. This option is analogous to the option in Pascal of defining the type of a variable directly, or by using a user-defined type. Therefore, we introduce the SCHEME command, which can be used to specify table definitions without actually creating a table. This command is especially useful when deeply nested relations are being defined or when the same nested relation scheme is to appear in more than one place.

The formal definitions for our sample corporation database follow.

```
SCHEME
  TABLE PARTSET
    ITEM part INTEGER UNIQUE NOT NULL
  TABLE PERSON
    ITEM name CHARACTER 10 UNIQUE
    ITEM dob CHARACTER 8
  TABLE EMPLOYEE
    ITEM empno INTEGER UNIQUE
    ITEM name CHARACTER 10
    ITEM sal REAL
    ITEM mgr INTEGER REFERENCES EMPLOYEE.empno
    ITEM (TABLE Children PERSON)
SCHEMA
  TABLE Corp
    ITEM dno INTEGER UNIQUE
    ITEM dname CHARACTER 10
    ITEM loc CHARACTER 10
    ITEM (TABLE Emp EMPLOYEE)
    ITEM (TABLE Usage PARTSET)
  TABLE Supply
    ITEM supplier INTEGER UNIQUE
    ITEM (TABLE Supplies PARTSET)
```

One noticeable absence from our language is the SQL CREATE INDEX command. Indices are in the realm

of physical database access concerns and should not be a user specified option. Unfortunately, in SQL, this command is also the means used to specify the UNIQUE constraint on attributes. In SQL/NF, this constraint has been moved to its rightful place in the schema definitions and so the CREATE INDEX command is no longer necessary at the user level.

6. Comparison with Other Languages

In this section, we look at other database languages which have been developed to deal with databases that are not based on the standard 1NF model. We only briefly mention non-SQL-like languages and provide a more detailed comparison of the SQL-like languages.

Non-SQL-like languages include those developed for functional data models [Zan, Shi] and those developed from a "Query-by-Example" model [JW, Hsi]. The GEM language [Zan] is a derivative of QUEL which works on a semantic data model of the Entity-Relationship type. The DAPLEX language [Shi] uses an English-like syntax which works on a functional data model. Both GEM and DAPLEX use a functional composition notation to relieve users of explicitly specifying joins. This composition is explicitly represented in the ¬1NF data model with the use of nested relations. GEM allows single attributes to be set-valued one level deep. For example, and attribute *color* may have value {green} or {yellow, red}. This corresponds to limiting rules in our model to the form $R = (A_1, A_2, \ldots, A_n)$ where each A_i is either zero order or a higher order attribute with associated rule $A_i = (B)$ where B is zero order. Neither GEM nor DAPLEX supports explicit nesting or unnesting of set-valued attributes, however, each retains a version of the SQL GROUP BY operation for executing aggregate functions.

The language, Unified Query-By-Example (UQBE) [Hsi], is based on Jacobs' database logic [Jac1] and the functional data model. UQBE queries are translated into either QBE or a Functional Query Language which can be translated into other languages like QUEL and SEQUEL. Jacobs' own QBE-like language, Generalized Query-By-Example (GQBE) [JW], is based strictly on database logic. These languages operate on ¬1NF relations, however, the two-dimensional format is quite different from a SQL-like language, so direct comparison is not made. In fact, Jacobs has defined a Generalized SQL (GSQL) language [Jac2] with power similar to the QBE-like languages. We will look at GSQL later in this section.

One SQL-like language which also uses a form of functional composition is SQL/N [Bra]. SQL/N is upward compatible with SQL and provides "natural language" quantifiers, like "FOR ALL" and "THERE IS 1", for joining relations over common attributes. "PARENT" and "CHILD" relationships between tuples are based on the foreign key concept. As we mentioned above, the ¬1NF model allows explicit representation of these relationships with the use of nested relations.

In the rest of this section, we provide more detailed comparisons of SQL/NF with two languages designed for ¬1NF databases, GSQL and the database language being developed at IBM Heidelberg for "NF^2" relations [PHH, SP]. Figure 6-1 shows some example queries, written in SQL/NF, GSQL, and the NF^2 query language.

