Data Retrieval and Aggregates in SQL^*/NR

Yiu-Kai Ng
Nael Qaraeen
Computer Science Department
Brigham Young University
Provo, Utah 84602, U.S.A.
Email: ng@cs.byu.edu

Abstract

Standard SQL is incapable of handling recursive database queries and nested relations. A proposed solution to allow recursion in SQL was given in SQL^* [KC93], while a solution to allow nested relations in SQL was given in SQL/NF [RKB87]. However, these two problems with SQL were handled separately, and an extended SQL that handles both recursive queries and nested relations is still lacking. To overcome this shortcoming, we propose an extended SQL, called SQL^*/NR, that not only can handle both recursive queries and nested relations, but also allows aggregate operators. A query Q in SQL^*/NR is processed by first transforming Q into rule expressions in LDL/NR, a logic database language for nested relations, and the transformed rule expressions are evaluated for retrieving the desired result of Q. Transforming Q into rule expressions in LDL/NR is desirable since LDL/NR handles recursion on nested relations with a built-in mechanism for recursive query processing. In this paper, we define SQL^*/NR and include an approach for transforming SQL^*/NR queries into rule expressions in LDL/NR. SQL^*/NR, as defined, enhances the expressive power of standard SQL and SQL/NF and has the expressive advantage over SQL^*.

Keywords: Recursion, nested relation, aggregates, logic programming, SQL

1 Introduction

SQL (Structured Query Language), a database query language that has been standardized, has gained lots of popularity and has been supported by most relational database systems since it was first introduced in the 1970's. SQL is widely accepted because it is a "user-friendly" query language that has a simple, declarative syntax and semantics. However, some database applications have revealed at least two limitations of SQL. First, standard SQL is incapable of handling recursive database queries (queries in which a relation is defined in conjunction with its own definition [KC93]). Second, standard SQL cannot handle nested relations that allow non-atomic (i.e., decomposable) attribute values. These deficiencies in standard SQL limit its expressive power and capability of handling complex data, respectively.

A proposed solution to the first problem of SQL was given in [KC93] who defined SQL^* that extends standard SQL to allow recursive queries. A solution to the second problem of SQL was proposed by [RKB87] who defined SQL/NF that extends standard SQL to handle nested relations. However, these two problems with standard SQL are not handled by a single SQL-type language. An extended SQL, called SQL^*/NR, that can handle both recursive queries and nested relations, is to be defined in this paper. The development of SQL^*/NR was motivated in part by the fact that the nested relational model can be used for storing and retrieving complex data, while an extended SQL that allows recursive queries on nested relations enriches the expressive power of standard SQL and SQL/NF, and has the expressive advantage of the nested relational structure over SQL^* that operates on flat relations. Furthermore, SQL^*/NR allows aggregate operators, such as min, max, sum, etc., that makes the language similar to standard SQL and more powerful. SQL^*/NR, as defined, enhances database query languages in [RKB87, Uni91] and recursive query languages for flat relations [KC93] by providing better versatility and a richer functionality for expressing complex data. Moreover, SQL^*/NR incurs most, if not all, of the standard SQL's advantageous features.

We present the details of SQL^*/NR as follows. In Section 2 the basic set of constructs in SQL^*/NR are described. Extended SQL constructs in SQL^*/NR, including subquery components that specify the basis and recursive definitions of a recursive query, are used for creating an SQL^*/NR query. To evaluate an SQL^*/NR query Q, we first transform Q into rule expressions in LDL/NR, a logic database language for nested relations. Rule expressions in LDL/NR are chosen as an internal representation of Q because they support languages of *declarative* nature and handle *recursion* and *nested relations*. Formal definitions of these rule expressions and justification for processing SQL^*/NR queries by using a logic database language are given in Section 2.3. In Section 3 SQL^*/NR queries with aggregate operators are introduced. In Section 4 we give a concluding remarks. In addition, we include in Appendix A the formal syntax of SQL^*/NR in BNF notations, and in Appendix B an approach for the transformation of SQL^*/NR queries with(out) aggregate operators into rule expressions in LDL/NR.

2 SQL^*/NR Queries and LDL/NR Rule Expressions

In this section we first describe SQL^*/NR subqueries which comprise the basic constructs of an SQL^*/NR query. Hereafter, we present the structure of an SQL^*/NR query, and show how SQL^*/NR subqueries are embedded in an SQL^*/NR query. Finally, rule expressions in LDL/NR, which are used for processing a given SQL^*/NR query, are introduced. LDL/NR is of interest because it captures the constraints of nested relations precisely. Furthermore, since for each complex-object type there is a nested-relation type with the same "information capacity" [Hul86], LDL/NR can handle complex-data type queries.

2.1 SQL^*/NR Subqueries

An SQL^*/NR subquery (or subquery for short) allows the user to retrieve data from a set of relations S. The subquery may comply with specific conditions or constraints that are applied to certain attributes of the relations in S. This is similar to the functionality provided by *SELECT, FROM*, and *WHERE* clauses of standard SQL.

The *Select-From-Where* structure of an SQL^*/NR subquery, which is a major construct of SQL^*/NR and is referred as a **S**elect-**F**rom-**W**here-Expression or SFW-Expression for short [RKB87], is made up of the following clauses:

SELECT attribute-list
FROM relation-list
[WHERE boolean-expression]

This expression can handle queries for nested relations. Since an SQL^*/NR subquery is written as an SFW-Expression, a subquery and an SFW-Expression will be interchangeably used throughout the remaining of this paper. Furthermore, we assume that all (atomic and non-atomic) attribute names appearing at all levels of nesting in a nested relation are unique, and apostrophe (') is an invalid symbol in any subquery since it is reserved for the transformation process of an SQL^*/NR query.

2.1.1 The Select Clause

The **Select** clause, which is the first component of an SFW-Expression, allows the user to specify data items to be included in the final result of a subquery by choosing attributes in different relations referenced in the SFW-Expression. The **Select** clause is of the following format:

SELECT *atomic-attr*$_1$, ..., *atomic-attr*$_n$, *non-atomic-attr*$_1$, ..., *non-atomic-attr*$_m$

The **Select** clause consists of the keyword SELECT, followed by a list of atomic and non-atomic attributes. A non-atomic attribute can be a nested relation that is constructed by an SFW-Expression called an *incremental subquery* [RKB87]. (See Example 4.) The **Select** clause is used for specifying all the desired (atomic and non-atomic) attributes to be included in the result of a subquery. It corresponds to the *extended projection* operation in the extended relational algebra [RKS88].

Assume that *Attr* is an (atomic or non-atomic) attribute in nested relation r. If *Attr* appears at the top level scheme of r, we can either include *Attr* in the **Select** clause, or concatenate r (or its reference name which is also called *tuple variable* in [KS91]) with *Attr* separated by a dot in the **Select** clause. Otherwise, *Attr* must be embedded one level deep in r, and we specify *Attr* in the **Select** clause with its preceding (embedded) relations (or their reference names) separated by dots. In addition, each chosen (atomic or non-atomic) attribute C can be referenced by a distinct, new column name to distinguish C from other chosen attributes. A new column name is specified by including the keyword **AS** and the new name following a chosen attribute name.

Example 1 Let Student be the nested relation in Figure 1. To retrieve all the departments in Student, we enter

SELECT Dept ... or *SELECT Student.Dept* ... .

To retrieve all the children of a student, we enter

SELECT Children ... or *SELECT Student.Children* ... .

