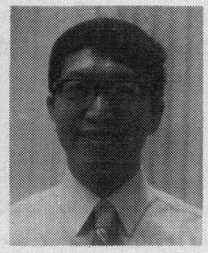
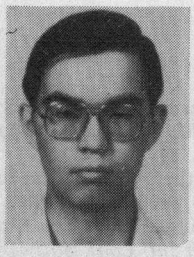


Member of IBM's Santa Teresa Laboratory during the summers of 1982 and 1983. He has published papers in switching and automata theory, computer graphics, operating systems, database, and most recently, software engineering and software metrics.

Stephen M. Thebaut (S'79-M'83) received the B.A. degree in mathematics from Duke University, Durham, NC, and the M.S. and Ph.D. degrees in computer science from Purdue University, West Lafayette, IN, in 1979 and 1983, respectively.

He joined the Department of Computer and Information Sciences, University of Florida, Gainesville, FL, as an Assistant Professor in 1983, and was supported by an IBM Postdoctoral Research Fellowship during the 1983-1984 academic year. His current research interests include software reliability, program maintenance, and the large-scale software development process.

Prof. Thebaut is a member of the Association for Computing Machinery, the IEEE Computer Society, and the 356 Registry.



Tze-Jie Yu received the B.S. degree in electrical engineering from National Taiwan University in 1979, and the M.S. degree in computer science from Purdue University, West Lafayette, IN, in 1983.

From 1981 to the present, he has been a Ph.D. candidate and Research Assistant at Purdue University. He worked for IBM's Santa Teresa Laboratory as a Systems Analyst during the summer of 1983. His research interests include software quality assessment, software tools, and software cost modeling.

Mr. Yu is a member of the Association for Computing Machinery and its Special Interest Group on Software Engineering, and a member of the IEEE Computer Society.

Lorri R. Paulsen received the B.S. degree in mathematics from Wagner College, Staten Island, NY, in 1968.

She joined IBM in 1968, and is currently a Program Development Manager for IBM General Products Division at Santa Teresa Laboratory, San Jose, CA. During her IBM career, she has been associated with systems design and development, working on the ASP Version 3 and JES3 products. In 1979 she became involved with the measurement of programmer's productivity, which led to her research with Software Science metrics and the prediction of error-prone modules. She is currently managing the development of software engineering tools to assist with the design and implementation of IBM products.

Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries

STEFANO CERI AND GEORG GOTTLOB

Abstract—**In this paper, we present a translator from a relevant subset of SQL into relational algebra. The translation is syntax-directed, with translation rules associated with grammar productions; each production corresponds to a particular type of SQL subquery.**

The translation is performed in two steps, associated with two different grammars of SQL. The first step, driven by the larger grammar, transforms SQL queries into equivalent SQL queries that can be accepted by a restricted grammar. The second step transforms SQL queries accepted by the restricted grammar into expressions of relational algebra. This approach allows performing the second step, which is the most difficult one, on a restricted number of productions.

The translator can be used in conjunction with an optimizer which operates on expressions of relational algebra, thus taking advantage of a body of knowledge on the optimization of algebraic expressions. Moreover, the proposed approach indicates a methodology for the correct specification and fast implementation of new relational query languages. Finally, the translator defines the semantics of the SQL language, and can be used for the proof of equivalence of SQL queries which are syntactically different.

Index Terms—**Program translation, query equivalence, query languages, query optimization, relational algebra, relational database model, SQL.**

Manuscript received June 13, 1984; revised September 6, 1984. This work was supported in part by a grant from Data Base Informatica.

The authors are with the Dipartimento di Elettronica, Politecnico di Milano, I 20133 Milano, Italy.

I. Introduction

THE use of the relational model and languages is becoming more and more popular for the development of new database management systems. Formal languages, such as the relational calculus and algebra, have been proposed by Codd [1]

0098-5589/85/0400-0324$01.00 © 1985 IEEE

and later developed in the literature of database management systems [2], [3]. In particular, relational algebra can be used as a model for describing many approaches to query optimization [3]-[6], [16]-[19].

Although some earlier proposals for relational manipulation languages have been in fact based on relational algebra [7] or calculus [8], the trend in the design of relational languages has at some point moved toward a different direction; this was probably due to the success of the SQL query language [9], developed for the System R prototype [10], which has become *de facto* a standard used by many other systems [11]-[13].

SQL is an easy-to-use language with a number of high-level constructs. In general, small SQL queries are quite easy to understand; however, SQL allows writing very complex queries, whose meaning is quite obscure, and whose optimization is extremely hard [14]. Sometimes, two complex queries in SQL happen to have the same meaning, although they are very different from a syntactic point of view. In this paper, we propose a method for translating SQL queries into expressions of relational algebra. This method also suggests an approach to the design of new query languages.

SQL has a basic structure based on the query block or subquery:

"SELECT target_list FROM relation_list WHERE predicate".

If the predicate corresponds to a selection formula in the sense of Ullman [2], then a query block is easily mapped to a relational algebra expression. However, SQL allows several features which are not trivially mapped:

1) the nesting of several query blocks

2) the use of keywords such as: EXISTS, ALL, ANY, IN in the connection between query blocks

3) the use of variables and of predicates connecting query blocks indirectly

4) the use of set comparison operations between two query blocks.

5) the grouping of tuples of a relation into subsets and the evaluation of aggregate functions on each subset.

In this paper, we show how all these features can be mapped to expressions of relational algebra.

Relational algebra has been preferred to relational calculus as target of our translator for the following reasons:

1) It is more procedural, because algebraic expressions also give the order of application of operations in the computation of the query; thus, it is a more appropriate model for query optimization and system implementation.

2) Many existing approaches to query optimization and equivalence are (or can easily be) expressed in relational algebra.

3) Relational algebra uses relations (or sets) as operands, while calculus is based on tuple variables. For the optimization of queries, particularly with distributed environments or special-purpose database machines, set-oriented models are more appropriate than tuple-oriented models [19].

We have used standard relational algebra, with the only addition of a new operation for the evaluation of aggregate functions, which was required by the last feature in the above list.

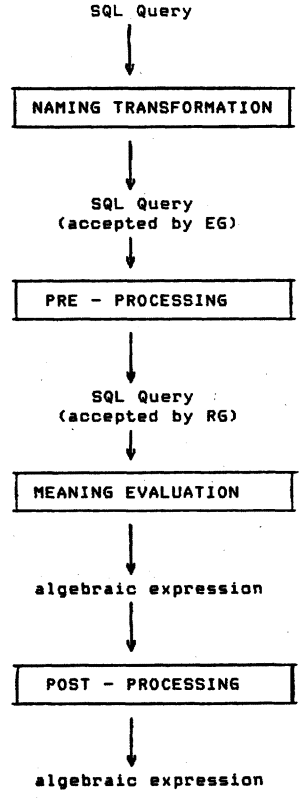


Fig. 1. Overall architecture of the translator.

A. Overview of the Proposed Approach

The overall architecture of the translator is shown in Fig. 1.

The translator is based on two grammars for the SQL language: an *extended grammar* EG, and a *restricted grammar* RG. EG does not accept all the statements accepted by the SQL grammar of [9], but it has approximately the same expressive power; it also includes some elements (such as the EXISTS and ALL keywords in the predicates) which were not present in the grammar of [9]. In the sequel, we will be more specific on the SQL statements which are not accepted by EG. The main difference between the grammar of [9] and EG is that EG allows a correspondence between types of SQL subqueries and productions of the grammar, thus allowing a syntax-directed translation [15]. RG has a smaller number of productions than EG, representing few types of SQL subqueries.

The queries which are accepted by the EG but not by the RG are *preprocessed* and transformed into queries which are accepted by the RG; thus, the preprocessing consists in a transformation from SQL into SQL. For the queries which are accepted by the RG, we have defined the *meaning* in relational algebra; thus, the meaning of a query consists in a translation from the query in SQL into an expression of relational algebra. Both the definitions of the preprocessing and of meaning are syntax-directed: rules are in correspondence with the productions of the EG and RG. The rationale for distinguishing the preprocessing from the meaning is that the meaning of an SQL subquery is, in general, rather difficult. Thus, we have determined a small set of subqueries which are representative of all other subqueries, and we have given the meaning just for them.

The preprocessing and meaning are the heart of our approach; however, before and after their applications, two other simple string transformations are applied to the input query in SQL and to the output expression in relational algebra. The former transformation is a *naming transformation:* attribute names are extended with relation names and variables are substituted with pairs relation-name.attribute-name. The latter transformation, called *postprocessing*, eliminates some operations which have no effect (such as the projection over the entire attribute schema of a relation) and performs some trivial query optimization transformations.

B. An Example of Translation from SQL to Relational Algebra

We assume that the reader is familiar with relational algebra and with the SQL language. Let us consider the relational schema:

S(*S#*,NAME), storing supplier's name and number

SP(*P#*,*S#*), storing supplier and part numbers for a supply and a typical SQL query with nesting, taken from Date's book [3, sect. 7.2.14]. The query retrieves the name of suppliers who do not supply product '*P2*':

```
SELECT NAME
FROM S
WHERE 'P2' ≠ ALL
   SELECT P#
   FROM SP
   WHERE S#=S.S#
```

Notice that the clause ALL is used to indicate that only those suppliers should be retrieved for which there is *no* supply tuple with *P#* equal to *P2*; if the ALL clause were omitted, then the query would retrieve suppliers having *at least one* supply tuple with *P#* different than *P2*.

The naming transformation produces the following query, in which attribute names are extended with relation names:

```
SELECT S.NAME
FROM S
WHERE 'P2' ≠ ALL
   SELECT SP.P#
   FROM SP
   WHERE SP.S#=S.S#
```

This query can be parsed using EG; the preprocessing produces as result the equivalent query (without the ALL keyword):

```
( SELECT S.NAME FROM S
  MINUS
  SELECT S.NAME
  FROM S
  WHERE 'P2' =
   SELECT SP.P#
   FROM SP
   WHERE SP.S#=S.S# )
```

This query can be parsed using RG; thus, its meaning can be evaluated. We anticipate the notation for some algebraic operations from Section II-A:

PJ[*A*]	(projection over a set *A* of attributes)
SL[*p*]	(selection of tuples satisfying predicate *p*)
CP	(Cartesian product)
DF	(difference)
JN[jp]	(join with join predicate jp)

We use a prefix notation for unary operations (e.g., PJ and SL), and an infix notation for binary operations (e.g., CP, DF, and JN). The meaning evaluation produces the following expression:

```
(PJ[S.SNAME] S) DF
(PJ[S.SNAME] SL[SP.P#='P2']
(PJ[SP.P#,S.NAME,S.S#] SL[SP.S#=S.S#] (S CP SP) ))
```

Finally, the postprocessing step is applied to the above expression. The postprocessing eliminates useless projections and transforms selections over Cartesian products into joins, giving:

```
(PJ[S.SNAME] S) DF
(PJ[S.SNAME] ((SL[SP.P#='P2'] S) JN[SP.S#=S.S#] SP))
```

C. Applications of This Approach

The proposed approach can be applied in several ways.

1) The output of the translator can be regarded as the input for an *optimizer* (in the implementation of a system, translation and compilation could be integrated within the compiler of the language). There is a body of knowledge on the optimization of relational queries, both in centralized [4]-[6] and distributed [16]-[18] systems, which assumes the expressions of relational algebra as input to the optimization. The algorithm presented in [4] does not immediately operate on algebraic expressions; however, in [3, p. 274], it is shown how to express the inputs to the algorithm algebraically. Some implemented optimizers [31], [32] consider in the optimization several features which cannot be expressed in algebraic form (such as the index selection or the join method used for the execution of the query); however, they perform implicitly some algebraic transformations (such as the early evaluation of selections and projections). Thus, the development of the translator is very helpful for an efficient and fast implementation of a compiler for SQL.