6.1 Generalized SQL

The GSQL language is a generalization to database logic of relational SQL. Nested relations are called *clusters*. GSQL does not support nested SFW-expressions in the FROM or WHERE clauses. All WHERE clause predicates, whether they apply to a nested relation or not, are included in the single WHERE clause of each query. If a predicate references an atomic attribute of the database relation then entire tuples are selected or

SQL/NF

```
1. SELECT dname,
          (SELECT  eno, ename
           FROM    Emps
           WHERE   sal > 35000)
   FROM    Company

2. SELECT dno, COUNT(Emps)
   FROM    Company

3. NEST    Company
   ON      ALL BUT loc AS Depts

4. UNNEST Company
   ON      Emps

5. SELECT dname, ename
   FROM    (UNNEST Company
            ON Emps)
```

GSQL

```
1. SELECT (dname, E) AS DE
          (eno, ename) AS E
   FROM   Company
   WHERE  sal > 35000

2. No translation known

3. SELECT (loc, Depts)
     (dno, dname, Emps) AS Depts
   FROM   Company

4. SELECT dno, dname, loc,
          eno, ename, sal
   FROM   Company

5. SELECT dname, ename
   FROM   Company
```

NF^2

```
1. SELECT ⟨ X.dname,
          (SELECT ⟨ XX.eno, XX.ename ⟩
           FROM   XX IN X.Emps
           WHERE  XX.sal > 35000) ⟩
   FROM   X IN Company

2. SELECT ⟨ X.dno, CARD(X.Emps) ⟩
   FROM   X IN Company

3. NEST   X IN Company
   ALONG  ⟨ X.dno, X.dname, X.Emps ⟩

4. UNNEST X IN Company
   ALONG  X.Emps

5. SELECT ⟨ Y.dname, Y.ename ⟩
   FROM   Y IN (UNNEST X IN Company
                ALONG  X.Emps)
```

Queries:

1. Get department names and employee numbers and names for all employees making more than $35,000.
2. Get department numbers and the number of employees in each department.
3. Create a new nested relation called *Depts* which contains all departments in each location.
4. Get the 1NF version of *Company*.
5. Get pairs of department names and employee names where the employee works in the department.

Figure 6-1. Five sample queries written in SQL/NF, GSQL, and NF^2 Query Language.
Note: NF^2 syntax estimated from [PHH] and GSQL syntax estimated from [Jac2].

rejected, however, if the attribute is in a nested relation then tuples from that nested relation are selected or rejected. In the SELECT clause, attributes may be included from anywhere in the relation. If some attributes are from nested relations, they are unnested appropriately. The attributes selected can be renested in an arbitrary way by specifying clusters in the SELECT clause (see query 1 in Figure 6-1.) Functions may be specified as in standard SQL, however, no support for GROUP BY is mentioned in [Jac2].

The major disadvantage of GSQL is its extreme lack of orthogonality, as witnessed by the lack of nested SFW-expressions in the SELECT and FROM clauses, and the hidden unnesting that goes on when attributes are selected. Of course, the same problems with functions in SQL, are present in GSQL, since there is no change from SQL in this area.

6.2 Query Language for NF^2 Relations

The NF^2 query language has syntax and properties that are similar to SQL/NF. In [PHH], some of the query facilities are described and were used to generate the example queries in Figure 6-1. The language includes many more built in functions than standard SQL, including several functions to work with a "list" data structure. They also retain the "GROUP BY" function and also include an inverse operation "DUNION." There is a new syntax for "GROUP BY" which aligns it with the syntax of the "NEST" and "UNNEST" functions:

GROUP *reference-name* IN *relation-name*
BY *reference-name.attribute-name*

There is, however, no indication of how this new grouped relation is used in a query, particularly in applying aggregate operators to the groups.