To retrieve all students' names and their children names, and reference a student's name and his/her children names by *Sname* and *Child-Name*, respectively, we enter

SELECT S.Name AS Sname, (SELECT Cname ...) AS Child-Name ...

where S is the reference name of Student. □

Student				
Name	SID	Children		Dept
		Cname	Age	
John	5678	David	5	CS
		Chris	10	
Bill	2134	Denise	8	Chemistry
		Jim	16	

Has_taken		
SID	Courses	
	Course#	Crhrs
5678	CS 220	3
	PE 100	1
2134	Math 411	3

Figure 1: Nested Relations *Student* and *Has_taken*

If only certain attributes of a relation r (which is embedded in a nested relation s) along with some attributes in relations $s_1, \ldots, s_n$ are to be included in an embedded relation in the result of a subquery Q, we could specify an *incremental subquery* SQ in Q. (It is assumed that s, $s_1, \ldots$, and s_n are specified in the **FROM** clause of SQ or Q.) An example of an incremental subquery is *(SELECT Cname ...)* in Example 1. Upon evaluating SQ, the result of SQ includes the desired attributes in r, $s_1, \ldots, s_n$. SQ is specified as an argument in the **Select** clause of Q, and the resultant relation of SQ is given a reference name (i.e., column-name) which is specified after SQ and the keyword **AS** in the **Select** clause.

2.1.2 The From Clause

The **From** clause includes a set of relations used for computing the result of an SFW-Expression. The format of the **From** clause is:

FROM $rel_1, \ldots, rel_n$

The **From** clause consists of the keyword FROM followed by a list of constructs $rel_i, 1 \leq i \leq n$, where rel_i denotes either a (an embedded) relation name (which could be followed by an optional *reference name*), a $NEST$ operation, or an $UNNEST$ operation. (Each of the $NEST$ and $UNNEST$ operations yields a relation.) The order in which the relation names and the $NEST$ and $UNNEST$ operations may be arranged in the **From** clause is of no importance. The **From** clause corresponds to the *extended cartesian product* operation in the extended relational algebra [RKS88].

Example 2 To use the relation Student in Figure 1 in an SQL^*/NR subquery Q, we include FROM *Student* in Q. To use Student, and Course# and Crhrs of the relation Courses embedded within the nested relation Has_taken in Figure 1, we include

FROM Student S, (UNNEST Has_taken ON Courses)

in a subquery, where S becomes the reference name of Student. □

(Embedded) Relations that are listed in the **From** clause are referenced during the evaluation of the SFW-Expression. An embedded relation r (such as $Courses$ in Has_taken) must be specified in the **FROM** clause of an incremental subquery (such as *(SELECT Cname from Courses)*) in a subquery Q, and the nested relation (such as Has_taken)

within which r is embedded must be included in the **FROM** clause of Q. Reference names of relations (such as S for Student in Example 2) allow the user to specify multiple copies of the same relation and same attribute name appeared in different relations that are included in the **From** clause. $NEST$ (resp. $UNNEST$), an aggregating (resp. disaggregating) operation, allow the user to restructure a relation making it more aggregated (resp. flatter), before such a relation is used by the **Select** or **Where** clause in the SFW-Expression. The $NEST$ and $UNNEST$ operations have the following formats:

(NEST <nested-relation-name> ON attribute-list AS <column-name>) and
(UNNEST <nested-relation-name> ON <attribute-list>)

where nested-relation-name and attribute-list in NEST and UNNEST reference a nested relation r and the attributes in r on which NEST and UNNEST are applied, respectively. The keyword AS in NEST allows the user to give a reference name (i.e., column-name) to the resultant relation of the NEST operation.

2.1.3 The Where Clause

We use the **Where** clause in an SFW-Expression to specify conditions on attributes that need to be satisfied. The **Where** clause is of the following format:

WHERE Boolean-Expression

The **Where** clause, which is optional in an SQL^*/NR subquery, consists of the keyword WHERE followed by a boolean expression that might include k different components grouped together using a combination of m different logical ANDs and n different logical ORs ($0 \leq m, n \leq k-1$, and $m+n \leq k-1$). The clause references atomic and non-atomic attributes of the (embedded) relations in the **From** clause. These attributes are compared to other attributes or constants using (set) comparison operators. The results of such comparisons are grouped together using logical ANDs or logical ORs (if there are any), and its evaluation yields the final result of the boolean expression of the **Where** clause. The **Where** clause corresponds to the *extended selection* (except the *set membership* operators, **in** and **notin**) in the extended relational algebra that is recursively applied to deal with selections on different levels of nesting in a nested relation, if such selections are required.

Example 3 To construct a subquery Q based on Student in Figure 1 in which one of the two conditions must be satisfied: either the age of a student's child is in between 1 and 5, or in between 12 and 15, inclusively, we include

WHERE (C.Age $\geq$ 1 AND C.Age $\leq$ 5) OR (C.Age $\geq$ 12 AND C.Age $\leq$ 15)

in Q, where C is the reference name of the relation Children embedded within Student. □

The selections also handle the comparisons of atomic and non-atomic attributes of two nested relations involved in the *extended natural join* in the extended relational algebra. In the *extended natural join* of two nested relations s and s', two tuples $t \in s$ and $t' \in s'$ are *joinable* if the extended intersection on the projections over common (atomic and non-atomic) attributes of t and t' is non-empty. This constraint is similar to the constraint of the traditional natural join operation, i.e., two tuples contribute to the join if they agree on common attributes.

2.2 SQL^*/NR Queries

This subsection describes the structure of an SQL^*/NR query (or query for short) which is constructed by using subqueries described in section 2.1.

An SQL^*/NR query Q, which can handle recursion on nested relations, consists of an INSERT INTO statement [KC93] that includes the keyword *INSERT INTO* followed by the relation R (that is computed recursively), one subquery called the *Basis-subquery*, and n $(0 \leq n)$ different *ALSO* statements, each of which contains a subquery called *Recursive-subquery*. The format of Q is as follows:

INSERT INTO R
    Basis-subquery
ALSO
    Recursive-subquery$_1$
...
ALSO
    Recursive-subquery$_n$

The relation R referenced in the INSERT INTO statement of an SQL^*/NR query Q is the relation to be computed recursively using itself and other relations $r_1, r_2, \ldots, r_n$ that are specified in the *Basis-subquery* or *Recursive-subqueries* of Q.

A recursive query in SQL^*/NR consists of at least two subqueries, a *Basis-subquery* and a *Recursive-subquery*, which are SQL^*/NR subqueries. The main difference between the *Basis-subquery* and the *Recursive-subqueries* in a query Q is that any relation specified in Q can be referenced by any of these subqueries, except the computed relation R which can only be referenced in the *Recursive-subqueries* but not in the *Basis-subquery*. Furthermore, all subqueries must yield *compatible* relation schemes[1].

Example 4 Let Connected be the nested relation in Figure 2 which contains the flight information about the group of cities that are connected directly with a particular city by a flight and their respective distances. We specify the following SQL^*/NR query that generates the relation *Reachable* in Figure 2. *Reachable* contains the information about a group of cities that can be reached, either through a direct or an indirect flight, from a particular city.

```
INSERT INTO Reachable
        SELECT Source-City, (SELECT Dest-City
                             FROM Route) AS Cities
        FROM Connected
ALSO
        SELECT R.Source-City, (SELECT Dest-City
                               FROM Route
                               WHERE Dest-City ≠ R.Source-City) AS Cities
        FROM Connected C, Reachable R
        WHERE C.Source-City in R.Cities
```

[1]Two relation schemes $R_1(A_1, \ldots, A_n)$ and $R_2(B_1, \ldots, B_n)$ are *compatible* if $A_i \in R_1$ and $B_i \in R_2$ $(1 \leq i \leq n)$ have the same domain and $A_i = B_i$.