2) The translation of SQL into expressions of relational algebra can be regarded as the definition of the *semantics* of SQL. This definition is important, because the interpretation of certain queries that can be written in SQL is not at all obvious.

3) The translation mechanism is useful for *proving the equivalence* between SQL queries. In SQL, it is possible to write queries which are syntactically very different, but have the same meaning [9]. By translating two SQL queries into expressions of relational algebra, it is possible to prove their equivalence, either because the two queries are immediately mapped to the same expression, or because it is possible to apply equivalence transformations to expressions in order to reduce one expression to the other one, hence proving the equivalence. Although there is no completeness result which indicates that any two equivalent expressions can be discovered, the use of equivalence transformations of relational algebra [2], [19] allows determination of most of them.

Note that most algebraic expressions produced by our translator can be converted into tableaux [3], and therefore tableaux techniques in [33]-[35] can be applied for the optimization or for the proof of equivalence.

The proposed approach suggests a *method for the design of new relational query languages or interfaces.*

1) A precise definition of the semantics allows to avoid ambiguities in the specification of the language.

2) The use of an algebraic form as internal representation of queries allows use of the query optimization mechanisms already developed in the literature of relational database systems.

Moreover, the two-step approach of the translator (preprocessing followed by the use of the meaning) indicates a method for developing the translator which merits attention; its major appeal is in determining a restricted number of productions of the language which really require having an associated meaning, while all other productions are reduced to them.

D. Limitations

In this paper, there is no claim to cover the complete expressive power of SQL; we can indicate properly the features which are not dealt with by our approach, and motivate our choices.

1) We only consider the data manipulation section of SQL, while we omit considering updates and data definition. The motivation of our choice is that both updates and data definition are, in general, very easy to understand; however, their translation to algebra is not straightforward or feasible. Consider also that, in typical SQL update statements, most of the complexity is due to the specification of the tuples to be updated, while the update rule is given by simple expressions. The specification of tuples to be updated is done by using SQL retrieval statements and, thus, can be dealt with by our translator.

2) Relational algebra does not deal with duplicates, while SQL allows the user to control duplicate elimination (via the SELECT or SELECT UNIQUE keywords). Since we translate SQL to relational algebra, we cannot leave this alternative: duplicates are always eliminated. In practice, we consider queries as if they always had the SELECT UNIQUE keyword; for brevity, we omit to indicate UNIQUE in queries' expressions. Reference [20] presents an extension of relational algebra with control over duplication; extensions of our approach in the direction of [20] are feasible.

3) Relational algebra does not control the order in which tuples are produced; consequently, we do not consider the ORDER BY clause of SQL. Moreover, in a query with the GROUP BY clause, it is not possible to control that tuples of the same group be contiguously listed in the result relation.

4) A subquery with the GROUP BY clause can be nested at any level of an SQL query according to the grammar of [9]; however, we have considered only SQL queries in which the GROUP BY clause applies to the top level subquery. Moreover, we omit considering the SET constructor in the HAVING clause of a query with GROUP BY.

5) We omit considering, for brevity, a few syntactic aspects of SQL. For instance, we do not consider the predicates "expression BETWEEN expression AND expression," or the use of separators "<" and ">" to denote lists of attributes or constants.

As we have noticed already, we need to introduce a new operation for dealing with aggregate functions. Although extensions of relational algebra with similar features have been studied already [21], [22], the power of equivalence transformations of expressions with function evaluation has not been fully investigated; this is a limitation to the capability of optimizing a query and proving the equivalence of two queries with our approach.

Finally, the use of two major phases in the translation (preprocessing and evaluation of the meaning) corresponds to a loss of efficiency compared to a single-step translator; however, our approach is justified by the major simplicity and clarity of the overall translation process.

E. Previous Related Work

This paper is related to several works in the following areas.

1) *Query optimization:* many works have studied the optimization of relational algebra expressions, which motivates the use of the translator. Among them, see [4]-[6], [16]-[18], [33]-[35].

2) *Other extensions to relational algebra:* [20]-[23].

3) *Other approaches to the translation of SQL:* [14], [24].

4) *Program conversion*, particularly the notion of equivalence between programs: [25]-[26].

Kim, in [14], investigated the optimization of SQL queries with nesting of subqueries. He proposed a translation from SQL into SQL, transforming complex queries into several related simpler queries. Reference [14] has in common with this paper the approach of classifying the possible types of nesting and of associating a solution scheme to each type; it differs from our approach in the following key features.

1) The translation is not syntax-directed; it is given informally on the ground of specific algorithms for each type of nesting.

2) The translation is built ad hoc for the SQL optimizer. In fact, the motivation of the transformations is to reduce the number of page fetches required by the optimizer.

3) The translation does not improve the understanding of the semantics of an SQL query. Generally, the translation of a complex query in [14] produces several small queries such that each of them (except the one which produces the result) is assigned to a temporary relation, which is then used to build the result. This fragmented form of an SQL query does not necessarily improve the understanding of its semantics.

Klug, in [22], introduced an extension to algebra and calculus for supporting aggregate functions, and showed the equivalence between the two definitions; our algebraic definition is close to that of [22]. Aggregate functions were also used by Ceri and Pelagatti in [23] in a multirelational model particularly suited to distributed databases, however, with a more complex formalism which is not useful for this paper. Furtado and Kerschberg have designed in [21] a quotient algebra which includes the possibility of dividing relations into groups, but does not deal with aggregate functions.

McLeod, in [24], presented a translator from QBE [27] into SQL. This was done for "having a consistent semantics of the two languages." Given that QBE does not have many of the difficulties of SQL, this translation is rather simple.

F. Structure of the Paper

Section II of this paper presents the extensions to relational algebra and some useful notation for our approach. Section III presents the first two steps of the translation, naming transformation and the preprocessing. Section IV presents the meaning and the postprocessing. Naming transformation and postprocessing, which are rather simple, are presented informally. Section V presents the use of the proposed approach for proving the equivalence between SQL queries. Section VI presents the conclusions. In order to show that our approach covers most of the SQL queries, the Appendix presents the translation into expressions of relational algebra of the SQL queries of Date's book [3].

II. Definitions and Notations

In this paper, we use the traditional operations of relational algebra [2] with the notation of [19]:

PJ[A]	(projection over a set A of attributes)
SL[p]	(selection of tuples satisfying predicate p)
CP	(Cartesian product)
UN	(union)
DF	(difference)
IN	(intersection)
JN[jp]	(join with join predicate jp)
SJ[jp]	(semijoin with join predicate jp).

In [2] it is shown how the last three operations can be derived from the first five basic operations.

In order to model aggregate function evaluation, we introduce another basic operation, called *function evaluation*. The operation applies to one relation R and is denoted as follows:

$$\mathrm{FN}[F1(B1), F2(B2), \cdots, Fn(Bn); A]\ R$$

Intuitively, the operation FN extends the attribute schema of R with new attributes corresponding to the functions $F1(B1)$, $F2(B2), \cdots, Fn(Bn)$. The functions are applied to the subsets of tuples of R having the same value for the attributes A. More properly:

1) $\mathrm{FN}[F(B); \phi]\ R$ is defined as

$$\mathrm{FN}[F(B); \phi]\ R = R\ \mathrm{CP}\ R^{F(B)}$$

where $R^{F(B)}$ is a relation having only one attribute whose name is "$F(B)$" and only one tuple whose value is given by evaluating the aggregate function $F(B)$ on R. If no function is specified, then the operation has no effect:

$$\mathrm{FN}[\phi; \phi]\ R = R$$

2) $\mathrm{FN}[F(B); A]\ R$ is defined as

$$\mathrm{FN}[F(B); A] = \bigcup_{t \in R} \mathrm{FN}[f(B); \phi]\ (\mathrm{SL}[A = t.A]\ \mathrm{R})$$

where t denotes a tuple of R and $t.A$ denotes the value(s) taken by attribute(s) A in R. We further define

$$\mathrm{FN}[\phi; A]\ R = R.$$

3) $\mathrm{FN}[F1(B1), F2(B2), \cdots, Fn(Bn); A]\ R$ is an abbreviation for

$$\mathrm{FN}[F1(B1); A]\ (\mathrm{FN}[F2(B2); A](\cdots \mathrm{FN}[Fn(Bn); A]\ R))).$$

The typical aggregate functions used in query languages are: AVG, MIN, MAX, SUM, COUNT. We do not enter in this paper in the definition of the semantics of aggregate functions, which is the same as in relational query languages. An example of application of the FN operation is shown in Fig. 2.

ENO	SAL	DNO
1	100	1
2	150	2
3	130	1
4	170	3
5	120	2
6	160	1
7	170	3

Relation EMP (ENO,SAL,DNO)

ENO	SAL	DNO	AVG(SAL)
1	100	1	130
2	150	2	135
3	130	1	130
4	170	3	170
5	120	2	135
6	160	1	130
7	170	3	170

Relation produced as result of FN[AVG(SAL),DNO] EMP

Fig. 2. Example of application of the FN operation.

TABLE I
Summary of Notation

car(set of relational expressions) indicates the cartesian product of all the relational expressions; if the argument is a singleton, car returns the unique relational expression.
attr(set of relations) is the set of the attributes in the attribute schemas of the specified relations.
rels(set of attributes) is the set of database relations having the specified attributes.
a(relational expression) is the set of attributes occurring in the result produced by the evaluation of a relational expression.
r(relational expressions) is the set of the database relations occurring in a relational expression.
extrattr(predicate) is the set of attributes which appear in a predicate.
extrel(predicate) is the set of database relations whose attributes appear in the predicate: extrel(p) = rels(extrattr(p)).
empty(string) is a boolean function which returns false if it is applied to the empty string, true otherwise.

In this paper, we will make use of some general notation, which is summarized rather informally in Table I.

We also need to introduce an abbreviation for expressions which use the binary operators UN, DF, IN. Let BIN represent a binary operation; we use

$$\mathrm{Rel\_expr}_1\ \mathrm{BIN}[A1; A2]\ \mathrm{Rel\_expr}_2$$

where $\mathrm{Rel\_expr}_1$ and $\mathrm{Rel\_expr}_2$ indicate two generic expressions and $A1$, $A2$ are attributes appearing respectively in $\mathrm{Rel\_expr}_1$ and $\mathrm{Rel\_expr}_2$, to indicate the following formula:

car({Rel_expr$_1$} ∪ (rels($A2$) - rels(Rel_expr$_1$))) BIN

car({Rel_expr$_2$} ∪ (rels($A1$) - rels(Rel_expr$_2$)))

The use of this abbreviation will become clear in the discussion of meanings. Note that we use the symbol "∪" to denote the union operation in the traditional, set-theoretic sense, and the symbol "UN" to denote the algebraic union.

III. Naming Transformation and Preprocessing

Naming transformation is applied to an SQL query in order to eliminate some syntactical problems and put it into a form that can be accepted by the extended grammar. By effect of the transformation:

1) Relation names which refer to different instances of the same base relation are distinguished by using apostrophes. Thus, a query

SELECT A FROM R WHERE B = SELECT C FROM R

becomes

SELECT A FROM R' WHERE B = SELECT C FROM R''.

2) Attribute names are extended with the relation name of the relation to which they refer. The above example becomes

SELECT $R'.A$ FROM R' WHERE $R'.B$ = SELECT $R''.C$ FROM R''.

