Formal Semantics of SQL Queries

M. NEGRI
Universita' di Brescia
and
G. PELAGATTI and L. SBATTELLA
Politecnico di Milano

The semantics of SQL queries is formally defined by stating a set of rules that determine a syntax-driven translation of an SQL query to a formal model. The target model, called Extended Three Valued Predicate Calculus (E3VPC), is largely based on a set of well-known mathematical concepts. The rules which can be manipulated using traditional, two-valued predicate calculus are also given; in this way, problems like equivalence analysis of SQL queries are completely solved. Finally, the fact that reasoning about the equivalence of SQL queries using two-valued predicate calculus, without taking care of the real SQL semantics can lead to errors is shown, and the reasons for this are analyzed.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages; H.2.4 [**Database Management**]: Systems—*query processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation*

General Terms: Languages

Additional Key Words and Phrases: Query equivalence, query semantics, SQL

1. INTRODUCTION

The SQL query language has become the most important language in the database field. Initially, SQL was developed for the system R relational DBMS prototype; then, it was adopted by several commercial relational systems; and recently, it has become both a de facto standard accepted by most DBMS manufacturers and an official standard by ANSI and ISO.

The main goal of this paper is to give a complete formal definition of the semantics of SQL queries. The semantics are defined in the following way: first, a formal model, called Extended three-valued Predicate Calculus

Authors' addresses: M. Negri, Dipartimento di Automazione Industriale, Facoltá di Ingegneria, Universitá di Brescia, Via Valotti 9, Brescia, Italy; G. Pellagatti and L. Sbattella, Dipartimento di Elettronica, Politecnico di Milano, Piazza Leonardo da Vinci 32, Milan, Italy.

(E3VPC) is defined, and then, a syntax-directed translation of SQL queries into E3VPC is specified.

The idea of translating SQL to a formal language is not new; several papers [4, 5, 16] on semantics and optimization of SQL queries have already adopted this approach. However, all these works refer to a subset of the SQL language with a restricted syntax and, in some cases, also with a restricted semantics (e.g. without three-valued logic). In [5], the existence of a preprocessing step is explicitely postulated which is capable of transforming every SQL query into an equivalent query of the restricted language being considered.

However, these equivalence transformations are not obvious, as SQL is based on three-valued logic with rules for interpreting "unknown" predicates that are not part of classical predicate calculus. The originality of this paper consists in the fact that the complete syntax and semantics of SQL are considered, thereby providing a safe foundation for any transformation step performed in order to reduce the SQL queries to a standard structure before optimization. In particular, this completeness of definition allows us to show that SQL queries exist that would be equivalent in a two-valued logic interpretation (for instance, by interpreting predicates on null values as False, instead of Unknown), but would not be equivalent with the real SQL semantics. Therefore, two-valued logic is not a sufficient foundation for equivalence transformations of SQL queries.

The structure of this paper is the following: Section 2 defines the formal model E3VPC, Section 3 presents the translation rules from SQL to E3VPC, and Section 4 defines the equivalence transformations that can be applied to E3VPC expressions. In particular, Section 4 states a set of rules that allow transformation of an E3VPC expression to a canonical form, which can be manipulated using the classical theorems of two-valued predicate calculus. Using these rules, the features of SQL responsible for making two queries nonequivalent that would be equivalent in two-valued logic are analyzed.

The formal model E3VPC defined in Section 2 is based on the usual two-valued, many sorted predicate calculus integrated with several extensions that are derived from [8, 15, 19], and a few completely new extensions. The main effort in the definition of E3VPC has been in the integration of all features into a unique, consistent formalism suitable for expressing the semantics of SQL. E3VPC has to satisfy the following requirements: first, being able to express the semantics of all SQL queries, and secondly, having a structure that makes the translation simple. The extent to which this latter goal is achieved is shown by the fact that the whole semantics of SQL queries is expressed by only 33 translation rules (see Table II). Therefore, E3VPC can be used as a foundation for theoretical and practical works on SQL.

The semantics defined in Section 3 is consistent with the SQL definition given by ANSI [3]. A proof of this statement is not given in this paper; however an almost-formal proof has been given in [21]. (A completely formal proof of equivalence is impossible with respect to a nonformal definition.) The claim that the definition given here is complete refers to the ANSI SQL; there are SQL dialects that contain features not defined here. In fact, one

contribution of this paper should be in the development of a unique, well-defined SQL language, in view of the fact that SQL is becoming the most important tool for interoperability between different DBMS.