Connected		
Source-City	*Route*	
	Dest-City	*Dist*
Chicago	New York	750
New York	Chicago	750
	Los Angeles	1970
Boston	Chicago	550

Reachable	
Source-City	*Cities*
	Dest-City
Chicago	{ New York, Los Angeles }
New York	{ Chicago, Los Angeles }
Boston	{ Chicago, New York, Los Angeles }

Figure 2: Nested Relations *Connected* and *Reachable*

Note that the SFW-Expression, (SELECT Dest-City FROM Route ...) AS Cities, is an incremental subquery. □

Further note that tuples in the resultant relation R generated by the evaluation of an SQL^*/NR query have unique atomic components, i.e., tuples with the same atomic components are merged, and duplicate tuples are removed with set equality holding on non-atomic attributes. These constraints are applied to each level of nesting in R.

2.3 Rule Expressions in *LDL/NR*

Rule expressions in LDL/NR are chosen as the internal representation of an SQL^*/NR query Q since there exists an efficient implementation of *recursion* in higher-order logic database systems (LDL/NR is a higher-order logic database language) that guarantees termination and preserves completeness of Q [LN95a]. A logic (deductive) database system, which has a built-in reasoning capability, may be used both as an inference system and as a representation language [GN90, GM92]. Moreover, the syntax and semantics of higher-order logic, which form the theoretical foundation of a deductive database language, are simple, well-understood, and formally well-defined. We take the advantages of the built-in recursive query processing mechanism provided by deductive database systems to process our SQL^*/NR queries through the transformation of an SQL^*/NR query Q into rule expressions in LDL/NR which are evaluated to yield the answer of Q.

LDL/NR restricts $HILOG$ [CC90] to nested relations, and is simpler in notation than $HILOG$-R [CK91] which requires type and named attribute to be attached to each argument of a type declaration and named attribute to each data value in a tuple. Furthermore, LDL/NR is more complete than LDL [STZ92] which allows only nested tuples rather than nested sets.

2.3.1 Syntax of Rule Expressions in *LDL/NR*

Rule expressions REs in LDL/NR are based on the notions of type, term, and formula which in turn are defined on an alphabet in LDL/NR. Constants and variables in REs are of *atomic type*. There are two other types, *set type* and *tuple type*, in REs.

Definition 1 A *type* in REs is inductively defined as follows: (i) an *atomic type* is a type, (ii) a *set type* is a type, and for a set-type $s\{r\}$, r is of either atomic type or tuple type, and (iii) a *tuple type* is a type, and for a tuple-type $p(s_1, \ldots, s_n)$, s_i $(1 \leq i \leq n)$ is of either atomic type or set type. □

Tuple type and set type can be used alternatively to form complex data types, as in [CC90, CK91]. The type declaration $dept(Dname,\ projects\{Pname\},\ employees\{ employee(Ename,\ EID)\})$ defines *dept* which is of tuple type with components *Dname*, which is of atomic type, and *projects* and *employees*, which are of set type.

Definition 2 Objects of atomic type are called *atomic terms*. A *term* is inductively defined as follows: (i) a constant is a term, (ii) a variable is a term, (iii) for a set-type $s\{r\}$, an instance $s\{t_1, \ldots, t_m\}$ is a term called *set term*, where t_i ($1 \leq i \leq m$) is of type r, and (iv) for a tuple-type $p(s_1, \ldots, s_n)$, an instance $p(t_1, \ldots, t_n)$ is a term called *tuple term*, where t_i is of type s_i, $1 \leq i \leq n$.□

Example 5 An instance of the tuple-type *dept* is

$dept(cs,\ projects\{db, se, lp\},\ employees\{employee(smith, 123),\ employee(jones,\ 567)\})$. □

Definition 3 A *(well-formed) formula*, which is constructed by terms, is inductively defined as follows: (i) a tuple term or set term is a formula, (ii) if T and S are terms which form the arguments of a (set) comparison operation θ, then $T\theta S$ is a formula, written as $\theta(T, S)$, (iii) if F is a formula and X is a variable, then $\exists X\ F$ and $\forall X\ F$ are formulas, and (iv) if F and G are formulas, so are $\neg F$, $F \vee G$, $F \wedge G$, $F \rightarrow G$, and $F \leftrightarrow G$. □

A tuple term, set term, or (set) comparison operator with arguments is called an *atom*. A *ground formula* (*term*) is a formula (term) without variables. A *closed formula* is a formula with no free occurrence of any variable.

Definition 4 A *rule (expression)* is of the form *head* :– *body*, where *head* is an atom and *body* is a conjunction of atoms. A *unit rule* is a rule with an empty body. A *fact* is a ground unit rule. □

Example 6 The following rule retrieves all employees who work on one other project besides the *db* project:

works_on_pj(Ename) :– employee(Ename, EID), works_on(EID, wprojs$\{Pname, db\}$). □

2.3.2 Semantics of Rule Expressions in LDL/NR

The declarative semantics of rule expressions in LDL/NR is given by the usual semantics of formulas in LDL/NR. Meaning for each symbol in a formula should be assigned in order to discuss the truth or falsity of the formula. The various quantifiers and connectives have a fixed meaning, but the meaning assigned to each term can vary [Llo87]. We first define LDL/NR universe and LDL/NR base.

Definition 5 Given an instance L of LDL/NR, the LDL/NR *universe* U of L, denoted U_L, is the set of all ground atomic terms (constants) of L, and the LDL/NR *base* B of L, denoted B_L, is the set of all ground unit rules of L. □

To define formally the meaning of a fact as a logical consequence of a set of facts and LDL/NR rules, we introduce the concepts of LDL/NR interpretation and LDL/NR model.

Definition 6 Given an instance L of LDL/NR, an interpretation for L is an LDL/NR *interpretation* if the following conditions are satisfied: (a) the domain of the interpretation is the LDL/NR *universe* U_L, (b) constants in L are assigned to "themselves" in U_L, (c) for a set term $s\{t_1, \ldots, t_m\}$ in L, the assignment of s is a mapping from U_L^m to $\{true, false\}$, and (d) for a tuple term $p(s_1, \ldots, s_n)$ in L, the assignment of p is a mapping from U_L^n to $\{true, false\}$. □

Definition 7 Given an LDL/NR interpretation I of an instance L of LDL/NR and a closed formula F of L, I is an LDL/NR *model*, which is a subset of B_L, for F if F is true with respect to I (or I is a model for F). If S is a set of closed formulas of L, then I is an LDL/NR *model* for S if I is a model for every formula of S. □

A set term $s\{X\}$, where X is a variable, denotes a set term of arbitrary cardinality. As in [CC90], it is assumed that the satisfaction of $s\{e_1, \ldots, e_n\}$ by an LDL/NR interpretation I implies the satisfaction of $s\{X\}$ by I, where $X \subseteq \{e_1, \ldots, e_n\}$. Furthermore, the satisfactions of $s\{a_1, \ldots, a_n\}$ and $s\{b_1, \ldots, b_m\}$ by I imply the satisfaction of $s\{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ by I.

2.4 Transforming SQL^*/NR Queries into LDL/NR Rule Expressions

We give a few examples below; each of these examples includes an SQL^*/NR query Q and the rule expressions in LDL/NR transformed from Q. (The description of an approach for transforming an SQL^*/NR query into LDL/NR rule expressions is given in Appendix B.2.)

Example 7 The transformed rule expressions for the SQL^*/NR query in Example 4 are:

reachable($Source\text{-}City_1$, cities{$Dest\text{-}City_1$}) :-
    connected($Source\text{-}City_1$, route{route'($Dest\text{-}City_1$, $Dist_1$)}).