3) Variables are eliminated. In SQL, variables are associated with relations in the FROM clause, as shown from the following example:

SELECT $X.A$
FROM R X, R Y
WHERE $X.B$=$Y.B$

where X and Y denote two copies of relation R. It is possible to associate variables to relation names with apostrophes; then, variables in the attribute names are replaced with relation names with apostrophes. In the example, let us associate X with R' and Y with R''; then, we have

SELECT $R'.A$
FROM R', R''
WHERE $R'.B$=$R''.B$

The *extended grammar* EG is shown in Fig. 3. As already indicated in the section on the limitations of this approach, the EG does not accept all SQL queries, but just a relevant subset of them. For all the accepted queries, the translator will produce an expression in relational algebra. The grammar is a standard BNF, with vertical bars denoting alternatives, and square brackets denoting optionalities. The suffix "_list" denotes a list of symbols; the terminal symbols <attribute>, <relation>, and <function> indicate the identifiers used in the grammar.

The *preprocessing* is a syntax-directed transformation of SQL queries into SQL queries; transformations are associated with the productions of EG denoted by an asterisk, corresponding to the subqueries: *in*, *in-set*, *not in*, *all*, *any*, *not-exists*, *contains*, *does-not-contain*, *set-equality*, *set-inequality*, *compound*. Each transformation is specified by rewriting the right-hand side of the production in such a way that all nonterminal symbols appear in the rewriting, while terminal symbols are typically different. For simplicity, we assume that whenever a transformation is applied, the rewriting produces a new SQL string which is parsed again from the beginning, although this is not required in many cases. The grammar has been carefully designed in order to have a translation which is independent of the parsing method used.

By effect of the preprocessing, the SQL subqueries accepted by EG are reduced to five basic SQL subqueries, which appear in the restricted grammar RG: *simple*, *group-by*, *complex* (which allows nestings), *exists* (which links two subqueries via an existential predicate), and *binary* (with union, difference and intersection). Thus, the rationale of the preprocessing is to reduce, by means of set-theoretic transformations, all the subqueries to these five basic subqueries.

In the definition of preprocessing and of the meaning, Q indicates a subquery that can be parsed by the considered production; $Q.$<non_terminal> denotes the string which is parsed by <non-terminal> in the right part of the production. $Q.$<non_terminal$_1$>.<non-terminal$_2$> denotes the string which is parsed by <non-terminal$_2$> in the production having <non-terminal$_1$> in its right part; this notation can be naturally extended to an arbitrary number of nonterminals.

1) In query. This subquery is associated with the <in_query> production of EG in Fig. 3:

<in_query> ::= SELECT <selector> FROM <relation_list>
WHERE <left_term> IN <ngb_query>

In this case, the preprocessing simply substitutes '=' for the keyword IN. By effect of the preprocessing, Q is rewritten as follows:

SELECT $Q.$<selector> FROM $Q.$<relation_list>
WHERE $Q.$<left_term> = $Q.$<ngb_query>

Thus, an *in* subquery is transformed into a *complex* subquery, accepted by RG.

2) In-set query. This subquery requires that an attribute take any value within a set of constants; it can be rewritten with a disjunctive expression in which the attribute is compared to each constant independently:

SELECT $Q.$<selector> FROM $Q.$<relation_list>
WHERE (. . . ($Q.$<attribute_spec> = $Q.$<constant1> OR
$Q.$<attribute_spec> = $Q.$<constant2> OR) . . .
$Q.$<attribute_spec< = $Q.$<constantn>)

Thus, an *in-set* subquery is transformed into a *simple* subquery, accepted by RG.

3) Not-in query. This subquery requires that an attribute take a value which is not within a given set of values, returned

```
---------------------------------------------------------------------------

  <simple_query> ::= SELECT <selector> FROM <relation_list>
                     [ WHERE <simple_predicate> ]

  <gb_query> ::= <unary_query> GROUP BY <gb_attr> [ HAVING <hav_condition>]

  <exists_query> ::= SELECT <selector> FROM <relation_list>
                     WHERE <exists_predicate>

  <complex_query> ::= [ELECT <selector> FROM <relation_list>
                      WHERE <complex_predicate>

  <binary_query> ::= ( <ngb_query> <set_op> <ngb_query> )

  <gb_attr> ::= <attribute_spec_list>

  <hav_condition> ::= <function_spec> <comp_op> <constant> |
                      <function_spec> <comp_op> <ngb_query>

  <selector> ::= <attribute_spec_list> <function_spec_list> | *

  <left_term> ::= <constant> | <attribute_spec>

  <simple_predicate> ::= ( <simple_predicate> <boolean> <simple_predicate> ) |
                         <attribute_spec> <comp_op> <attribute_spec> |
                         <attribute_spec> <comp_op> <constant>

  <exists_predicate> ::= EXISTS <ngb_query>

  <complex_predicate> ::= <left_term> <comp_op> <ngb_query>

  <compound_predicate> ::= ( <simple_predicate> <boolean> <complex_part> ) |
                           ( <complex_part> <boolean> <complex_part> ) |
                           ( <compound_predicate> <boolean> <complex_part> )

  <complex_part> ::= <exists_predicate> | <complex_predicate>

  <function_spec> ::= <function> ( <attribute_spec> ) |COUNT(*)

  <attribute_spec> ::= <relation>.<attribute>

  <boolean> ::= AND | OR

  <comp_op> ::= = | ≠ | < | ≤ | > | ≥

  <set_op> ::= UNION | INTERSECT | MINUS

  <function_name> ::= SUM | MIN | MAX | AVERAGE

  <query> ::= <gb_query> | <ngb_query>

  <ngb_query> ::= <unary_query> | <binary_query> | ( <unary_query> )

  <unary_query> ::= <simple_query> | <exists_query> | <complex_query> |
                    <compound_query> | <not_exists_query> | <all_query> |
                    <in_query> | <not_in_query>  | <contains_query> |
                    <does_not_contain_query> | <set_equality_query> |
                    <set_inequality_query> | <in_set_query>

*  <in_query> ::= SELECT <selector> FROM <relation_list>
                  WHERE <left_term> IN <ngb_query>

*  <in_set_query> ::= SELECT <selector> FROM <relation_list>
                     WHERE <attribute_spec> IN <set_of_constants>

*  <not_in_query> ::= SELECT <selector> FROM <relation_list>
                     WHERE <left_term> [IS] NOT IN <ngb_query>

*  <all_query> ::= SELECT <selector> FROM <relation_list>
                  WHERE <left_term> <comp_op> ALL <ngb_query>

*  <any_query> ::= SELECT <selector> FROM <relation_list>
                  WHERE <left_term> <comp_op> ANY <ngb_query>

*  <not_exists_query> ::= SELECT <selector> FROM <relation_list>
                         WHERE NOT <exists_predicate>

*  <contains_query> ::= SELECT <selector> FROM <relation_list>
                       WHERE <ngb_query_1> CONTAINS <ngb_query_2>

*  <does_not_contain_query> ::= SELECT <selector> FROM <relation_list>
                               WHERE <ngb_query_1> DOES NOT CONTAIN
                                     <ngb_query_2>

*  <set_equality_query> ::= SELECT <selector> FROM <relation_list>
                           WHERE <ngb_query_1> = <ngb_query_2>

*  <set_inequality_query> ::= SELECT <selector> FROM <relation_list>
                             WHERE <ngb_query_1> ≠ <ngb_query_2>

*  <compound_query> ::= SELECT <selector> FROM <relation_list>
                       WHERE <compound_predicate>

---------------------------------------------------------------------------
```

Fig. 3. Extended SQL grammar; asterisks denote the productions to which a preprocessing transformation is associated.

by the target list of $Q.<$ngb_query$>$. This can restated by subtracting the tuples with any of those values. Notice that using the inequality operator ('≠') instead of the NOT IN keywords would be wrong. The transformation is

```
( SELECT Q.<selector> FROM Q.<relation_list>
  MINUS
  SELECT Q.<selector> FROM Q.<relation_list>
  WHERE Q.<left_term> = Q.<ngb_query> )
```

4) All query. This subquery requires that an attribute take a value which is, for instance, greater than all the values returned by the target list of $Q.<$ngb_query$>$; this can be restated by subtracting the tuples which satisfy the condition built with the opposite comparator, i.e., such that the attribute value is less or equal than some of the elements of the set $Q.<$ngb_query$>$. Thus,

```
( SELECT Q.<selector> FROM Q.<relation_list>
  MINUS
  SELECT Q.<selector> FROM Q.<relation_list>
  WHERE Q.<left_term> neg(Q.<comp_op>) Q.<ngb_query> )
```

where neg($Q.<$comp_op$>$) returns the opposite comparison operator (e.g., neg('$>$') = '$\leqslant$').

5) Any query. The ANY keyword can be simply ignored:

```
SELECT Q.<selector> FROM Q.<relation_list> WHERE
Q.<left_term> Q.<comp_op> Q.<ngb_query>
```

6) Not-exists query. A nonexistence condition is restated by using set difference and an existence condition:

```
( SELECT Q.<selector> FROM Q.<relation_list>
  MINUS
  SELECT Q.<selector> FROM Q.<relation_list>
  WHERE Q.<exists_predicate> )
```

7) Contains query. If a set $S1$ contains a set $S2$, there must not exist an element in $S2 - S1$; not-existence is then rewritten in terms of a set difference and an existence condition (it would be possible to use the not-existence condition, and then to apply the preprocessing one more time):

```
( SELECT Q.<selector> FROM Q.<relation_list>
  MINUS
  SELECT Q.<selector> FROM Q.<relation_list> WHERE EXISTS
  ( Q.<ngb_query_2> MINUS Q.<ngb_query_1> ) )
```

8) Does-not-contain query. This case is similar to the previous one: the set $S1$ does not contain the set $S2$ if there exists an element in $S2 - S1$. Thus

```
SELECT Q.<selector> FROM Q.<relation_list> WHERE EXISTS
( Q.<ngb_query_2> MINUS Q.<ngb_query_1> )
```

9) Set-equality query. The equality between two sets $S1$ and $S2$ requires that both $S1$ contain $S2$ and $S2$ contain $S1$:

```
SELECT Q.<selector> FROM Q.<relation_list> WHERE
  ( Q.<ngb_query_1> CONTAINS Q.<ngb_query_2> AND
    Q.<ngb_query_2> CONTAINS Q.<ngb_query_1> )
```

10) Set-inequality query. The inequality between two sets $S1$ and $S2$ requires that either $S1$ does not contain $S2$ or $S2$ does not contain $S1$ (or both):

```
SELECT Q.<selector> FROM Q.<relation_list> WHERE
  ( Q.<ngb_query_1> DOES NOT CONTAIN Q.<ngb_query_2> OR
    Q.<ngb_query_2> DOES NOT CONTAIN Q.<ngb_query_1> )
```

11) Compound query. A compound subquery contains a Boolean combination of expressions such that one or both expressions are in turn subqueries. They are reduced to queries with binary operations INTERSECTION and UNION. Notice that a <compound_predicate> can be parsed according to three different productions, each one with three different nonterminals, such that the first and last of these nonterminals are different. The transformation can be given regardless of the particular production used in the parsing of Q. Thus, $Q.<$compound_predicate$>.i$, with $i = 1$ or $i = 3$, indicates the ith nonterminal used with any of the parsings of <compound_predicate>. The transformation is:

```
( SELECT * FROM Q.<relation_list>
  WHERE Q.<compound_predicate>.1
  T(Q.<compound_predicate>.<boolean>)
  SELECT * FROM Q.<relation_list>
  WHERE Q.<compound_predicate>.3 )
```

where

T(AND) = INTERSECT

T(OR) = UNION

By effect of these transformations, all the subqueries of EG have been restricted to the five basic subqueries of RG; Fig. 4 summarizes the transformations. Notice that some transformations do not lead immediately to subqueries accepted by RG, but require further preprocessing.