1.1 Relationships to Other Work

This paper is related to several others in different areas.

(a) *Specification and translation of SQL.* The original specification of SQL is contained in [6]. The translation to relational algebra is defined in [5]. Both [5] and [6] refer to an SQL language that is different from the ANSI-SQL, as defined in [3]. A formal specification of SQL, including the problem of three-valued predicates, has not been given.

(b) *Treatment of null-values in databases.* The problem of null-values in databases has been studied in [7, 8, 11, 13, 14, 20, 27]. These works are orthogonal to this paper, as they discuss the correct approach that should be given to the problem of null-values, considering the SQL approach (which is truth-functional) too limited. This paper, instead, assumes the SQL approach as its basis, having the definition of SQL semantics as a target, not the proposal of a new approach to null-values.

(c) *Query optimization.* The works on query optimization (for example, [1, 4, 10, 12, 16, 17, 22, 24, 25, 28]) are loosely related to this paper, as the understanding of the semantics of a query language is a premise for optimization. Moreover, inasmuch as the specification is given by translating SQL to a formal model, the transformations on the obtained formal expressions can be used for optimization.

(d) Equivalence of queries. The problem of equivalence between relational expressions is important and has already been studied ([2, 5, 18, 23]). These works, however, do not consider the three-valuedness of predicates.

This paper is also related to existing formulations of two- and three-valued predicate calculus, as described in the next section.

2. EXTENDED THREE-VALUED TUPLE PREDICATE CALCULUS (E3VPC)

Extended three-valued Tuple Predicate Calculus is based on the usual two-valued, many sorted predicate calculus [15, 16, 26], enriched with several extensions that are necessary in order to use it for the definition of the SQL semantics. These extensions are:

compact forms [15, 16, 21, 26],
aggregation [4, 16, 19],
three-valued logic [8],
interpretation (of unknown) *operator* [4, 8, 20],
and two minor other extensions:
null comparison operator $\overset{\omega}{=}$
external reference operator (↑).

Since E3VPC includes many well-known aspects, the main effort in its definition has been in determining a unique and consistent structure of E3VPC expressions, and in determining a small set of new features; therefore, in Subsection 2.1, the complete structure of an E3VPC expression is defined, but in Subsection 2.2, only the interpretation of the new features is defined in detail.

2.1 The Structure of E3VPC Expressions

An E3VPC expression has the structure

$$\{t(v_1, \ldots, v_n) : \| P(v_1, \ldots, v_n) \|^{\alpha}\}$$

where $v_1, \ldots, v_n$ are tuple variables, $t(v_1, \ldots, v_n)$ is the target list of the expression, $P(v_1, \ldots, v_n)$ is a predicate formula, and $\| \ldots \|^{\alpha}$ is the interpretation operator, where α can assume one of the two values, T (true) and F (false).

Target list. The target list $t(v_1, \ldots, v_n)$ specifies the structure of the result of the expression and is defined as

$$v_1 \; in \; R_1, \ldots, v_n \; in \; R_n$$

where $v_i \; in \; R_i$ is a range formula that bounds the variable v_i to a relation R_i. Relations are assumed to be sets of distinct objects in E3VPC, and therefore tuples are assumed to contain a unique tuple identifier.

Predicate formula. The structure of a predicate formula is defined recursively, using the notions of *terms* and of *comparison operators*, as follows:

(1) *Terms*
- —a constant is a term;
- —for each variable v_i and attribute name A_j, $v_i.A_j$ and $v_i\uparrow.A_j$ are terms ($\uparrow$ is called external reference operator);
- —if S is an E3VPC expression, A denotes an attribute, and f is an aggregate function, then f(A)S is a term;

$$f \in \{\text{COUNT}, \text{AVG}, \text{SUM}, \text{MIN}, \text{MAX}, \text{COUNTD}, \text{AVGD}, \text{SUMD}\}.$$

(2) *Comparison operators*
- —the $=$, $\neq$, $\geq$, ... symbols denote the usual comparison operators, with the classical meaning (recall that, if a term of a comparison is a null value, the comparison has an undefined result);
- —the $\overset{\omega}{=}$ symbol denotes the null-comparison operator, which differs from the $=$ operator because it yields true when both compared terms are null values.