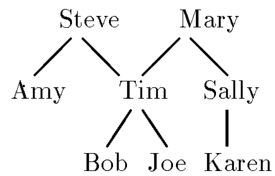
reachable($Source\text{-}City_2$, cities{$Dest\text{-}City_1$}) :-
    connected($Source\text{-}City_1$, route{route'($Dest\text{-}City_1$, $Dist_1$)}),
    reachable($Source\text{-}City_2$, cities{$Dest\text{-}City_2$}),
    in($Source\text{-}City_1$, cities{$Dest\text{-}City_2$}),
    $Dest\text{-}City_1 \neq Source\text{-}City_2$.

Subscripted attributes, such as $Source\text{-}City_1$ and $Source\text{-}City_2$, can be used for distinguishing attributes with the same name from different relations. □

Example 8 Let Parent be the nested relation in Figure 3, and let Q be the following SQL^*/NR query that retrieves all the people who are related to others. The resultant relation Related of Q is shown in Figure 3. It is assumed that Person A is related to person B if either (i) A and B are sibling, (ii) B's parent is related to A, or (iii) A's parent is related to B.

Note that any reference name N assigned to an atomic attribute A in a **Select** clause does not actually change A to N in the rule expressions transformed from Q. It is because N is used in Q only to satisfy the constraint of compatible relation scheme or denote a new reference name of A in Q. This assumption holds for each of the following examples.

Parent	
Pname	*Children*
	Person
Steve	{ Amy, Tim }
Mary	{ Tim, Sally }
Tim	{ Bob, Joe }
Sally	{ Karen }



Related	
Person	*Relatives*
	Relative
Amy	{ Tim, Bob, Joe }
Tim	{ Amy, Sally, Karen }
Sally	{ Tim, Bob, Joe }
Bob	{ Joe, Amy, Sally, Karen }
Joe	{ Bob, Amy, Sally, Karen }
Karen	{ Tim, Bob, Joe }

Figure 3: Nested Relations *Parent* (and Its Graph Form) and *Related*

INSERT INTO Related
  SELECT P_1.Children.Person, (SELECT Person AS Relative
                  FROM P_2.Children
                  WHERE Relative $\neq$ P_1.Children.Person) AS Relatives
  FROM Parent P_1, Parent P_2
  WHERE P_1.Pname = P_2.Pname
ALSO
  SELECT R.Person, (SELECT Person AS Relative
                  FROM Children) AS Relatives
  FROM Parent P, Related R
  WHERE P.Pname in R.Relatives
ALSO
  SELECT Children.Person, (SELECT Relative
                  FROM Relatives) AS Relatives
  FROM Parent P, Related R
  WHERE Pname = R.Person

The transformed rule expressions of Q are:

$$\text{related}(Person_1, \text{relatives}\{Person_2\}) \text{ :- } \text{parent}(Pname_1, \text{children}\{Person_1\}), \text{parent}(Pname_2, \text{children}\{Person_2\}), Pname_1 = Pname_2, Person_2 \neq Person_1.$$

$$\text{related}(Person_2, \text{relatives}\{Person_1\}) \text{ :- } \text{parent}(Pname_1, \text{children}\{Person_1\}), \text{related}(Person_2, \text{relatives}\{Relative_2\}), \mathbf{in}(Pname_1, \text{relatives}\{Relative_2\}).$$

$$\text{related}(Person_1, \text{relatives}\{Relative_2\}) \text{ :- } \text{parent}(Pname_1, \text{children}\{Person_1\}), \text{related}(Person_2, \text{relatives}\{Relative_2\}). Pname_1 = Person_2. \ \square$$

Example 9 Let Parent be the nested relation in Figure 3, and let Married be the flat relation in Figure 4. Let Q be the following SQL^*/NR query that retrieves all groups of people who are of the same generation, and the resultant relation Same-Generation of Q is shown in Figure 4.

It is assumed that a person P and a group of people G are of the same generation if P's parent and a parent of a person in G are of the same generation or are respectively of the same generation as a married couple. It is further assumed that P is of the same generation as himself, and every person who is either a child or a parent is in the flat relation People (which is a relation with a single attribute *person*).

In Q, a set operation $UNION$, which is an infix operator in SQL^*/NR, is used in the following format:

SFW-Expression UNION SFW-Expression.

(The syntax, semantics, and transformation steps of UNION in SQL^*/NR are easily determined.)

```
INSERT INTO Same-Generation
  SELECT (SELECT Person AS Sg-Person      /* create tuples with singular value */
          FROM P-Grp) AS Sg-Grp
  FROM (NEST People on Person AS P-Grp)
ALSO
  SELECT (SELECT Person AS Sg-Person
          FROM P1.Children
          UNION
          SELECT Person AS Sg-Person
          FROM P2.Children) AS Sg-Grp
  FROM Parent P1, Parent P2, Same-Generation SG
  WHERE P1.Pname in SG.Sg-Grp AND P2.Pname in SG.Sg-Grp
ALSO
  SELECT (SELECT Person AS Sg-Person
          FROM P1.Children
          UNION
          SELECT Person AS Sg-Person
          FROM P2.Children) AS Sg-Grp
  FROM Parent P1, Parent P2, Same-Generation SG1, Same-Generation SG2, Married
  WHERE P1.Pname in SG1.Sg-Grp AND P2.Pname in SG2.Sg-Grp AND
        Spouse-A in SG1.Sg-Grp AND Spouse-B in SG2.Sg-Grp
```

The transformed rule expressions of Q, in which *sg* denotes the resultant nested relation Same-Generation, are:

sg(sg-grp$\{Person_1\}$) :- people(p-grp$\{Person_1\}$).

sg(sg-grp$\{Person_1, Person_2\}$) :- parent($Pname_1$, children$\{Person_1\}$),
parent($Pname_2$, children$\{Person_2\}$),
sg(sg-grp$\{Sg\text{-}Person_3\}$),
in($Pname_1$, sg-grp$\{Sg\text{-}Person_3\}$),
in($Pname_2$, sg-grp$\{Sg\text{-}Person_3\}$).

Married	
Spouse-A	*Spouse-B*
Steve	Mary
Mary	Steve

Same-Generation
Sg-Grp
Sg-Person
{ Steve }
{ Mary }
{ Amy, Tim, Sally }
{ Bob, Joe, Karen }

Figure 4: A Flat Relation *Married* and a Nested Relation *Same-Generation*

$$\begin{aligned}
\text{sg}(\text{sg-grp}\{Person_1, Person_2\}) \text{ :- } & \text{parent}(Pname_1, \text{children}\{Person_1\}), \\
& \text{parent}(Pname_2, \text{children}\{Person_2\}), \\
& \text{sg}(\text{sg-grp}\{Sg\text{-}Person_3\}), \\
& \text{sg}(\text{sg-grp}\{Sg\text{-}Person_4\}), \\
& \text{married}(Spouse\text{-}A_5, Spouse\text{-}B_5), \\
& \mathbf{in}(Pname_1, \text{sg-grp}\{Sg\text{-}Person_3\}), \\
& \mathbf{in}(Pname_2, \text{sg-grp}\{Sg\text{-}Person_4\}), \\
& \mathbf{in}(Spouse\text{-}A_5, \text{sg-grp}\{Sg\text{-}Person_3\}), \\
& \mathbf{in}(Spouse\text{-}B_5, \text{sg-grp}\{Sg\text{-}Person_4\}). \ \Box
\end{aligned}$$

3 SQL^*/NR Queries with Aggregates

In this section we discuss the inclusion of aggregate operators, namely avg, min, max, sum, and count, in SQL^*/NR. The inclusion of aggregates in SQL^*/NR offers a wider range of queries in SQL^*/NR that makes the language more appealing to standard SQL users. Furthermore, supporting aggregates in SQL^*/NR also provides a form of data storage reduction by allowing SQL^*/NR users the ability to evaluate aggregates on attributes in a database relation "on the fly" instead of storing the values generated by such operators in the database.