Some care must be used in the *not-in*, *all*, *not-exists*, and *contains* subqueries, whose preprocessing produces SQL queries with a MINUS, when a *function specification* appears in the selector. In these cases, the function evaluation must be performed *after* the evaluation of the difference. This is achieved by introducing a temporary relation TEMP to which the function is applied, and then assigning to TEMP the same SQL ex-

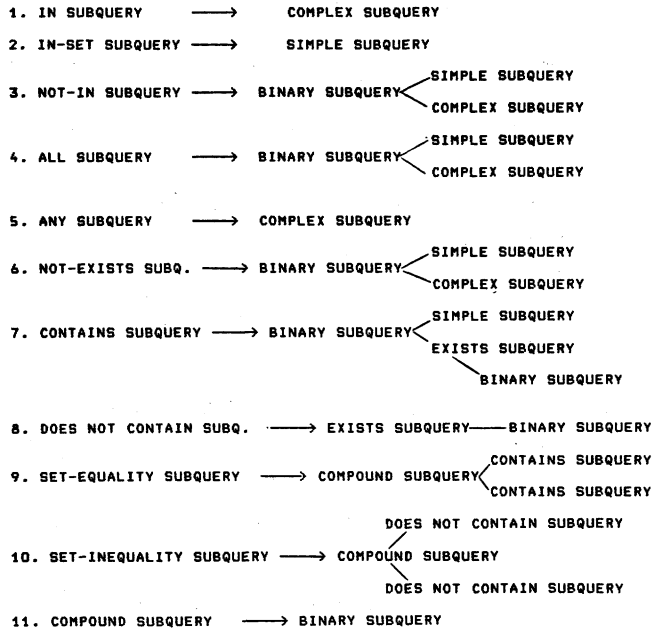


Fig. 4. Synthesis of preprocessing transformations.

pression given by the preprocessing rule in which $Q.<\text{selector}>$ is substituted by $*$. For instance, the *not-in* subquery:

SELECT $F(R.A)$ FROM R WHERE $R.B$ IS NOT IN
SELECT $S.B$ FROM S

is rewritten as follows:

SELECT $F(R.A)$ FROM TEMP
TEMP = (SELECT $*$ FROM R
MINUS
SELECT $*$ FROM R WHERE $R.B$ =
SELECT $S.B$ FROM S)

As the *final transformation* of the preprocessing, the star '$*$' in the SELECT clause, denoting all attributes, is replaced by the names of all attributes of the relation(s) appearing in the FROM clause. Given the relation $R(A, B, C)$, the query

SELECT $*$ FROM R

becomes

SELECT $R.A, R.B, R.C$ FROM R

IV. Meaning and Postprocessing

The *restricted grammar* RG is shown in Fig. 5. The *meaning* is a syntax-directed transformation of SQL queries accepted by the RG into relational algebra expressions; transformations are associated with the productions of RG. Our meaning is similar in style to the definition of the semantics of programming languages which can be found in [28], although with a more informal approach. In particular, we do not distinguish between sets and lists, and we treat individual elements as if they were singletons (or lists with one element) without requiring additional notation.

Each transformation can be either *terminal* or *nonterminal;* if a transformation is nonterminal, it incorporates in its definition the invocation of the meaning on another subquery, which appears as a nonterminal symbol of the considered production. The evaluation of nonterminal transformations may require one to have already evaluated the meaning of these subqueries; in these cases, the evaluation of the meaning proceeds bottom-up by considering nonterminal transformations first, and then considering terminal transformations which require them.

Five basic productions of RG, indicated by an asterisk, are associated in a nontrivial way to their meaning: <simple_query>, <gb_query>, <exists_query>, <complex_query>, <binary_query>. The transformation of a <simple_query> is terminal, and also the transformation of a <gb_query> is terminal when the corresponding SQL query is not nested. All other transformations are nonterminal. Two transformations are associated with unary nestings, i.e., nestings with just one subquery; they represent existential conditions (<exist_query>) or comparison conditions involving subqueries (<complex_query>). One transformation is associated with binary nestings, i.e., nestings of two subqueries related by a binary operator (UNION, MINUS, INTERSECTION).

Three productions of RG, the axiom <query>, <ngb_query>, and <unary_query> (indicated by a plus in RG), have a quite simple meaning, given by the meaning of the single nonterminal in the right-hand side. Thus:

meaning(Q: Q is a <query>) = meaning($Q.1$)
meaning(Q: Q is a <ngb_query>) = meaning($Q.1$)
meaning(Q: Q is a <unary_query>) = meaning($Q.1$)

where $Q.1$ denotes the first nonterminal of the right side of the production, as indicated in the preprocessing of compound subqueries. The five basic transformations are analyzed in the sequel.

A. Simple Query

Intuitively, a simple subquery should be mapped to an expression of the type:

$$\text{PJ}[Q.<\text{selector}>]\ \text{SL}[Q.<\text{simple\_predicate}>]\ \text{car}(Q.<\text{relation\_list}>)$$

where the function car denotes the Cartesian product as from Table I. However, the translation is more complex because of the following problems.

1) The simple predicate is optional, and might be missing in the query.

2) The selector might include aggregate functions.

3) The simple predicate might use "external" relations, i.e., relations which appear within subqueries at a higher level of nesting.

The first problem is solved by eliminating SL from the expression when the predicate is missing; the second problem is solved by using FN in the relational expression; the third problem is solved by incorporating in the Cartesian product the "external" relations. The meaning is specified by the following transformation, which considers two cases: the case in which the simple predicate is missing, and the case in which it is present. The specification uses the notation from Table I.

```
+  <query> ::= <gb_query> | <ngb_query>

+  <ngb_query> ::= <unary_query> | <binary_query> | ( <unary_query> )

+  <unary_query> ::= <simple_query> | <exists_query> | <complex_query> |

*  <simple_query> ::= SELECT <selector> FROM <relation_list>
                      [ WHERE <simple_predicate> ]

*  <gb_query> ::= <unary_query> GROUP BY <gb_attr> [ HAVING <hav_condition>]

*  <exists_query> ::= SELECT <selector> FROM <relation_list>
                      WHERE <exists_predicate>

*  <complex_query> ::= SELECT <selector> FROM <relation_list>
                       WHERE <left_term> <comp_op> <ngb_query>

*  <binary_query> ::= ( <ngb_query_1> <set_op> <ngb_query_2> )

   <gb_attr> ::= <attribute_spec_list>

   <hav_condition> ::= <function_spec> <comp_op> <constant> |
                       <function_spec> <comp_op> <ngb_query>

   <selector> ::= <attribute_spec_list> <function_spec_list>

   <simple_predicate> ::= ( <simple_predicate> <boolean> <simple_predicate> ) |
                          <attribute_spec> <comp_op> <attribute_spec> |
                          <attribute_spec> <comp_op> <constant>

   <exists_predicate> ::= EXISTS <ngb_query>

   <left_term> ::= <attribute_spec> | <constant>

   <function_spec> ::= <function> ( <attribute_spec_list> )

   <attribute_spec> ::= <relation>.<attribute>

   <boolean> ::= AND | OR

   <set_op> ::= UNION | MINUS | INTERSECT

   <comp_op> ::= = | ≠ | < | ≤ | > | ≥

   <function_name> ::= COUNT | SUM | MIN | MAX | AVERAGE
```

Fig. 5. Restricted SQL grammar; asterisk and plus denote the productions to which a meaning transformation is associated.

meaning(Q : Q is a simple query) =

if empty(Q.<simple_predicate>)

then {case 1}

PJ[Q.<selector>]
FN[Q.<selector>.<function_list>; ϕ]
car(Q.<relation_list>)

else {case 2}

PJ[Q.<selector> $\cup$ other(Q)]
FN[Q.<selector>.<function_list>; other(Q)]
SL[Q.<simple_predicate>]
car(Q.<relation_list> $\cup$ extrels(Q.<simple_predicate>))

other(Q) = attr(extrels(Q.<simple_predicate>) - Q.<relation_list>)

In the *first case*, the third problem cannot arise, and the translation is straightforward; the only difficulty is to include the evaluation of FN over the (possible) functions of the function list. Since grouping of tuples are not required, the second element of the specification is empty. For instance,

SELECT $F(R.A)$, $S.B$ FROM R, S

is translated as

PJ[$F(R.A)$, $S.B$] FN[$F(R.A)$; ϕ] (R CP S)

In the *second case* the function extrels extracts from the simple predicate all the relations appearing in it; the "external" relations, if any, are determined by subtracting the relations of the target list from these relations; other(Q) represents the set of attributes of the "external" relations. The attributes of other(Q) are projected and are also used as grouping attributes in FN. The rationale of these two operations involving other(Q) is the following. Let us assume that Q has just one "external" relation S appearing in its predicate. Thus, other(Q) denotes the attributes of S. S is "immediately" entered in the Cartesian product of the relations of Q, and thus the predicate can be evaluated; however, the attributes of S will be useful for translating the upper level subquery which has S in its relation list. Thus, the attributes of other(Q) need to be in the projection associated with Q. Moreover, the evaluation of aggregate functions must be done *regardless* of the attributes of other(Q), because it applies just to the relations in the relation list of the subquery Q. This effect is obtained by grouping the tuples of the Cartesian product of S with the relations of Q by *different values* of the tuples of S.

For instance

SELECT $F(R.A)$, $R.C$ FROM R WHERE $R.B = S.A$

where S has the schema $S(A, B)$, is translated into

$PJ[F(R.A), R.C, S.A, S.B]$ FN[F$(R.A); S.A, S.B]$
SL$[R.B{=}S.A]$ $(R$ CP $S)$

Clearly, the subquery of this example does not represent a complete SQL query, but only a subquery at a deep level of nesting.

This and the following transformation include the operation FN in the meaning even when there is no aggregate function in the query; in this case, the first term in the specification of FN, which indicates the functions which need to be evaluated, returns the empty set, and the application of FN does not have any effect; for ease of notation, when FN is applied to an empty set we simply do not indicate the operation. For instance,

SELECT $R.A$ FROM R

is translated into

PJ$[R.A]$ R

B. Group-By Query

Recall that a GROUP BY clause can be only at the top level of a query; however, nestings can be present both in the WHERE clause and in the HAVING clause. In the meaning of a group_by query, four cases need to be distinguished, as follows.

1) The HAVING clause is missing and there is no aggregate function in the target list.

2) the HAVING clause is missing and there is an aggregate function in the target list.

3) The HAVING clause is of the type: <function_spec> <comp_op> <constant> (without nesting).

4) The HAVING clause is of the type: <function_spec> <comp_op> <ngb_query> (with nesting).

The meaning is written by giving one relational expression for each case.

```
meaning(Q : Q is a group_by query) =
  if empty(Q.<hav_condition>)
  then if empty(Q.<selector>.<function_spec_list>)
     then {case 1} meaning(Q.<unary_query>)
     else {case 2}
        PJ[Q.<unary_query>.<selector>]
        FN[Q.<unary_query>.<selector>.<function_spec_list>; Q.<gb_attr>]
        meaning(Q.<unary_query>*)

  else {not empty(Q.<hav_condition>)}
     if Q.<hav_condition>.3 = <constant>
     then {case 3}
        PJ[ Q.<unary_query>.<selector>]
        FN[Q.<unary_query>.<selector>.<function_spec_list> -
            Q.<hav_condition>.<function_spec>; Q.<gb_attr>]
        SL[Q.<hav_condition>]
        FN[Q.<hav_condition>.<function_spec>; Q.<gb_attr>]
        meaning(Q.<unary_query>*)

     else {case 4}
        PJ[ Q.<unary_query>.<selector>]
        FN[Q.<unary_query>.<selector>.<function_spec_list> -
            Q.<hav_condition>.<function_spec>; Q.<gb_attr>]
        ( FN[Q.<hav_condition>.<function_spec>; Q.<gb_attr>]
          meaning(Q.<unary_query>*) )
        SJ[ Q.<hav_condition>.<function_spec>
            Q.<hav_condition>.<comp_op>
            Q.<hav_condition>.<ngb_query>.<selector>]
        meaning(Q.<hav_condition>.<ngb_query>)
```

The *first case* is quite simple; since relational algebra cannot control the placement of tuples within the result, the GROUP BY clause has no effect, and the meaning of the subquery is just the meaning of $Q.$<unary_query>.