(3) *Atomic predicates*
- —if t_1, t_2 are terms and θ is a comparison operator, then $t_1 \theta t_2$ is an atomic predicate;
- —T (true) is an atomic predicate;

—F (false) is an atomic predicate;

—U (unknown) is an atomic predicate.

(4) If P is an atomic predicate, then $\| P \|^{\alpha}$ is also an atomic predicate, where $\| \ldots \|^{\alpha}$ is the interpretation operator already mentioned.

(5) An atomic predicate is a predicate formula.

(6) Let P and Q be predicate formulas, then $\neg P$, $P \vee Q$, and $P \wedge Q$ are predicate formulas.

(7) Let $\{t(v_1, \ldots, v_n): \| P(v_1, \ldots, v_n) \|^{\alpha}\}$ be an E3VPC expression, and $Q(v_1, \ldots, v_n)$ a predicate formula, then the quantified predicates

$$\exists \{t(v_1, \ldots, v_n) : \| P(v_1, \ldots, v_n) \|^{\alpha}\} Q(v_1, \ldots, v_n)$$

$$\forall \{t(v_1, \ldots, v_n) : \| P(v_1, \ldots, v_n) \|^{\alpha}\} Q(v_1, \ldots, v_n)$$

are also predicate formulas.

(8) No other formulas are predicate formulas.

2.2 Interpretation of an E3VPC Expression

The result of the evaluation of an E3VPC expression is a set of tuples defined by the free variables (the variables of the target list) satisfying the predicate formula, as in usual many sorted tuple predicate calculus [15, 16].

Inasmuch as predicates and predicate formulas are three-valued in E3VPC, the interpretation of logical connectives used in rule 6 is not the usual one of two-valued logic, but the one given in [8]; as a consequence of the use of three-valued logic, however, the meaning of an expression

$$\{x \, in \, R : P(x)\}$$

(which is not a valid E3VPC expression) is undefined, because it is not defined whether a tuple x′, such that P(x′) = U belongs to it or not. Therefore, the interpretation operator, which transforms a three-valued predicate into a two-valued predicate, is required in the predicate formula of an E3VPC expression.

Interpretation operator. The interpretation of unknowns is defined as follows: let $P(x)$ be a three-valued predicate formula and $Q(x)$ be a two-valued predicate formula, then

Definition 1. $Q(x)$ is a true-interpreted two-valued equivalent of $P(x)$ if

$$P(x) = T \Rightarrow Q(x) = T$$

$$P(x) = F \Rightarrow Q(x) = F$$

$$P(x) = U \Rightarrow Q(x) = T$$

for all x.

Definition 2. $Q(x)$ is a false-interpreted two-valued equivalent of $P(x)$ if

$$P(x) = T \Rightarrow Q(x) = T$$

$$P(x) = F \Rightarrow Q(x) = F$$

$$P(x) = U \Rightarrow Q(x) = F$$

for all x.

The notation $\| P(x) \|^T$ means a predicate that is a true-interpreted two-valued equivalent of P(x), and $\| P(x) \|^F$ means a predicate which is a false-interpreted two-valued equivalent of P(x).

The fundamental rules that allow transformation of the relationships involving interpreted predicates are given in Section 4.

Terms. The definition of terms in E3VPC has the usual meaning, except for the use of the external reference operator.

The *external reference operator* (↑) allows modification of the usual scope rules of a variable, and greatly simplifies the translation of SQL queries. Given an E3VPC expression, $\{v \text{ in } R{:}P(v, v\uparrow)\}$, the ↑ operator indicates that the variable $v\uparrow$ does not have to be considered within the scope of the range formula v *in* R of the expression; therefore, $v\uparrow$ belongs to the scope of the closest outer expression containing a range formula v *in* . . . . For example, in the expression $\{v \text{ in } S{:}\forall\{v \text{ in } R{:}\| v.A_1 = v\uparrow.B_1 \|^{\alpha}\} \ldots\}$, in the predicate $v.A_1 = v\uparrow.B_1$, the first v refers to the range formula v *in* R, and the second v refers to the range formula v *in* S.