Built-in aggregate operators supported by SQL^*/NR are embedded within a subquery, called *Aggregate-subquery*. Attributes in a nested relation can be chosen as arguments of aggregate operators that apply to different groups of tuples. We define aggregates and a grouping construct[2] in SQL^*/NR queries, and include an approach for transforming an SQL^*/NR query with aggregates into LDL/NR rule expressions.

3.1 Aggregate Subqueries

An *Aggregate-subquery* is an SQL^*/NR subquery as discussed in Section 2.1 with the inclusion of aggregate operators in the **Select** clause and a **group by** construct. An *Aggregate-subquery*, referenced as the Select-From-Where-GroupBy Expression (SFWGB-Expression for short), is of the form:

SELECT attribute-list-with-aggregates

[2]A grouping construct includes a grouping operator, i.e., **group-by**, that collects a number of tuples in a nested relation based on a subset of attributes in these tuples that have the same value.

FROM relation-list
[WHERE boolean-expression]
Group By column-names

The **Select** clause in an SFWGB-Expression has the following syntax:

SELECT *atomic-attr*$_1$, ..., *atomic-attr*$_n$, *non-atomic-attr*$_1$, ..., *non-atomic-attr*$_m$, *agg-str*$_1$, ..., *agg-str*$_p$

where *atomic-attr*$_i$ ($1 \leq i \leq n$) and *non-atomic-attr*$_j$ ($1 \leq j \leq m$) are as defined in Section 2.1.1, and *agg-str*$_k$, $1 \leq k \leq p$, is of the form:

Agg ([DISTINCT] [relation-name.] attribute-name) AS agg-attr

where **Agg** is either **min, max, sum, count** or **avg** which is applied to **attribute-name**. There is an **AS** clause which specifies a reference name (i.e., agg-attr) to the result generated from the aggregation. The **DISTINCT** keyword, which is optional, eliminates duplicates before the aggregate operator is applied.

The difference between an SFW-Expression and an SFWGB-Expression lies in the fact that a **Select** clause in an SFWGB-Expression contains aggregate operators that is applied to one or more attributes on the same level of nesting in a nested relation. For example, to retrieve the average age of the children of each student, where the Student relation in as shown in Figure 1, we enter:

SELECT ... avg(Student.Children.Age) AS Children-Age-Avg,

Furthermore, there exists a *group-by* construct in an SFWGB-Expression but not in any SFW-Expression. In an SFWGB-Expression, the **group-by** construct is of the form:

Group By *atomic-attr*$_1$, ..., *atomic-attr*$_n$

where all *atomic-attr*$_i$, $1 \leq i \leq n$, are atomic attributes at the same level of nesting in a nested relation to which the **Group By** operation is applied. These attributes specified in the **Group By** clause are used to form groups of tuples. Tuples that have the same value on all the attributes *atomic-attr*$_1$, ..., *atomic-attr*$_n$ specified in the **Group By** clause are placed in one group. These groups are then used by the aggregate operators given in the **Select** clause of an SFWGB-Expression.

3.2 SQL^*/NR Queries with Aggregate Subqueries

An SQL^*/NR query with aggregates is an extended version of the SQL^*/NR query discussed in Section 2.2 with the addition of an *Aggregate-subquery* included in a **THEN INTO** statement. The format of an SQL^*/NR query with aggregates is as follows:

INSERT INTO R_1
    $Basis\text{-}subquery$
ALSO
    $Recursive\text{-}subquery_1$
...
ALSO

Parent		
PName	*Children*	
	C-Name	*C-Age*
Daniel	Tom	60
	Merle	55
Merle	Steve	30
	Ann	25
Steve	Joseph	5

Temp		
A-Name	*Descendants*	
	D-Name	*D-Age*
Daniel	Tom	60
	Merle	55
	Steve	30
	Ann	25
	Joseph	5

Dec-Info	
A-Name	*D-Avg-Age*
Daniel	35

Figure 5: Relations *Parent*, *Temp*, and *Dec-Info*

$$\begin{array}{l} \quad\quad Recursive\text{-}subquery_n \\ \text{THEN INTO } R_2 \\ \quad\quad Aggregate\text{-}subquery \end{array}$$

where the **THEN INTO** statement includes the keyword *THEN INTO* and a nested relation R_2 followed by an *Aggregate-subquery* (an SFWGB-Expression). R_1, which is the resultant relation generated by the **INSERT INTO** statement, is treated as a temporary (input) relation to which the *Aggregate-subquery* applies to yield the final desired relation R_2. The following example includes an SQL^*/NR query with aggregates.

Example 10 Let Parent be the nested relation in Figure 5. We specify the following query that generates the relation *Dec-info* in Figure 5. *Dec-info* contains the average age of all the descendants of Daniel.

```
INSERT INTO Temp
        SELECT PName AS A-Name, (SELECT C-Name AS D-Name, C-Age AS D-Age
                                 FROM Children) AS Descendants
        FROM Parent
        WHERE PName = "Daniel"
ALSO
        SELECT T.A-Name, (SELECT C-Name AS D-Name, C-Age AS D-Age
                          FROM Children) AS Descendants
        FROM Parent P, Temp T
        WHERE P.PName in T.Descendants.D-Name
THEN INTO Dec-Info
        SELECT A-Name, AVG(D-Age) AS D-Avg-Age
        FROM Temp
        Group By A-Name □
```

As shown in Figure 5, the temporary relation *temp* is generated as a result of evaluating the recursive query which retrieves all the descendants of "Daniel". Applying the *Aggregate-subquery* in the **THEN INTO** statement to *temp* yields the desired resultant relation *Dec-info*.

3.3 Transforming SQL^*/NR Queries with Aggregates into Rule Expressions in LDL/NR

The transformation of the **INSERT INTO** statement of an SQL^*/NR query Q with aggregates to LDL/NR rule expressions is similar to that used for an SQL*/NR query without aggregates. The description of an approach for transforming the **THEN INTO** statement with an *Aggregate-subquery* in Q into LDL/NR rule expressions is given in Appendix B.1.4. In this subsection we give an example to show the rule expressions in LDL/NR transformed from an SQL^*/NR with aggregates.

Example 11 Let Q be the SQL^*/NR query with aggregates given in Example 10. The transformed rule expressions of Q are:

temp($Pname$, descendants{descendants'(C-$Name$, C-Age)}) :-
    parent($Pname$, children{children'(C-$Name$, C-Age)}),
    $Pname$ = "Daniel".

temp(A-$name_2$, descendants{descendants'(C-$Name_1$, C-Age_1)}) :-
    parent($Pname_1$, children{children'(C-$Name_1$, C-Age_1)}),
    temp(A-$Name_2$, descendants{descendants'(D-$Name_2$, D-Age_2)}),
    in($Pname_1$, descendants{descendants'(D-$Name_2$)}).

dec-info(A-$Name$, D-Avg-Age) :-
        group-by(temp(A-$Name$, descendants{descendants'(D-$Name_1$, D-Age_1)}),
            [A-$Name$],
            [D-Avg-Age = AVG(D-Age_1)]). □

4 Summary and Future Work

We have proposed an extended SQL, called SQL^*/NR, that not only can handle both recursive queries and nested relations, but also allows aggregate operators. An approach, which transforms an SQL^*/NR query Q into rules expressions in LDL/NR which can be evaluated to retrieve the desired answers to Q, has also been presented in Appendix B. The proposed SQL^*/NR constructs, which includes query and subquery, have been implemented [Qar95]. The implementation includes accepting an SQL^*/NR query Q (defined according to the syntax described in Appendix A), transforming Q into rule expressions REs in LDL/NR (according to the transformation approach described in Appendix B), and evaluating REs to yield the desired result of Q.