The *second case* requires the evaluation of functions in the target list. The notation $Q.$<unary_query>* indicates that $Q.$<unary_query>.<selector> must be rewritten, incorporating all the attributes of the $Q.$<unary_query>.<relation_list>. This is required in order to correctly evaluate the functions. For instance,

SELECT $F(R.A)$ FROM R WHERE $R.C{=}7$ GROUP BY $R.B$

where R has the schema $R(A, B, C)$, is translated into

PJ[$F(R.A)$] FN[$F(R.A); B$]
meaning(SELECT $R.A, R.B, R.C$ FROM R WHERE $R.C$=7)

In the *third case*, the HAVING clause requires the following:

1) to evaluate the function on the expression returned by meaning($Q.$<unary_query>$*$);

2) to evaluate: <function_spec> <comp_op> <constant> as a simple selection;

3) to evaluate other (possible) functions of the target list.

Notice that the HAVING clause corresponds to a selection which is applied *after* having evaluated the meaning of $Q.$<unary_query>$*$; this corresponds to evaluating the WHERE clause earlier than the HAVING clause. For instance,

SELECT $F1(R.A)$ FROM R WHERE $R.C$=7 GROUP BY $R.B$ HAVING $F2(R.C) > 2$

is translated into

PJ[$F1(R.A)$] FN[$F1(R.A), B$] SL[$F2(R.C) > 2$] FN[$F2(R.C), B$]
meaning(SELECT $R.A, R.B, R.C$ FROM R WHERE $R.C$=7)

The *fourth case* presents two types of nestings, in $Q.$<unary_query> and in $Q.$<ngb_query>; the corresponding meanings are linked by a semijoin operation which is built on $Q.$<hav_condition>. Prior to the semijoin, it is required to evaluate the aggregate function of $Q.$<hav_condition>; after the semijoin, it is instead required to evaluate other possible aggregate functions in the target list. A parenthesis indicates the appropriate order of evaluation. For instance,

SELECT $F1(R.A)$ FROM R WHERE $R.C$=7 GROUP BY $R.B$ HAVING $F2(R.C)$>
SELECT $S.C$ FROM S

is translated into

PJ[$F1(R.A)$] FN[$F1(R.A), B$] (FN[$F2(R.C), B$]
meaning(SELECT $R.A, R.B, R.C$ FROM R WHERE $R.C$=7))
SJ[$F2(R.C) > C$] meaning(SELECT $S.C$ FROM S)

C. Exists Query

The meaning of an exists subquery Q is synthetic but quite difficult:

meaning(Q : Q is an exists query) =

PJ[$Q.$<selector> $\cup$ other(q)]
FN[$Q.$<selector>.<function_spec_list>; other(q)]
car(($Q.$<relation_list> - connect(Q)) $\cup$ meaning($Q.$<ngb_query>))

connect(Q) = r(meaning($Q.$<ngb_query>)) -
($Q.$<ngb_query>.<relation_list>)

other(Q) = attr(connect(Q) - $Q.$<relation_list>)

Two cases must be distinguished:

1) the case in which there is no connection between $Q.$<ngb_query> and the rest of the query; in this case, connect(Q) is empty;

2) the (more natural) case in which Q and $Q.$<ngb_query> are related.

In the *first case*, $Q.$<ngb_query> acts as a predicate which can return true or false regardless of Q. Thus, if the meaning of $Q.$<ngb_query> returns the empty relation, the meaning of Q is also the empty relation. This effect is obtained by the Cartesian product operator, which is applied to the meaning of $Q.$<ngb_query>. Thus, if the corresponding expression is the empty relation, the result relation associated with Q will also be empty. For instance,

SELECT $R.A$ FROM R WHERE EXISTS
SELECT $S.A$ FROM S WHERE $S.B > 7$

is translated into

PJ[$R.A$] (R CP meaning(SELECT $S.A$ FROM S WHERE $S.B > 7$))

In the *second case*, connect(Q) indicates the relations which are not native of the $Q.$<ngb_query>, but appear in the relational expression corresponding to it because of some link to subqueries which are at an higher level than $Q.$<ngb_query>. The difference ($Q.$<relation_list> - connect(Q)) indicates the relations from the relation list of Q which are *not* already in the meaning of $Q.$<ngb_query>. The difference (connect(Q) - $Q.$<relation_list>) indicates instead the relations which are not native of Q but are in its meaning because, in turn, they are linked to some subquery at an higher level than Q; other(Q) indicates the attributes of these relations, which must appear in both the PJ and FN operations (see the motivation of the use of other(Q) in Section IV-A). For instance:

SELECT $R.A$ FROM R WHERE EXISTS
SELECT $S.A$ FROM S WHERE $S.B$=$R.A$

where S has the schema $S(A, B)$, is translated as follows:

PJ[R.A] meaning(Q.<ngb_query>)

with

meaning($Q.<$ngb_query$>$) = PJ[$S.A, R.A, R.B$] SL[$S.B=R.A$] (S CP R)
connect(Q) = R
other(Q) = ϕ.

D. Complex Query

A complex subquery Q has a unary nesting with $Q.<$ngb_query$>$ represented by a comparison: <left_term> <comp_op><ngb_query>. The meaning is

meaning(Q : Q is a complex query) =

PJ[$Q.<$selector$>$ ∪ other(Q)]
FN[$Q.<$selector$>.<$function_spec_list$>$; other(Q)]
SL[$Q.<$left_term$>$
$Q.<$comp_op$>$
$Q.<$ngb_query$>.<$selector$>$]
car(($Q.<$relation_list$>$ - connect(Q)) ∪
meaning($Q.<$ngb_query$>$))

connect(Q) = r(meaning($Q.<$ngb_query$>$)) -
($Q.<$ngb_query$>.<$relation_list$>$)

other(Q) = attr(connect(Q) - $Q.<$relation_list$>$)

The comparison is translated by a selection operation. The same care as in the *exists* query must be used with connect(Q) and other(Q). For instance:

SELECT $S.A$ FROM S WHERE $S.C$=
SELECT $R.C$ FROM R WHERE $R.B$=$S.B$

where S has relation schema $S(B, C)$, is translated as follows:

PJ[$S.A$] SL[$S.C$=$R.C$] meaning(SELECT $R.C$ FROM R WHERE $R.B$=$S.B$)

with

connect(Q) = S
other(Q) = ϕ

The remaining SQL subquery is a *simple* subquery; by applying the meaning definition to it, the complete SQL query is translated into

PJ[$S.A$] SL[$S.C$=$R.C$] PJ[$R.C, S.A, S.B, S.C$] SL[$R.B$=$S.B$] (R CP S)

E. Binary Query

A binary subquery Q translates the SQL keywords INTERSECT, UNION, and DIFFERENCE between two non-group-by subqueries into the binary relational operations IN, UN, and DF between their meanings. The meaning is

meaning(Q : Q is a binary query) =

meaning($Q.<$ngb_query_1$>$) tr($Q.<$set_op$>$) meaning($Q.<$ngb_query_2$>$)

with tr(INTERSECT) = IN[other1(Q); other2(Q)]
tr(UNION = UN[other1(Q); other2(Q)]
tr(MINUS) = DF[other1(Q); other2(Q)]

other1(Q) = a(meaning($Q.<$ngb_query_1$>$)) -
attr($Q.<$ngb_query_1$>.<$relation_list$>$)

other2(Q) = a(meaning($Q.<$ngb_query_2$>$)) -
attr($Q.<$ngb_query_2$>.<$relation_list$>$)

The difficulty of the translation is that the operations need to be performed on the target lists of the subqueries, while the translated subqueries also contain, in some cases, attributes which are not "native" of the subqueries and are useful in higher level subqueries. This problem is solved by making use of extended binary operations, whose specification includes the "non-native" attributes other1(Q) and other2(Q); the definition of the extended binary operations in terms of traditional algebraic operations was given in Section II. Let us exemplify the translation of binary subqueries and the use of extended binary operations. Consider the SQL query:

SELECT $R.A$ FROM R WHERE EXISTS
(SELECT $S.B$ FROM S
INTERSECT
SELECT $T.B$ FROM T WHERE $T.C$=$R.C$)

The binary subquery corresponds to the SQL statements enclosed by the parenthesis. The meaning of the two subqueries $Q.<$ngb_query_1$>$ and $Q.<$ngb_query_2$>$ is

meaning($Q.<$ngb_query_1$>$ = PJ[$S.B$] S
meaning($Q.<$ngb_query_2$>$ = PJ[$T.B, R.A, R.C$]
SL[$T.C$=$R.C$] (T CP R)

where R has the attribute schema $R(A, C)$. Notice that the second subquery contains R as external relation; R is required by the top-level, exist subquery. Clearly, the intersection of the two meanings cannot be evaluated directly, since the results of the two expressions differ in degree and type of their attributes. In fact, the intersection must be evaluated for each possible tuple of R; thus, the correct meaning is obtained by performing the Cartesian product of the meaning of $Q.<$ngb_query_1$>$ with R, and then performing the traditional intersection. The extended notation of the intersection is used in order to abbreviate the expression. The meaning of Q then becomes

meaning($Q.<$ngb_query_1$>$) IN
[ϕ; $R.A, R.C$] meaning($Q.<$ngb_query_2$>$)

(where IN indicates the intersection operation), corresponding

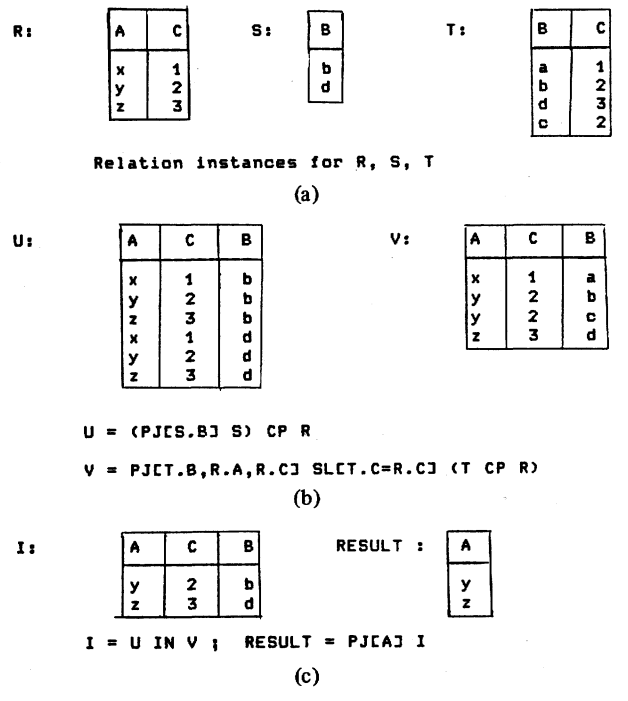


Fig. 6. Progressive construction of the meaning of the example-query of Section IV-E.

to

$$(\text{meaning}(Q.<\text{ngb\_query\_1}>)\ \text{CP}\ R)\ \text{IN}$$
$$(\text{meaning}(Q.<\text{ngb\_query\_2}>))$$

and therefore

$$(\text{PJ}[S.B]\ S\ \text{CP}\ R)\ \text{IN}$$
$$(\text{PJ}[T.B, R.A, R.C]\ \text{SL}[T.C{=}R.C]\ (T\ \text{CP}\ R)))$$

Finally, the evaluation of the meaning of the top-level exists query produces the complete meaning of the query:

$$\text{PJ}[R.A]((\text{PJ}[S.B]\ S\ \text{CP}\ R)\ \text{IN}$$
$$(\text{PJ}[T.B, R.A, R.C]\ \text{SL}[T.C{=}R.C]\ (T\ \text{CP}\ R)))$$

Fig. 6 shows the progressive construction of this meaning on instances of relations R, S, and T.