7. Conclusion

SQL is a powerful query language, responsible for a lot of the current acceptance of relational databases. SQL-like languages are easy to learn and provide improved data independence over former database query languages. We have extended SQL to enhance its ease of use and expanded its expressiveness to deal with ¬1NF relations. The ¬1NF model allows us to depict relationships more clearly and with less redundancy than a 1NF model. The ability to restructure relations by the nest and unnest operations gives the user unlimited control over the format in which he sees his data. This structuring leads also to the elimination of the complex "GROUP BY" operation.

Our model keeps the relational model "pure" in that all data is represented as relations, or, recursively, as relations within relations. There is no longer two types of structures—relations and partitioned relations (created by "GROUP BY".) This consistency makes application of functions and use of SFW-expressions straightforward and logical.

In summary, the major advantages of our language are

- Orthogonality of expressions. Wherever a relation could logically occur, a SFW-expression is allowed.
- Orthogonality of functions. Functions are applied to relations, and not to attributes which stood for relations.
- Arbitrary restructuring of relations via the nest and unnest operators.
- Elimination of "GROUP BY" and "HAVING" clauses.
- Use of references names to simplify queries and to rename attributes.
- More complete and logical treatment of null values, including a method for performing outer joins and elimination of subsumed tuples.
- Upward compatibility from a strict 1NF system, in which SFW-expressions in the SELECT clause must evaluate to single values, and relation-values are not allowed in the database.

8. Bibliography

[AB] Abiteboul, S. and N. Bidoit, "Non First Normal Form Relations: An Algebra Allowing Data Restructuring," Rapports de Recherche No 347, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France (November 1984).

[BRS] Bancilhon, F., P. Richard and M. Scholl, "On Line Processing of Compacted Relations," *Proceedings of the Eighth International Conference on Very Large Databases*, Mexico City (September 1982), 263–269.

[Bra] Bradley, J., "Application of SQL/N to the Attribute-Relation Associations Implicit in Functional Dependencies," *International Journal of Computer and Information Sciences 12*, 2 (1983), 65–86.

[C+] Chamberlin, D., et al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM Journal of Research and Development 20*, 6 (November 1976), 560–575.

[Dat1] Date, C., "A Critique of the SQL Database Language," *ACM SIGMOD Record 14*, 3 (November 1984), 8–54.

[Dat2] Date, C., "Some Principles of Good Language Design with especial reference to the design of database languages," *ACM SIGMOD Record 14*, 3 (November 1984), 1–7.

[Dat3] Date, C., "The Outer Join," *ICOD-2 Proceedings Second International Conference on Databases*, Cambridge (September 1983), 76–106.

[FV1] Fischer, P. and D. Van Gucht, "Determining when a Structure is a Nested Relation," *Proceedings of the Eleventh International Conference on Very Large Databases*, Stockholm (August 1985).

[FV2] Fischer, P. and D. Van Gucht, "Weak Multivalued Dependencies," *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo (April 1984), 266–274.

[GP] Gründig, L. and P. Pistor, "Land-Informations-Systeme und Ihre Anforderungen an Databank-Schnittstellen." In *Sprachen für Datenbanken*, J. Schmidt, Ed., Informatik Fachberichte Nr. 72, Springer-Verlag, Berlin, 1983.

[Hsi] Hsieh, Y., "An Unified Query-By-Example Information Manipulation Language Based on Functional Data Model," *Proceedings of the 7th International Computer Software Applications Conference*, Chicago (November 1983), 571–579.

[Jac1] Jacobs, B., *Applied Database Logic I: Fundamental Database Issues*, Prentice-Hall, Englewood Cliffs, 1984.

[Jac2] Jacobs, B., *Applied Database Logic II: Heterogeneous Distributed Query Processing*, Prentice-Hall, Englewood Cliffs, 1985.

[JW] Jacobs, B. and C. Walczak, "A Generalized Query-by-Example Data Manipulation Language Based on Database Logic," *IEEE Transactions on Software Engineering* 9, 1 (January 1983), 40–56.

[Jae1] Jaeschke, G., "Nonrecursive Algebra for Relations with Relation Valued Attributes," TR 84.12.001, Heidelberg Scientific Center, IBM Germany (December 1984).