For future work, we plan to examine the optimization of LDL/NR rule expressions that are generated from an SQL^*/NR query by adopting the "magic sets" approach. This optimization approach is a rewriting method that takes the advantages of the efficiency of the top-down evaluation strategy (which considers only "necessary" results). The simplicity and dependability of the bottom-up evaluation strategy in [LN95b], which uses set-term matching instead of unification, can be adopted for evaluating rewritten rules expressions in LDL/NR generated by the "magic sets" method.

Appendix

A SQL^*/NR BNF

The following is a modified BNF definition of the queries in SQL^*/NR. Non-distinguished symbols are enclosed with “<>”. The structure [...] indicates an optional entry, and the structure {...} indicates an additional zero or more repetitions of the entry.

```
<query expression> ::- INSERT INTO <nested-relation-name> <basis-subquery>
                       [{ALSO <recursive-subquery>}]
                       [THEN INTO <nested-relation-name> <aggregate-subquery>]
<basis-subquery>, <recursive-subquery> ::- SELECT <attribute-list>
                                           FROM <relation-list>
                                           [WHERE <boolean-expression>]
<aggregate-subquery> ::- SELECT <attribute-list-with-aggregates>
                         FROM <relation-list>
                         [WHERE <boolean-expression>]
                         GROUP BY <column-name> {<column-name>}
<attribute-list> ::- {<atomic-attr>}{<non-atomic-attr>}
<atomic-attr> ::- [<relation-name>.] <attribute-name> [AS <column-name>]
<non-atomic-attr> ::- [<rel-name>.] <embedded-relation-name> [AS <column-name>] |
                      (<basis-subquery>) AS <column-name>
<relation-name> ::- <rel-name> | <rel-name>.<rel-name>
<rel-name> ::- <nested-relation-name> | <embedded-relation-name> | <reference-name>
<relation-list> ::- <relation> {<relation>}
<relation> ::- <nested-relation-name> [<reference-name>] |
               [<nested-relation-name> | <reference-name> .]
               <embedded-relation-name> [<reference-name>] |
               (NEST <nested-relation-name> ON attribute-list AS <column-name>) |
               (UNNEST <nested-relation-name> ON <attribute-list>)
<boolean-expression> ::- (<boolean-expression> <bool-optr> <boolean-expression>) |
                         <boolean-expression> <bool-optr> <boolean-expression> |
                         <LR-OP> <Comp-P> <RR-OP> |
                         <LS-OP> <Comp-SetM> <RS-OP>
<bool-optr> ::- AND | OR
<LR-OP>, <RR-OP> ::- <nested-rel-attr> | <constant>
<LS-OP> ::- <nested-rel-attr>
<RS-OP> ::- <nested-relation>
<nested-rel-attr> ::- <attribute-name> | <rel-name>.<nested-rel-attr>
<nested-relation> ::- <rel-name> | <rel-name>.<nested-relation>
<Comp-P> ::- < | > | ≤ | ≥ | = | ≠
<Comp-SetM> ::- in | notin
<attribute-name>,<column-name>, <embedded-relation-name>, <nested-relation-name>,
                    <reference-name> ::- alphabetic-character {alphanumeric-character}
<attribute-list-with-aggregates> ::- <attribute-list>
          <agg> ([DISTINCT] [<relation-name>.] <attribute-name>) AS <attribute-name>
          {<agg> ([DISTINCT] [<relation-name>.] <attribute-name>) AS <attribute-name>}
<agg> ::- MIN | MAX | SUM | COUNT | AVG
```

B A Transformation Approach

This section, which includes an approach for the transformation of an SQL^*/NR query Q into rule expressions in LDL/NR, is divided into two subsections. First, we discuss how to transform each of the *Select*, *From*, *Where*, and *Group By* clauses in a subquery of Q into a subexpression of a rule expression in LDL/NR. (Of course, any subquery of Q without **Where** or **Group By** clause can be handled accordingly.) Then, we describe the transformation of the **INSERT INTO**, **ALSO**, and **THEN INTO** statements in Q which are based on the transformation of the subqueries of Q proposed in the first subsection. A complete transformation algorithm is given in [Qar95].

B.1 Transforming an SQL^*/NR Subquery SQ into a Rule Expression RE

B.1.1 Transforming the From Clause of SQ

The syntax of the **From** clause, as described in Section 2.1.2, consists of the keyword FROM followed by a list of arguments $r_1, \ldots, r_n$, where r_p $(1 \leq p \leq n)$ may reference a relation, an embedded relation, a NEST or an UNNEST operation. For the tuple terms generated from the transformation of relations specified in the **From** clause of a subquery SQ, variables (attributes) having the same name in more than one tuple term (relation) must be distinguishable. To accomplish that, we append the subscript p in r_p $(1 \leq p \leq n)$ to all the atomic attributes in r_p during the transformation of r_p. Each r_p in the **From** clause is transformed accordingly as follows:

1. r_p is the relation rn with attributes $A_1, \ldots, A_m$ and an optional reference name.

   rn is included as a tuple term in the body of the resultant rule expression RE with $A_{1_p}, \ldots, A_{m_p}$ as its arguments. If A_i $(1 \leq i \leq m)$ is an atomic attribute, then the subscript p in r_p is appended to A_i to yield A_{i_p}; otherwise, $A_{i_p} = A_i$. Each non-atomic attribute (i.e., embedded relation) A_i with attributes $B_1, \ldots, B_j$ becomes the set term $A_i\{A'_i(B_1, \ldots, B_j)\}$, and the transformation process of A_i is recursively applied to each level of nesting in A_i. (If j = 1, then A'_i is ignored, and the transformed set term is $A_i\{B_1\}$.) Any reference name specified in r_p is recorded as an alias of rn in the look-up table which is used during the transformation process. The transformation of r_p yields $rn(A_{1_p}, \ldots, A_{m_p})$.

2. r_p is the embedded relation ern with an optional reference name.

   r_p must be specified in the **From** clause of an incremental subquery ISQ in SQ. We first determine the subscript s associated with the relation rel within which ern is embedded as an attribute, and rel must be included as a relation in the **From** clause of SQ. We associate s with ern in the look-up table which allows the **Select** or **Where** clause of ISQ to reference attributes in ern, and append s to all the variables which are transformed from the atomic attributes in ern.

3. r_p is the relation(reference)-name.embedded-relation-name $rn.ern$ with an optional reference name.

   Same as Case 2.

4. r_p is a NEST operation of the format

   (NEST Nested-Relation-Name ON Attribute-List AS Column-Name).