F. Postprocessing

The postprocessing is applied to the relational expression produced by the evaluation of the meaning. The transformations are purely syntactic; by effect of the postprocessing:

1) All the selections on Cartesian products of relations are either transformed into joins, or directly applied to relations; for instance,

$$\text{SL}[R.A{=}S.A]\ \text{SL}[S.A>7]\ (R\ \text{CP}\ S)$$

becomes

$$R\ \text{JN}[R.A{=}S.B]\ \text{SL}[S.A>7]\ S.$$

2) All the projections which have no effect are eliminated. Thus, a projection over the complete attribute schema of one relation is eliminated; in a sequence of projections which are not interleaved with other operations, just the first projection is maintained. For instance,

$$\text{PJ}[R.A]\ \text{PJ}[R.A, R.B]\ \text{PJ}[R.A, R.B, R.C]\ \text{SL}[R.D>8]\ R$$

becomes

$$\text{PJ}[R.A]\ \text{SL}[R.D>8]\ R.$$

3) Expressions associated with temporary relations are composed into an unique expression. Recall that temporary relations were introduced by the preprocessing transformations associated with some SQL subqueries in order to deal with aggregate functions; they are recognized by the special name TEMP (possibly with apostrophes). When a temporary relation is used, an SQL query Q is transformed into two syntactically independent SQL queries $Q1$ and $Q2$, such that $Q1$ produces TEMP as a result, and $Q2$ uses TEMP in its target list. The meaning transformation is applied to both queries independently. In the postprocessing, the relational expression giving the meaning of $Q1$ is substituted for the name TEMP in the relational expression of $Q2$, thus putting together the two meanings into a single expression. For instance,

SELECT $F(R.A)$ FROM TEMP
TEMP = SELECT $R.A, R.C$ FROM R WHERE $R.B{=}5$

has the meaning

$$\text{PJ}[F(R.A)]\ \text{FN}[F(R.A); \phi]\ \text{TEMP}$$

with

$$\text{meaning(TEMP)} = \text{PJ}[R.A, R.C]\ \text{SL}[R.B{=}5]\ R$$

and, after the postprocessing, becomes

$$\text{PJ}[F(R.A)]\ \text{FN}[F(R.A); \phi]\ \text{PJ}[R.A, R.C]\ \text{SL}[R.B{=}5]\ R.$$

Postprocessing can be regarded the last step of query transformation as well as the first step of query optimization.

V. Application of the Translator to the Proof of Equivalence of SQL Queries

In Section I-C we have already discussed the applications of the translator presented in this paper; the most important ap-

plication is the possibility of applying query optimization algorithms to the output of the translator. We do not further exemplify this application; [4]-[6] and [16]-[18] indicate query processing techniques which can be directly applied to an algebraic expression. In this section, we show instead the application of the translator to the proof of equivalence of syntactically different SQL queries.

A. A Simple Proof of Equivalence

Consider the relations

TOY-SUPPLIER(SNO, NAME)
SUPPLY(SNO, PNO, QTY)
PART(PNO, SIZE)

and the three queries $Q1, Q2, Q3$:

```
Q1 : SELECT PART.PNO FROM PART, SUPPLY, TOY-SUPPLIER
     WHERE (PART.PNO=SUPPLY.PNO AND SUPPLY.SNO=TOY-SUPPLIER.SNO)

Q2 : SELECT PART.PNO FROM PART WHERE EXISTS
       SELECT * FROM TOY-SUPPLIER WHERE EXISTS
         SELECT * FROM SUPPLY WHERE (PART.PNO=SUPPLY.PNO AND
         SUPPLY.SNO=TOY-SUPPLIER.SNO)

Q3 : SELECT PART.PNO FROM PART WHERE EXISTS
       SELECT * FROM TOY-SUPPLIER WHERE TOY-SUPPLIER.SNO =
         SELECT SUPPLY.SNO FROM SUPPLY WHERE PART.PNO=SUPPLY.PNO
```

These queries are syntactically different, but have the same semantics; they retrieve the part number of parts such that a supply order has been issued to some toy supplier. The queries are directly accepted by the RG, and their meaning is

```
meaning(Q1) = PJ[PART.PNO]
              SL[PART.PNO=SUPPLY.PNO AND SUPPLY.SNO=TOY-SUPPLIER.SNO]
              (PART CP SUPPLIER CP SUPPLY)

meaning(Q2) = PJ[PART.PNO]
              PJ[PART.PNO,PART.SIZE,TOY-SUPPLIER.SNO,TOY-SUPPLIER.PNO,
                 TOY-SUPPLIER.QTY]
              PJ[PART.PNO,PART.SIZE,TOY-SUPPLIER.SNO,TOY-SUPPLIER.PNO,
                 TOY-SUPPLIER.QTY, SUPPLY.SNO]
              SL[PART.PNO=SUPPLY.PNO AND SUPPLY.SNO=TOY-SUPPLIER.SNO]
              (PART CP SUPPLIER CP SUPPLY)

meaning(Q3) = PJ[PART.PNO]
              PJ[PART.PNO,PART.SIZE,TOY-SUPPLIER.SNO,TOY-SUPPLIER.PNO,
                 TOY-SUPPLIER.QTY]
              PJ[PART.PNO,PART.SIZE,TOY-SUPPLIER.SNO,TOY-SUPPLIER.PNO,
                 TOY-SUPPLIER.QTY, SUPPLY.SNO]
              SL[PART.PNO=SUPPLY.PNO]
              ( TOY-SUPPLIER CP
                SL[SUPPLY.SNO=TOY-SUPPLIER.SNO]
                (PART CP SUPPLY) )
```

The postprocessing eliminates some of the projections and transforms the selections over Cartesian products into joins; after the postprocessing, the three queries are mapped to the same algebraic expression:

```
PJ[PART.PNO]
( PART JN[PART.PNO=SUPPLY.PNO] SUPPLY
  JN [SUPPLY.SNO=TOY-SUPPLIER.SNO] TOY-SUPPLIER )
```

This result proves the equivalence of $Q1, Q2$, and $Q3$.

B. A More Complex Proof of Equivalence

Consider the relations:

EMP(ENO,DNO,SAL)
DEPT(DNO,LOC)

and the following three queries $Q1, Q2$, and $Q3$:

```
Q1 : SELECT DEPT.DNO
     FROM DEPT, EMP
     WHERE DEPT.DNO=EMP.DNO
     GROUP BY DEPT.DNO
     HAVING AVG(SAL) > 3000

Q2 : SELECT DEPT.DNO
     FROM DEPT, EMP
     WHERE DEPT.DNO=EMP.DNO
     GROUP BY EMP.DNO
     HAVING AVG(SAL) > 3000

Q3 : SELECT DEPT.DNO
     FROM DEPT
```

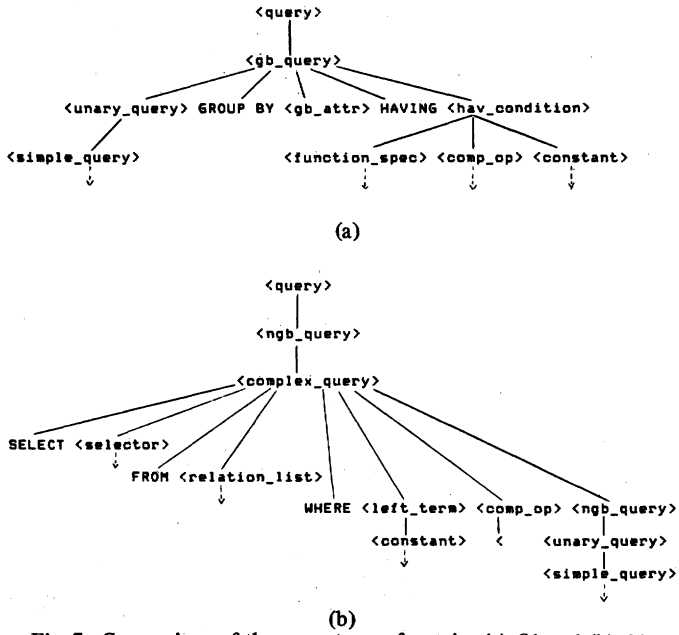


Fig. 7. Comparison of the parse trees of queries (a) $Q1$ and (b) $Q3$.

```
WHERE 3000 < SELECT AVG(SAL)
             FROM EMP
             WHERE EMP.DNO=DEPT.DNO
```

Note that $Q2$ is a query that could not be accepted by some SQL compilers; for instance, in [12], "it is not possible (nor meaningful) to include column names not referenced in the GROUP BY clause in the select list." In this case, however, $Q2$ is a meaningful query.

$Q1$, $Q2$, and $Q3$ are accepted by the RG. Also in this case, the three queries are syntactically different: $Q1$ and $Q2$ differ for the use of a different attribute in the GROUP-BY clause; $Q3$ is very different from $Q1$ and $Q2$ (compare the parse tree of $Q1$ and $Q3$ in Fig. 7). We show that $Q1$, $Q2$, and $Q3$ are equivalent. By applying the meaning and postprocessing to these queries, we obtain the expressions

$E1$: PJ[DEPT.DNO] SL[AVG(SAL) > 3000] FN[AVG(SAL); DEPT.DNO]
EMP JN[DEPT.DNO=EMP.DNO] DEPT

$E2$: PJ[DEPT.DNO] SL[AVG(SAL) > 3000] FN[AVG(SAL); EMP.DNO]
EMP JN[DEPT.DNO=EMP.DNO] DEPT

$E3$: PJ[DEPT.DNO] SL[AVG(SAL) > 3000] FN[AVG(SAL); DEPT.DNO,DEPT.LOC]
EMP JN[DEPT.DNO=EMP.DNO] DEPT

The three expressions differ only in the second element of the specification of the FN operation. Their equivalence descends from the following proposition, which is not formally proved.

Proposition: If the two functional dependencies [2] $R.A \rightarrow R.B$ and $R.B \rightarrow R.A$ hold, then

$$\text{FN}[F(C), R.A]\ R = \text{FN}[F(C), R.B]\ R$$

Intuitively, if the two functional dependencies hold, then the grouping by different values of $R.A$ and $R.B$ are identical; hence the proposition.

This proposition can be applied to prove that $Q1$ and $Q2$ are equivalent by substituting EMP.DNO for A and DEPT.DNO for B: since these two attributes appear in the equi-join specification, for the result of the join it holds that

$$\text{EMP.DNO} \rightarrow \text{DEPT.DNO and DEPT.DNO} \rightarrow \text{EMP.DNO}$$

The proposition can also be applied to prove that $Q1$ and $Q3$ are equivalent, assuming that DEPT.DNO → DEPT.LOC (i.e., that DNO is a key of the relation DEPT). Finally, the equivalence of $Q2$ and $Q3$ is obtained by transitivity.

In this example, the expressions produced by the translator are quite close; however, some further equivalence property between algebraic expressions and some information on the considered database is required in order to complete the proof. The more interesting aspect of this example is that the queries $Q1$ and $Q3$ are *not* equivalent in general (as an SQL programmer might think).