[Jae2] Jaeschke, G., "Recursive Algebra for Relations with Relation Valued Attributes," TR 84.01.003, Heidelberg Scientific Center, IBM Germany (January 1984).

[JS] Jaeschke, G. and H. Schek, "Remarks on the Algebra of Non First Normal Form Relations," *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Los Angeles (March 1982), 124–138.

[KTT] Kambayashi, Y., K. Tanaka and K. Takeda, "Synthesis of Unnormalized Relations Incorporating More Meaning," *Information Sciences 29* (1983), 201–247.

[Kun] Kunii, H., "Graph Data Language: A High Level Access-Path Oriented Language," PhD Dissertation, TR-216, Department of Computer Science, University of Texas at Austin (May 1983).

[Mac] Macleod, I., "SEQUEL as a Language for Document Retrieval," *Journal of the American Society for Information Science* (September 1979), 243–249.

[Mak] Makinouchi, A., "A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model," *Proceedings of the Third International Conference on Very Large Databases*, Tokyo (October 1977), 447–453.

[OO] Özsoyoğlu, G. and Z. Özsoyoğlu, "An Extension of Relational Algebra for Summary Tables," *Proceedings of the 2nd International (LBL) Conference on Statistical Database Management*, Los Angeles (September 1983), 202–211.

[OY] Özsoyoğlu, Z. and L. Yuan, "A Normal Form for Nested Relations," *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Portland (March 1985), 251–260.

[PHH] Pistor, P., B. Hansen and M. Hansen, "Eine sequelarige Sprachschnittstelle für das NF2-Modell." In *Sprachen für Datenbanken*, J. Schmidt, Ed., Informatik Fachberichte Nr. 72, Springer-Verlag, Berlin, 1983.

[RKS1] Roth, M., H. Korth and A. Silberschatz, "Null Values in ¬1NF Relational Databases," Technical Report (to appear), Department of Computer Science, University of Texas at Austin.

[RKS2] Roth, M., H. Korth and A. Silberschatz, "Theory of Non-First- Normal-Form Relational Databases," TR-84-36, Department of Computer Science, University of Texas at Austin (December 1984).

[Sch1] Schek, H.-J., "Methods for the administration of textual data in database systems." In *Information Retrieval Research*, Oddy, Robinson, Van Rijsbergen and Williams, Eds., Buttersworth, London, 1981, 218–235.

[Sch2] Schek, H.-J., "Towards a Basic Relational NF^2 Algebra Processor," *International Conference on Foundations of Data Organization*, Kyoto, Japan (May 1985), 173–182.

[SP] Schek, H.-J. and P. Pistor, "Data Structures for an Integrated Data Base Management and Information Retrieval System," *Proceedings of the Eighth International Conference on Very Large Data Bases*, Mexico City (September 1982), 197–207.

[ScS1] Schek, H.-J. and M. Scholl, "An Algebra for the Relational Model with Relation-Valued Attributes," TR DVSI-1984-T1, Technical University of Darmstadt, Darmstadt, West Germany (1984).

[ScS2] Schek, H.-J. and M. Scholl, "Die NF^2-Relationenalgebra zur Einheitlichen Manipulation Externer, Konzeptueller und Interner Datenstrukturen." In *Sprachen für Datenbanken*, J. Schmidt, Ed., Informatik Fachberichte Nr. 72, Springer-Verlag, Berlin, 1983.

[Shi] Shipman, D., "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems 6*, 1 (March 1981), 140–173.

[TF] Thomas, S. and P. Fischer, "Nested Relational Structures." In *The Theory of Databases*, P. Kanellakis, Ed., JAI Press, to appear, 1985.

[X3H2] X3H2-84-2, "(Draft Proposed) Relational Database Language," American National Standards Committee on Computer and Information Processing, January, 1984.

[Zan] Zaniolo, C., "The Database Language GEM," *Proceedings of the ACM-SIGMOD Conference on Management of Data*, San Jose (May 1983), 207–218.