   Assume that the Nested-Relation-Name is rn which has attributes A_1, ..., A_m and the Attribute-List consists of attributes $B_1, \ldots, B_n$, where $\{B_1, \ldots, B_n\} \subseteq \{A_1, \ldots, A_m\}$. Let $\{C_1, \ldots, C_k\} = \{A_1, \ldots, A_m\}$ - $\{B_1, \ldots, B_n\}$. If $C_i, 1 \leq i \leq k$ $(B_j, 1 \leq j \leq n)$ is an atomic attribute, then the subscript p in r_p is appended to C_i (B_j) to yield C_{i_p} (B_{j_p}); otherwise, $C_{i_p} = C_i$ $(B_{j_p} = B_j)$. This process is applied to each level of nesting in attribute C_i (B_j) that is non-atomic. rn is then included in the body of RE with arguments $C_{1_p}, \ldots, C_{k_p}$ and the set term Column-Name cn which contains the tuple term cn' with arguments $B_{1_p}, \ldots, B_{n_p}$ (If n = 1, then cn' is ignored.). The transformation of r_p yields either

   $$rn(C_{1_p}, \ldots, C_{k_p}, cn\{cn'(B_{1_p}, \ldots, B_{n_p})\}) \text{ or } rn(C_{1_p}, \ldots, C_{k_p}, cn\{B_{1_p}\})$$

   which is then included in the body of RE.

5. r_p is an UNNEST operation of the format

   (UNNEST Nested-Relation-Name ON Attribute-List).

   Assume that the Nested-Relation-Name is rn which has attributes A_1, ..., A_m besides the embedded relations $r_1, \ldots, r_k$ specified in the Attribute-List. Further assume that each r_i $(1 \leq i \leq k)$ has attributes $B_1^i, B_2^i, \ldots, B_{i_j}^i$. If $A_q, 1 \leq q \leq m$ $(B_q^s, 1 \leq s \leq k, 1 \leq q \leq s_j)$ is an atomic attribute, then the subscript p in r_p is appended to A_q (B_q^s) to yield A_{q_p} $(B_{q_p}^s)$; otherwise, $A_{q_p} = A_q$ $(B_{q_p}^s = B_q^s)$. This process is applied to each level of nesting in each attribute A_q (B_q^s) that is non-atomic. The transformation of r_p thus yields

   $$rn(A_{1_p}, \ldots, A_{m_p}, \ldots, B_{1_p}^1, \ldots, B_{1_{jp}}^1, \ldots, B_{1_p}^k, \ldots, B_{k_{jp}}^k)$$

   which is then included in the body of RE.

B.1.2 Transforming the Where Clause of SQ

To transform the boolean expression BE in the **Where** clause of SQ, two cases should be considered:

1. BE = $LR\text{-}OP$ Comp-P $RR\text{-}OP$ or $LS\text{-}OP$ Comp-SetM $RS\text{-}OP$

   BE is transformed according to the type of comparison (relational or set membership) operator in BE.

   (a) BE is $LR\text{-}OP$ Comp-P $RR\text{-}OP$

   $LR\text{-}OP$ A ($RR\text{-}OP$ B) must be either an (embedded) attribute name or a constant. If A (B) is a constant, then A (B) remains unchanged; otherwise, all relation names or reference names preceding A (B) are stripped which yields A' (B'), and the subscript i (j) is appended to A' (B') to yield A'_i (B'_j), where i (j) is the subscript associated with the relation within which A (B) is embedded, and this subscript was assigned during the transformation of the **From** clause in SQ. The resultant transformation of BE yields the subexpression L **Comp-P** R, where L is either A or A'_i, and R is either B or B'_j.

(b) BE is *LS-OP* Comp-SetM *RS-OP*

LS-OP A must be a list of one or more (embedded) attributes $Attr_1, \ldots, Attr_n$, and *RS-OP* B is an (embedded) nested relation. To transform A (B), we strip all the relation names or reference names (if there are any) preceding $Attr_1, \ldots, Attr_n$ (B) to yield $Attr'_1, \ldots, Attr'_n$ (B'). Hereafter, the subscript i associated with the nested relation within which $Attr_1, \ldots, Attr_n$ are embedded is appended to $Attr_1, \ldots, Attr_n$ or $Attr'_1, \ldots, Attr'_n$ to yield $Attr_{1_i}, \ldots, Attr_{n_i}$ or $Attr'_{1_i}, \ldots, Attr'_{n_i}$. The subscript j associated with the nested relation B or with the relation within which B' is embedded is appended recursively to every atomic attribute $C_k, 1 \leq k \leq m$, in B (B'). (Subscripts i and j are assigned during the transformation of the **From** clause in SQ.) The resultant transformation of BE yields the subexpression **in**(**notin**)(L, R), where L is either $(Attr_{1_i}, \ldots, Attr_{n_i})$ or $Attr'_{1_i}, \ldots, Attr'_{n_i}$, R is either $B(C_{1_j}, \ldots, C_{m_j})$ or $B'(C_{1_j}, \ldots, C_{m_j})$, and **in**(**notin**) is a built-in predicate in LDL/NR.

2. $BE = (BE_1$ AND/OR $BE_2)$ or BE_1 AND/OR BE_2

BE_1 and BE_2 are recursively processed until they become the basic boolean expressions EXP (i.e., *LR-OP* Comp-P *RR-OP* or *LS-OP* Comp-SetM *RS-OP*), and Case 1 is applied to EXP to yield all the desired subexpressions. During the recursive process, the logical AND and logical OR are replaced by ',' and ';', respectively. Furthermore, the precedence of evaluation specified by the parentheses, i.e., (), (if there is any) is retained in the transformed boolean expression which produces a list of predicates to be included in the body of RE.

B.1.3 Transforming the Select Clause of SQ without Aggregate Operators

As described in Section 2.1.1, the **Select** clause has the following format:

$$\text{SELECT } A_1, \ldots, A_n,\ C_1, \ldots, C_m$$

For each (embedded) attribute $Attr_i$ specified in A_i ($1 \leq i \leq n$) in the **Select** clause, the subscript p, which is given to the relation within which $Attr_i$ is embedded during the transformation of the **From** clause in SQ, is appended to $Attr_i$ to yield $Attr_{i_p}$. Any optional reference name N specified in A_i does not affect the transformation of $Attr_i$ since N is used in SQ only to satisfy the constraint of compatible relation scheme or denote a new reference name of $Attr_i$. $Attr_{i_p}$ is included in the head tuple term of RE as shown below.

$$\text{r}(Attr_{1_p}, \ldots, Attr_{n_p}, \ldots)$$

where r is the relation name following the keyword INSERT INTO in the given query Q that includes SQ.

On the other hand, for each non-atomic attribute NA_j specified in C_j ($1 \leq j \leq m$) in the **Select** clause, there are two different cases (depending on the type of NA_j) to be considered.

1. NA_j is the Embedded-Relation-Name ern with attributes $B_1, \ldots, B_m$.

   ern is transformed into a set term which forms an argument ern of the head tuple term r. The subscript q, which is given to the relation s within which ern is embedded during the transformation of the **From** clause in SQ, should have been appended to attributes $B_1, \ldots, B_m$ in ern to yield $B_{1_q}, \ldots, B_{m_q}$. This occurs because s must be specified in the **FROM** clause of SQ, and the subscript q in B_{i_q}, $1 \leq i \leq m$, is handled as discussed earlier. $B_{1_q}, \ldots, B_{m_q}$ are included as arguments of the tuple term ern', that is generated by the transformation algorithm, in ern (If m = 1, then ern' is ignored.). Once again, the optional reference name specified in C_j does not affect the transformation of NA_j for the same reason mentioned in the case of transforming A_i. Hence, the transformed C_j yields either

   $$ern\{ern'(B_{1_q}, \ldots, B_{m_q})\} \text{ or } ern\{B_{1_q}\}$$

   in $r(\ldots, ern\{ern'(B_{1_q}, \ldots, B_{m_q})\}, \ldots)$ or $r(\ldots, ern\{B_{1_q}\}, \ldots)$.