VI. Conclusions

In this paper, we have presented a translator from a relevant subset of SQL into relational algebra. The rationale of our approach is summarized in the following points.

1) A syntax directed translation approach has been used, with translation rules associated with grammar productions. Each translation rule is in correspondence with a well-recognized SQL subquery.

2) There are two distinct types of translation: the preprocessing, from an SQL query to an SQL query, and the meaning transformation, from an SQL query to an expression of relational algebra. This separation allows limiting the meaning definition, which is quite difficult, to five basic subqueries.

3) The translator allows using an optimizer which operates on algebraic expressions, taking advantage of many algorithms presented in the literature.

4) This approach gives a definition of the semantics of SQL and can be used for the proof of equivalence of different SQL queries.

The proposed approach suggests a methodology in the design of new relational query languages:

1) it constitutes an unambiguous specification of the language;

2) it indicates a method for the fast implementation of the translator, which can be used as the first part of a compiler (or interpreter) of the language.

The definition of our translation approach has been long and difficult; however, much less effort was required in order to develop a prototype of the meaning translator. We have used YACC [29], a compiler compiler available with the UNIX operating system, on a VAX 780; the implementation has required about two weeks, mostly spent in understanding the features of YACC. We plan to integrate the translator with

RDB [30], a relational database system based on algebraic primitives, which currently has a menu-driven user interface.

Appendix

Translation of SQL Queries from Date [3]

In this Appendix, all the queries appearing in Date [3] are translated into expressions of relational algebra by applying the transformations presented in this paper, with the limitations presented in Section I-D. Some queries that do not fit in our grammar because of syntactical details have been subject to minor transformations. The relational schema used in Date's queries is

S: (*S*#,SNAME,STATUS,CITY)
P: (*P*#,PNAME,COLOR,WEIGHT,CITY)
SP: (*S*#,*P*#,QTY)

Queries are denoted by the section of Date's book to which they refer; $T(Q)$ indicates the final relational algebra expression produced by the translator.

---

*Q*1(7.2.0): SELECT *S*#,STATUS FROM *S* WHERE CITY = 'PARIS'

$T(Q1)$ = PJ[*S*.*S*#,*S*.STATUS] SL[*S*.CITY='PARIS'] *S*

---

*Q*2(7.2.1): SELECT *P*# FROM SP

$T(Q2)$ = PJ[SP.*P*#] SP

---

*Q*3(7.2.2): SELECT * FROM *S*

$T(Q3) = S$

---

*Q*4(7.2.3): SELECT *S*# FROM *S*
WHERE (CITY = 'PARIS' AND STATUS > 20)

$T(Q4)$ = PJ[*S*.*S*#] SL[*S*.CITY='PARIS' AND *S*.STATUS>20] *S*

---

*Q*5(7.2.4): SELECT *S*#, STATUS FROM *S* WHERE CITY = 'PARIS'
ORDER BY STATUS DESC

The ORDER clause of *Q*5 is not considered; therefore, $T(Q5) = T(Q1)$.

---

*Q*6(7.2.5): SELECT *P*#, CITY FROM SP,*S* WHERE SP.*S*# = *S*.*S*#

$T(Q6)$ = PJ[SP.*P*#,*S*.CITY] (SP JN[SP.*S*#=*S*.*S*#] *S*)

---

*Q*7(7.2.6): SELECT FIRST.*S*#, SECOND.*S*#
FROM *S* FIRST, *S* SECOND
WHERE (FIRST.CITY = SECOND.CITY
AND FIRST.*S*# < SECOND.*S*#)

After naming transformation, *Q*7 becomes

SELECT *S*.*S*#, *S*'.*S*#
FROM *S* , *S*'
WHERE (*S*.CITY = *S*'.CITY
AND *S*.*S*# < *S*'.*S*#)

$T(Q7)$ = PJ[*S*.*S*#,*S*'.*S*#] (*S* JN[*S*.CITY=*S*'.CITY AND *S*.*S*#<*S*'.*S*#] *S*')

---

*Q*8(7.2.7): SELECT SNAME
FROM *S*, SP
WHERE (*S*.*S*# = SP.*S*# AND SP.*P*# = '*P*2')

$T(Q8)$ = PJ[*S*.SNAME] (*S* JN[*S*.*S*#=SP.*S*#] SL[SP.*P*#='*P*2'] SP)

```
Q9 : SELECT SNAME
     FROM S
     WHERE S# = ANY (SELECT S#
                     FROM SP
                     WHERE P# = 'P2')
```

$T(Q9)$ = PJ[S.SNAME] (S JN[S.S#=SP.S#] (PJ[SP.S#] SL[SP.P#='P2'] SP))

Notice that $Q8$ and $Q9$ are used by Date to exemplify two equivalent SQL queries, and in facts $T(Q8)$ and $T(Q9)$ are equivalent algebraic expressions (in order to show the equivalence, it is sufficient to add to $T(Q8)$ the projection PJ[SP.S#], which can be generated using the idempower of projection [2], [19]). Notice that also $Q13$ and $Q18$ are equivalent to $Q8$ and $Q9$.

```
Q10(7.2.8): SELECT S# FROM S
            WHERE STATUS < ANY (SELECT STATUS
                                FROM S)
```

$T(Q10)$ = PJ[S.S#] (S JN[S.STATUS<S'.STATUS] (PJ[S'.STATUS] S'))

```
Q11(7.2.9): SELECT SNAME FROM S WHERE S# IN
              (SELECT S# FROM SP WHERE P# = 'P2')
```

$T(Q11) = T(Q9)$

```
Q12(7.2.10): SELECT SNAME FROM S WHERE S# IN
               (SELECT S# FROM SP WHERE P# IN
                  (SELECT P# FROM P WHERE COLOR = 'RED'))
```

$T(Q12)$ = PJ[S.SNAME] (S JN[S.S#=SP.S#] (PJ[SP.S#]
(SP JN[SP.P#=P.P#] (PJ[P.P#] SL[P.COLOR='RED'] P))))

```
Q13(7.2.11): SELECT SNAME FROM S WHERE 'P2' IN
               (SELECT P# FROM SP WHERE S# = S.S#)
```

$T(Q13)$ = PJ[S.SNAME] ((SL[SP.P#='P2'] SP) JN[SP.S#=S.S#] S)

```
Q14(7.2.12): SELECT S# FROM SP WHERE P# IN
               (SELECT P# FROM SP WHERE S# = 'S2')
```

$T(Q14)$ = PJ[SP.S#] (SP JN[SP.P#=SP'.P#]
(PJ[SP'.P#] SL[SP'.S#='S2'] SP'))

```
Q15(7.2.15): SELECT P# FROM SP SPX WHERE P# IN
               (SELECT P# FROM SP WHERE S# ≠ SPX.S#)
```

$Q15$ is transformed by a naming transformation into

```
Q15: SELECT SP.P# FROM SP WHERE SP.P# IN
       (SELECT SP'.P# FROM SP' WHERE SP'.S# ≠ SP.S#)
```

$T(Q15)$ = PJ[SP.P#] (SP' JN[SP.P#=SP'.P# AND SP'.S#≠SP.S#] SP)

```
Q16(7.2.14): SELECT SNAME FROM S WHERE 'P2' ≠ ALL
               (SELECT P# FROM SP WHERE S# = S.S#)
```

By the preprocessing, $Q16$ is rewritten as

$Q16$: (SELECT S. SNAME FROM S
MINUS
SELECT S. SNAME FROM S WHERE '$P2$' =
SELECT SP.$P\#$ FROM SP WHERE SP.$S\#$ = S. $S\#$)

$T(Q16)$ = (PJ[S. SNAME] S) DF
(PJ[S. SNAME] ((SL[SP.$P\#$='$P2$'] SP) JN[SP.$S\#$=S.$S\#$] S))

$Q17$(7.2.15): SELECT $S\#$ FROM S WHERE CITY =
(SELECT CITY FROM S WHERE $S\#$ = 'S1')

$T(Q17)$ = PJ[S. $S\#$] (S JN[S. CITY=S'.CITY] SL[S'.$S\#$='S1'] S')

$Q18$(7.2.16): SELECT SNAME FROM S WHERE EXISTS
SELECT * FROM SP WHERE ($S\#$ = S. $S\#$ AND $P\#$ = 'P2')

$T(Q18)$ = PJ[S. SNAME] (SL[SP.$P\#$='$P2$'] SP JN[SP.$S\#$=S. $S\#$] S)

$Q19$(7.2.17): SELECT SNAME FROM S WHERE NOT EXISTS
SELECT * FROM SP WHERE ($S\#$=S. $S\#$ AND $P\#$='$P2$')

$T(Q19)$ = (PJ[S. SNAME] S) DF
(PJ[S. SNAME] (SL[SP.$P\#$='P2'] SP JN[SP.$S\#$=S.$S\#$] S))

Notice that $Q16$ and $Q19$ are also equivalent.

$Q20$(7.2.18): SELECT SNAME FROM S WHERE NOT EXISTS
SELECT * FROM P WHERE NOT EXISTS
SELECT * FROM SP WHERE
($S\#$ = S. $S\#$ AND $P\#$ = P. $P\#$)

$T(Q20)$ = (PJ[S. SNAME] S) OF (PJ[S. SNAME] ((S CP P) DF
(PJ[S. $S\#$,S. SNAME,S. STATUS,S. CITY,P. $P\#$,P.PNAME,
P. COLOR,P. WEIGHT,P.CITY]
(SL[SP.$S\#$=S. $S\#$ AND SP.$P\#$=P. $P\#$] ((S CP P) CP SP)))))

This expression can be transformed by some trivial rules into

$T(Q20)$ = (PJ[S. SNAME] S) DF (PJ[S. SNAME] ((S CP P) DF
((S CP P) SJ[SP.$S\#$=S.$S\#$ AND SP.$P\#$=P. $P\#$] SP))))

$Q21$(7.2.19): SELECT $S\#$ FROM SP SPX WHERE NOT EXISTS
SELECT * FROM TEMP WHERE NOT EXISTS
SELECT * FROM SP WHERE ($S\#$=SPX.$S\#$ AND $P\#$=TEMP.$P\#$)

with TEMP = SELECT $P\#$ FROM SP WHERE $S\#$ = $S2$

$Q21$ is transformed by the naming transformation into

$Q21$: SELECT SP.$S\#$ FROM SP WHERE NOT EXISTS
SELECT TEMP.$P\#$ FROM TEMP WHERE NOT EXISTS
SELECT SP'.$S\#$, SP'.$P\#$, SP'.QTY FROM SP'
WHERE (SP'.$S\#$ = SP.$S\#$ AND SP'.$P\#$ = TEMP.$P\#$)

$T(Q21)$ = (PJ[SP.S#] SP) DF
(PJ[SP.S#] ((TEMP CP SP) DF
(PJ[TEMP.P#, SP.S#, SP.P#, SP.QTY]
SL[SP′.S#=SP.S# AND SP′.P#=TEMP.P#]
((TEMP CP SP′) CP SP))))

T(TEMP) = PJ[SP.P#] SL[SP.S#='$S2$'] SP

The substitution of T(TEMP) for TEMP in $T(Q21)$ must be done with some care.
a) since T(TEMP) introduces a third instance of SP, this instance is named SP″.
b) Consequently, the attributes TEMP.P# in $T(Q1)$ are replaced by SP″.P#.
The complete translation of $Q21$ is as follows:

$T(Q21)$: (PJ[SP.S#] SP) DF
(PJ[SP.S#] (((PJ[SP″.P#] SL[SP″.S#='$S2$'] SP″) CP SP) DF
(PJ[SP″.P#, SP.S#, SP.P#, SP.QTY]
SL[SP′.S#=SP.S# AND SP′.P#=SP″.P#]
(((PJ[SP″.P#] SL[SP″.S#='$S2$'] SP″) CP SP′) CP SP))))

$Q22$(7.2.20): SELECT P# FROM P WHERE WEIGHT > 18
UNION
SELECT P# FROM P WHERE S# = 'S2'

$T(Q22)$ = ((PJ[P.P#] SL[P.weight > 18] P) UN (PJ[P.P#] SL[P.S#='$S2$'] P))

$Q23$(7.2.21): SELECT P#, WEIGHT*454 FROM P

Queries involving arithmetic expressions in their relation list are not considered by EG. Adding this feature does not represent a theoretical problem, since the semantics of arithmetic expressions is rather simple.