Appendix
SQL/NF BNF

The following is a modified BNF definition of the queries facilities, DML, and DDL in SQL/NF. We used RDL [X3H2] as a baseline definition. Non-distinguished symbols are enclosed with "⟨⟩". The structure [ ] indicates an optional entry, and the structure "..." indicates an additional zero or more repetitions of the previous entry. Braces are used for grouping in the BNF. Except where modified by braces, sequencing has precedence over disjunction (indicated by "|").

Query Facilities

⟨query expression⟩ ::- ⟨query spec⟩ | ⟨structured query⟩ | function (⟨query expression⟩)
| ⟨nested query expression⟩ | ⟨query expression⟩ ⟨set operator⟩ ⟨query expression⟩

⟨structured query⟩ ::- NEST ⟨nested query expression⟩ ON ⟨column list⟩ [AS ⟨column name⟩]
| UNNEST ⟨nested query expression⟩ ON ⟨column list⟩
| ORDER ⟨nested query expression⟩ BY ⟨sort spec⟩...

⟨sort spec⟩ ::- {⟨unsigned integer⟩ | ⟨column name⟩} [ASC | DESC]

⟨query spec⟩ ::- ⟨select from spec⟩ [WHERE ⟨search condition⟩ [PRESERVE ⟨table list⟩]]

⟨select from spec⟩ ::- SELECT ⟨select list⟩ FROM ⟨table list⟩ | ⟨table name⟩

⟨select list⟩ ::- ALL | ⟨column list⟩ | ⟨select spec⟩ [{, ⟨select spec⟩}...]

⟨select spec⟩ ::- ⟨column expression⟩ | ⟨reference name⟩.ALL

⟨column expression⟩ ::- ⟨value expression⟩ [AS ⟨column name⟩]

⟨table list⟩ ::- ⟨table spec⟩...

⟨table spec⟩ ::- ⟨nested query expression⟩ [AS ⟨column name⟩]

⟨search condition⟩ ::- ⟨boolean term⟩ | ⟨search condition⟩ OR ⟨boolean term⟩

⟨boolean term⟩ ::- ⟨boolean factor⟩ | ⟨boolean term⟩ AND ⟨boolean factor⟩

⟨boolean factor⟩ ::- [NOT] ⟨boolean primary⟩

⟨boolean primary⟩ ::- ⟨predicate⟩ | (⟨search condition⟩)

predicate ::- ⟨comparison predicate⟩ | ⟨between predicate⟩
| ⟨in predicate⟩ | ⟨like predicate⟩
| ⟨exists predicate⟩ | ⟨null predicate⟩

⟨comparison predicate⟩ ::- ⟨value expression⟩ ⟨comp op⟩ ⟨value expression⟩

⟨comp op⟩ ::- = | < | > | <= | >= | <> | [NOT] ELEMENT OF | [NOT] CONTAINS | [NOT] SUBSET OF

⟨between predicate⟩ ::- ⟨value expression⟩ [NOT] BETWEEN ⟨value expression⟩ AND ⟨value expression⟩

⟨in predicate⟩ ::- ⟨value expression tuple list⟩ IN ⟨nested query expression⟩

⟨value expression tuple list⟩ ::- ⟨value expression⟩ | <⟨value expression⟩ [{, ⟨value expression⟩}...] >

⟨like predicate⟩ ::- ⟨not further defined⟩

⟨exists predicate⟩ ::- EXISTS ⟨nested query expression⟩

⟨null predicate⟩ ::- ⟨column spec⟩ IS [NOT] NULL

⟨nested query expression⟩ ::- ⟨table name⟩ | (⟨query expression⟩)

⟨column list⟩ ::- [ALL BUT] ⟨column spec⟩ [{, ⟨column spec⟩}...]