2. C_j is an (SFW-Expression) AS Column-Name.

   The **Select** clause of the SFW-Expression (which is an incremental subquery) ISQ in SQ is transformed into a structure called *inc-select* which has the set term Column-Name cn. The attributes $Att_1, \ldots, Att_n$, which are transformed from the attributes specified in the **Select** clause of ISQ according to the transformation approach discussed in this subsection, are included as arguments of the tuple term cn' in cn. (If n = 1, then cn' is ignored.) *Inc-select* is then included as an argument of the head tuple term r of RE. Each embedded relation r_i ($1 \leq i \leq n$) listed in the **From** clause of ISQ is associated with a subscript which is given to the relation within which r_i is embedded during the transformation of the **FROM** clause in SQ. If ISQ includes a **Where** clause W, then W is transformed and included in the body of RE as detailed in the transformation of the **Where** clause. Hence, the transformed C_j yields either

   $r(\ldots, cn\{cn'(Att_1, \ldots, Att_n)\}, \ldots) :- \ldots,$ **[transformed-where-clause]**, $\ldots$.

   or

   $r(\ldots, cn\{Att_1\}, \ldots) :- \ldots,$ **[transformed-where-clause]**, $\ldots$.

   in RE.

B.1.4 Transforming SQL^*/NR subqueries with Aggregates into Rule Expressions

The process of transforming the **From** and **Where** clauses of an SQL^*/NR subquery with aggregates is exactly the same as that for an SQL^*/NR subquery without aggregates as discussed in the previous subsections. We choose to adopt similar notation used by

[SR91, Gel93] to represent the **group by** construct and aggregate operators in an LDL/NR rule expression. The following is the general form of a **group by** subgoal (tuple term) that is to be included in the body of a rule expression RE. This subgoal contains the transformed aggregate constructs specified as arguments in the **Select** clause of the *Aggregate-subquery*:

$$\text{group-by}(\text{r(t)}, [X_1, \ldots, X_m], [Y_1 = Agg_1\ ([\text{DISTINCT}]\ (Attr_1)), \ldots,$$
$$Y_n = Agg_n\ ([\text{DISTINCT}]\ (Attr_n)])$$

where

- r is the relation to which the grouping is applied and is included in the **From** clause of the *Aggregate-subquery*.

- t represents the arguments in r which are either constants, variables, or set terms.

- The grouping list $X_1, \ldots, X_m$ consists of one or more variables that must appear in t and on which the grouping is based. ($X_1, \ldots, X_m$ are the attributes specified in the **group by** clause of the *Aggregate-subquery*.)

- The list of $Y_1, \ldots, Y_n$ is the aggregation list. Each Y_i, $1 \leq i \leq n$, is a new variable (attribute) which is the reference name following an aggregate operator and the AS keyword in the **Select** clause of the *Aggregate-subquery*. $Y_1, \ldots, Y_n$ are included as arguments in the head tuple term of RE.

- $Attr_1, \ldots, Attr_n$ are attribute names in r to which aggregate operators $Agg_1, \ldots, Agg_n$ (e.g. sum and count) are applied, respectively. A built-in predicate for each one of the aggregate operators that is represented by $Agg_1, \ldots, Agg_n$ (e.g., MIN, MAX, SUM, AVG, and COUNT), respectively, in the **group by** subgoal is provided in LDL/NR.

- Within the body of RE, the **group by** subgoal represents a relation over attributes (variables) $X_1, \ldots, X_m$ in the grouping list and attributes (variables) $Y_1, \ldots, Y_n$ in the aggregation list.

B.2 Transforming an SQL^*/NR Query into Rule Expressions

As mentioned in Section 2.2, an SQL^*/NR query Q without aggregates consists of one *Basis-subquery* and n ($0 \leq n$) different *Recursive-subqueries*. Since *Basis-Subquery* and *Recursive-subqueries* are both SQL^*/NR subqueries, each one of these subqueries is transformed as discussed in Section B.1. The relation name R in the INSERT INTO statement of Q denotes the name of the resultant relation, and is used as the head tuple term of every rule expression generated for each subquery of Q as mentioned in Appendix B.1.3.

The transformation of the *Basis-subquery* and *Recursive-subquery* in the **INSERT INTO** statement of an SQL^*/NR query Q with aggregates is exactly the same as discussed in the previous paragraph. The transformation of the *Aggregate-subquery* in the **THEN INTO** statement of Q has been described in the previous subsection. The relation name R_2 specified after the **THEN INTO** keyword denotes the name of the resultant relation which is constructed by using the temporary relation R_1 specified in the **INSERT INTO** statement, and is used as the head tuple term of the rule expression resulting from transforming the *Aggregate-subquery* in Q. The attributes of R_2 include all the atomic, non-atomic, and aggregate-attributes specified in the **Select** clause of the *Aggregate-subquery*.

References

[CC90] Q. Chen and W. Chu. *Deductive and Object-Oriented Database*, chapter HILOG: A High-order Logic Programming Language for Non-1NF Deductive Databases, pages 431–452. Elsevier Science Publishers, 1990. W. Kim, et al. (Editors).

[CK91] Q. Chen and Y. Kambayashi. Nested Relation Based Database Knowledge Representation. In *Proceedings of 1991 ACM SIGMOD International Conference on Management of Data*, pages 328–337. ACM, 1991.

[Gel93] A. V. Gelder. Foundation of Aggregation in Deductive Databases. In *Proceedings of the 3rd Intl. Conference on Deductive and Object-Oriented Databases*, pages 13–33. Springer-Verlag, 1993. Lecture Notes in Computer Science, 470.

[GM92] J. Grant and J. Minker. The Impact of Logic Programming on Databases. *Communications of ACM*, 35(3):67–81, March 1992.

[GN90] H. Gallaire and J.-M. Nicolas. Logic and databases: An assessment. In *the Third International Conference on Database Theory*, pages 177–186. Springer-Verlag, December 1990. Lecture Notes in Computer Science, 470.

[Hul86] Richard Hull. Relative Information Capacity of Simple Relational Database Schemata. *SIAM Journal of Computing*, 15(3):856–886, August 1986.

[KC93] K. Koymen and Q. Cai. *SQL**: A Recursive SQL. *Information Systems*, 18(2):121–128, 1993.

[KS91] H. Korth and A. Silberschatz. *Database System Concepts, Second Edition.* McGraw-Hill, Auckland, 1991.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming, Second, Extended Edition.* Spring-Verlag, New York, 1987.

[LN95a] S. J. Lim and Y-K Ng. Set-Term Matching in a Logic Database Language. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications*, pages 189–196, Singapore, April 1995.

[LN95b] S. J. Lim and Y-K Ng. Set-Term Unification in a Logic Database Language. In *Proceedings of the 1st Annual International Computing and Combinatorics Conference (to appear)*, Xian, China, August 1995. Springer-Verlag.

[Qar95] N. Qaraeen. SQL*/NR: A Recursive Query Language for Nested Relations. Master's thesis, Brigham Young University, Provo, Utah, May 1995.

[RKB87] M. Roth, H. Korth, and D. Batony. SQL/NF: A Query Language for ¬1NF Relational Databases. *Information Systems*, 12(1):99–114, 1987.

[RKS88] M. Roth, H. Korth, and A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.

[SR91] S. Sudarshan and R. Ramakrishman. Aggregation and Relevence in Deductive Databases. In *Proceedings of 17th VLDB.*, pages 501–511, Barcelona, Spain, 1991.

[STZ92] O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of Set Terms in the Logic Data Language (LDL). *Logic Programming*, 12(1):89–119, 1992.

[Uni91] UniSQL. *UniSQL/X Database Management System User's Manual.* UniSQL Inc., Austin, Texas, 1991.