$Q24$(7.2.22): SELECT S# FROM S WHERE (STATUS > 25 OR STATUS IS NULL)

By treating the null value as an ordinary value ∂, $Q24$ is rewritten as

$Q24$: SELECT S# FROM S WHERE (STATUS > 25 OR STATUS = ∂)

$T(Q24)$ = PJ[S.S#] SL[S.STATUS > 25 OR S.STATUS=∂] S

$Q25$(7.3.1): SELECT COUNT (*) FROM S

$T(Q25)$ = PJ[COUNT(S.S#, S.SNAME, S.STATUS, S.CITY)]
FN[COUNT(S.S#, S.SNAME, S.STATUS, S.CITY); ϕ] S

$Q26$(7.3.2): SELECT COUNT(UNIQUE S#) FROM SP

Here we postulabe the existence of an aggregate function COUNT_U that counts only different iterms.

$Q26$: SELECT COUNT_U(S#) FROM SP

$T(Q26)$ = PJ[COUNT_U(SP.S#)] FN[COUNT_U(SP.S#); ϕ] SP

$Q27$(7.3.3): SELECT COUNT(*) FROM SP WHERE P# = 'P2'

$T(Q27)$ = PJ[COUNT(SP.S#, SP.P#, SP.QTY)]
FN[COUNT(SP.S#, SP.P#, SP.QTY), ϕ]
SL[SP.P#='P2'] SP

Q28(7.3.4): SELECT SUM(QTY) FROM SP WHERE *P#* = '*P2*'

T(*Q*28) = PJ[SUM(SP.QTY)] FN[SUM(SP.QTY); ϕ] SL[SP.*P#*='*P2*'] SP

Q29(7.3.5): SELECT *S#* FROM *S* WHERE STATUS <
SELECT MAX(STATUS) FROM *S*

T(*Q*29) = PJ[*S.S#*] (*S* JN[*S.* STATUS<MAX(*S'*.STATUS)]
(PJ[MAX(*S'*.STATUS)] FN[MAX(S'.STATUS); ϕ] *S'*))

Q30(7.3.6): SELECT P#, SUM(QTY) FROM SP
GROUP BY P#

T(*Q*30) = PJ[SP.P#, SUM(SP.QTY)] FN[SUM(SP.QTY); SP.P#] SP

Q31(7.3.7): SELECT *P#* FROM SP
GROUP BY *P#*
HAVING COUNT(*) > 1

T(*Q*31) = PJ[SP.*P#*] SL[COUNT(SP.*S#*, SP.*P#*, SP.QTY) > 1]
FN[COUNT(SP.*S#*, SP.*P#*, SP.QTY); SP.*P#*] SP

Q32(7.3.8): SELECT *P#*, MAX(QTY) FROM SP
WHERE QTY > 200
GROUP BY *P#*
HAVING SUM(QTY) > 300

T(*Q*32) : PJ[SP.*P#*, MAX(SP.QTY)] FN[MAX(SP.QTY); SP.*P#*]
SL[SUM(SP.QTY)>300] FN[SUM(SP.QTY; SP.*P#*)
SL[SP.QTY>200] SP

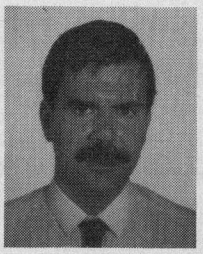
Acknowledgment

The authors thank J. Ullman for proposing this research to them, and for indicating the use of a syntax-directed translation; G. Pelagatti for valuable suggestions in the initial development of our approach; and E. Neuhold, D. Beech, R. Tortini, M. Tunesi, and the referees for providing useful comments.

References

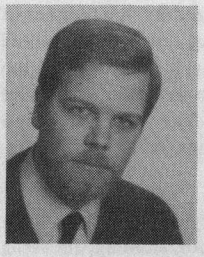
[1] E. F. Codd, "A relational model for large shared data banks," *Commun. ACM*, vol. 13, pp. 377-387, June 1970.
[2] J. D. Ullman, *Principles of Database Systems*, 2nd ed. Rockville, MD: Comput. Sci., Press, 1983.
[3] C. J. Date, *An Introduction to Database Systems*, 3rd ed. Reading, MA: Addison-Wesley, 1981.
[4] E. Wong and K. Youssefi, "Decomposition: A strategy for query processing," *ACM Trans. Database Syst.*, vol. 1, pp. 223-241, Sept. 1976.
[5] A. V. Aho, Y. Sagiv, and J. D. Ullman: "Efficient optimization of a class of relational expressions," *ACM Trans. Database Syst.*, vol. 4, pp. 435-454, Dec. 1979.
[6] J. M. Smith and P. Y. T. Chang, "Optimizing the performance of a relational algebra database interface," *Commun. ACM*, vol. 18, pp. 568-579, Oct. 1975.
[7] S. J. P. Todd, "The Peterlee relational test vehicle—A system overview," *IBM Syst. J.*, vol. 15, pp. 285-308, Oct. 1976.
[8] M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of ingres," *ACM Trans. Database Syst.*, vol. 1, pp. 189-222, Sept. 1976.
[9] D. D. Chamberlin *et al.*, "SEQUEL 2: A unified approach to data definition, manipulation, and control," *IBM J. Res. Develop.*, vol. 20, pp. 560-575, Nov. 1976; erratum in vol. 21, pp. 94-95, Jan. 1977.
[10] M. M. Astrahan *et al.*, "System R: Relational approach to database management," *ACM Trans. Database Syst.*, vol. 1, pp. 97-137, June 1976.
[11] H. M. Weiss, "The ORACLE data base management system," *Mini-Micro Syst.*, pp. 111-114, Aug. 1980.
[12] P. Groth, *IBM DATABASE 2 SQL Usage Guide*, IBM, Santa Teresa, CA, Doc. GG24-1583-00, 1983.
[13] B. Czarnik, S. Schuster, and D. Tsichritzis, "ZETA: A relational database management system," in *Proc. ACM Pacific Regional Conf.*, San Francisco, CA, Apr. 1975, pp. 21-25.
[14] W. Kim, "On optimizing an SQL-like nested query," *ACM Trans. Database Syst.*, vol. 7, pp. 443-469, Sept. 1982.
[15] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1977.
[16] P. Bernstein *et al.*, "Query processing in a system for distributed databases (SDD-1)," *ACM Trans. Database Syst.*, vol. 6, pp. 602-625, Dec. 1981.
[17] P. M. G. Apers, A. R. Hevner, and B. Yao, "Optimization algorithms for distributed queries," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 57-68, Jan. 1983.
[18] S. Ceri and G. Pelagatti, "Allocation of operations in distributed database access," *IEEE Trans. Comput.*, vol. C-31, pp. 119-129, Feb. 1982.
[19] —, *Distributed Databases: Principles and Systems*. New York: McGraw-Hill, 1984.
[20] U. Dayal, N. Goodman, and R. Katz, "An extended algebra with control over duplicate elimination," Comput. Corp. Amer., Boston, MA, Oct. 1981.

[21] A. Klug, "Equivalence of relational algebra and relational calculus query languages having aggregate functions," Dep. Comput. Sci., Univ. Wisconsin, Madison, Tech. Rep. 389, June 1980.
[22] A. Furtado and L. Kerschberg, "An algebra of quotient relations," in *Proc. ACM-SIGMOD Int. Conf.*, Aug. 1977, pp. 1-7.
[23] S. Ceri and G. Pelagatti, "Correctness of query execution strategies in distributed databases," *ACM Trans. Database Syst.*, vol. 8, pp. 577-607, Dec. 1983.
[24] D. J. McLeod, "Sequel and query by example: Translation and compatibility," IBM Res. Rep. RJ1730, Feb. 1976.
[25] S. Y. W. Su, H. Lam, and Der Her Lo, "Transformation of data traversals and operations in application programs to account for semantic changes of databases," *ACM Trans. Database Syst.*, vol. 6, pp. 255-294, June 1982.
[26] N. C. Shu, B. C. Housel, and V. Y. Lum, "Convert: A high level translation definition language for data conversion," *Commun. ACM*, vol. 18, pp. 112-122, Oct. 1975.
[27] M. Zloof, "Query by example," in *Proc. Nat. Comput. Conf.*, vol. 44, May 1975.
[28] D. Bjorner and C. B. Jones, *Formal Specification and Software Development.* Englewood Cliffs, NJ: Prentice-Hall, 1982.
[29] S. C. Johnson, "YACC: Yet another compiler compiler," Bell Lab., Murray Hill, NJ, Comput. Sci. Tech. Rep. 32, 1975.
[30] S. Ceri and S. Crespi, "Relational databases in the design of program construction systems," *ACM SIGSOFT Software Eng. Notes*, vol. 8, pp. 17-29, July 1983.
[31] P. Selinger *et al.*, "Access path selection in a relational database system," in *Proc. ACM SIGMOD Int. Conf.*, Boston, MA, May 1979, pp. 23-34.
[32] A. Rosenthal and D. Reiner, "An architecture for query optimization," in *Proc. ACM SIGMOD Int. Conf.*, Orlando, FL, June 1982, pp. 246-255.
[33] A. V. Aho, Y. Sagiv, and J. Ulmann, "Equivalence of relational expressions," *SIAM J. Comput.*, vol. 8, no. 2, pp. 218-246, 1979.
[34] Y. Sagiv and M. Yannakakis, "Equivalence among relational expressions with the union and difference operators," *ACM J.*, vol. 27, pp. 633-655, Oct. 1980.
[35] A. Klug, "On inequality tableaux," Univ. Wisconsin–Madison, Comput. Sci. Tech. Rep. 403, Nov. 1980.



Stefano Ceri received the doctorate degree in electrical engineering from the Politecnico di Milano, Milano, Italy, in 1978, and the master's degree in computer science from Stanford University, Stanford, CA, in 1982.

At present, he is a Research Associate at the Department of Electrical Engineering, Politecnico di Milano; in 1983 and 1984 he was a Visiting Professor at Stanford University. He performs research in database management, primarily in distributed databases, database design, and the use of relational databases for the development of software systems. His principal activity for the past five years has been in distributed databases, where he has studied the correctness, optimization, and synchronization of distributed transactions, and the design of the distribution of database schemata. He is the coauthor of the book, *Distributed Databases: Principles and Systems* (New York: McGraw-Hill, 1984), and is the author of over 30 papers published in journals and at refereed conferences.



Georg Gottlob received the Dipl.-Ing. degree and the doctoral degree in computer science from the Technical University of Vienna, Vienna, Austria.

From 1980 to 1982 he was with the Institute for Applied Informatics at the Technical University of Vienna. Currently he is Research Associate at the Department of Electrical Engineering, Politecnico di Milano, Milano, Italy. His research interests are database systems (including distributed databases) and theory of computer science. He also was enrolled as a consultant in the design and implementation of a distributed file system for microcomputers.