⟨function⟩ ::- MAX | MIN | AVG | SUM | COUNT | DISTINCT | SUBSUME

⟨set operator⟩ ::- UNION | DIFFERENCE | INTERSECT

⟨data type⟩ ::- ⟨character string type⟩ | ⟨numeric type⟩

⟨value expression⟩ ::- ⟨term⟩ | ⟨value expression⟩ {+|−} ⟨term⟩

⟨term⟩ ::- ⟨factor⟩ ::- | ⟨term⟩ {*|/} ⟨factor⟩

⟨factor⟩ ::- [+|−] ⟨primary⟩

⟨primary⟩ ::- ⟨nested query expression⟩ | ⟨value spec⟩ | ⟨column spec⟩ | (⟨value expression⟩)

⟨value list⟩ ::- ⟨value spec⟩...

⟨value spec⟩ ::- ⟨literal⟩ | NULL

⟨literal⟩ ::- ⟨character string literal⟩ | ⟨numeric literal⟩ | ⟨tuple literal⟩ | ⟨don't care literal⟩

⟨tuple literal⟩ ::- <⟨value spec⟩ [{, ⟨value spec⟩}...] >

⟨column spec⟩ ::- [{⟨reference name⟩.}...]⟨column name⟩

DML

⟨dml statement⟩ ::- ⟨store statement⟩ | ⟨modify statement⟩ | ⟨erase statement⟩

⟨store statement⟩ ::- STORE ⟨table name⟩ [(⟨column list⟩)] {VALUES ⟨value list⟩ | ⟨query expression⟩

⟨modify statement⟩ ::- MODIFY ⟨table name⟩ [AS ⟨reference name⟩]
SET ⟨set clause⟩... [WHERE ⟨search condition⟩]

⟨set clause⟩ ::- ⟨column name⟩ = {⟨value expression⟩ | (⟨dml statement⟩)}

⟨erase statement⟩ ::- ERASE ⟨table name⟩ [AS ⟨reference name⟩] [WHERE ⟨search condition⟩]

DDL

⟨ddl statement⟩ ::- ⟨schema⟩ | ⟨scheme⟩

⟨schema⟩ ::- SCHEMA {⟨table definition⟩ | ⟨view definition⟩}...

⟨table definition⟩ ::- TABLE ⟨table name⟩ {⟨table element⟩... | ⟨scheme name⟩}

⟨table element⟩ ::- ⟨column specification⟩ | CONSTRAINTS ⟨table constraint definition⟩...

⟨column specification⟩ ::- ITEM {⟨column definition⟩ | (⟨table definition⟩)}

⟨column definition⟩ ::- ⟨column name⟩ ⟨data type⟩ [⟨column constraint spec⟩...] [⟨default clause⟩]

⟨column constraint spec⟩ ::- ⟨not null clause⟩ | ⟨unique clause⟩ | ⟨references clause⟩ | ⟨check clause⟩

⟨not null clause⟩ ::- NOT NULL

⟨unique clause⟩ ::- UNIQUE

⟨references clause⟩ ::- REFERENCES ⟨column spec⟩ [⟨update rule⟩] [⟨delete rule⟩]

⟨check clause⟩ ::- CHECK ⟨search condition⟩

⟨default clause⟩ ::- DEFAULT ⟨literal⟩

⟨table constraint definition⟩ ::- ⟨unique constraint defnition⟩ | ⟨referential constraint definition⟩
| ⟨check constraint definition⟩

⟨unique constraint defnition⟩ ::- UNIQUE ⟨column list⟩

⟨referential constraint definition⟩ ::- REFERENCES ⟨column list⟩ WITH ⟨column list⟩
[⟨update rule⟩] [⟨delete rule⟩]

⟨update rule⟩ ::- ⟨action⟩ MODIFY

⟨delete rule⟩ ::- ⟨action⟩ ERASE

⟨action⟩ ::- CASCADE | NULLIFY | RESTRICT

⟨check constraint definition⟩ ::- CHECK ⟨search condition⟩ [⟨defer clause⟩]

⟨defer clause⟩ ::- IMMEDIATE | DEFERRED

⟨view definition⟩ ::- VIEW ⟨table name⟩ AS ⟨query expression⟩

⟨scheme⟩ ::- SCHEME ⟨scheme definition⟩...

⟨scheme definition⟩ ::- TABLE ⟨scheme name⟩ ⟨table element⟩...