The interpretation of an aggregate function f(A), S is the same as in [19], where the E3VPC expression S corresponds to an *alpha* of [19].

An aggregate function applied to a given set of tuples produces a single value as its result (for example, MAX(A) evaluates the maximum among the values of the attribute A when applied to S). These functions return a null value when applied to an empty set, except for COUNT and COUNTD that return 0; they discard all tuples containing a null value in the selected attribute A before their evaluation, except for COUNT which does not discard any tuple in its evaluation; finally, COUNTD, SUMD, and AVGD also discard identical values of the selected attribute before their evaluation.

Quantified predicates and compact forms. The expression of quantified predicates in E3VPC, as stated by rule 7 of Subsection 2.1, is based on compact forms. Compact forms are well known in traditional two-valued logic, and are used to increase readability of expressions; as will be shown in Section 3, compact forms are also very useful for representing the SQL query structure.

The notation in compact form, $\forall\{x \text{ in } R{:}\| W(x) \|^{\alpha}\} P(x)$, means that "for all tuples x that belong to the set $\{x \text{ in } R{:}\| W(x) \|^{\alpha}\}$, $P(x)$"; and the notation, $\exists\{x \text{ in } R{:}\| W(x) \|^{\alpha}\} P(x)$, means that "there is at least one tuple in the set $\{x \text{ in } R{:}\| W(x) \|^{\alpha}\}$ such that $P(x)$". If the set $\{x \text{ in } R{:}\| W(x) \|^{\alpha}\}$ is empty, then the value of the universally quantified formula is true, and the value of the existentially quantified formula is false.

Notice that the interpretation operator is necessary for extending the definition of compact forms to three-valued logic. The meaning of a quantified predicate in E3VPC is defined as follows:

$$\forall\{x \text{ in } R : \| W(x) \|^{\alpha}\} P(x)$$

= T if the set is empty or for all elements of the set $P(x)$ = T;

= F if the set is not empty and there is at least an element x' with $P(x') = \mathrm{F}$;

= U otherwise (the set is not empty, there is no element x' with $P(x') = \mathrm{F}$ and there is at least an element x'' with $P(x'') \neq \mathrm{T}$; of course, $P(x'') = \mathrm{U}$).

$$\exists\{x \text{ in } R : \| W(x) \|^{\alpha}\} P(x)$$

= F if the set is empty or for all elements of the set $P(x) = \mathrm{F}$;

= T if the set is not empty and there is at least an element x' with $P(x') = \mathrm{T}$;

= U otherwise (the set is not empty, there is no element x', with $P(x') = \mathrm{T}$ and there is at least an element x'' with $P(x'') \neq \mathrm{F}$; of course, $P(x'') = \mathrm{U}$).

3. SEMANTICS OF SQL QUERIES

In this section, the semantics of SQL queries (as stated in [3]) is defined by defining their syntax-directed translation to E3VPC expressions.

SQL syntax. The syntax of SQL queries used in this paper is shown in Table I; the boldface elements are SQL keywords, lowercase strings are terminals (except keywords), and uppercase strings are nonterminals.

Table I is split into four parts: the *query-block*, which is the basic SQL structure, the *where search condition* and the *having search condition*, which define the selection predicates, and finally the *subquery block*, which defines the nested subquery used in the complex predicates.

The syntax given in Table I covers the whole syntax of SQL queries as defined in [3]; however, it has been partially reformulated in order to simplify the description, and to make the translation process easier. The modifications are the following:

—the notation "Boolean expression of" and "list of" have been used as a short form for a set of very obvious syntactic rules;

—arithmetic expressions of attributes have been eliminated, because they are absolutely irrelevant to the points considered in this paper, and their introduction is obvious;

—correlation names, which are optional in [3], are mandatory;

—several rules have been split and additional nonterminals introduced, in order to direct the translation process more easily.

Translation rules. The translation of an SQL query to an E3VPC expression is a syntax-directed transformation defined by a set of translation rules associated with the syntax rules.

Each translation rule is associated with a corresponding syntax rule; it defines that the translation $TR\langle X\rangle$ of the nonterminal $\langle X\rangle$ on the LHS of the syntax rule is the concatenation of the translations $TR\langle Y_1\rangle \ldots TR\langle Y_n\rangle$ with some additional E3VPC symbols interleaved, where $\langle Y_i\rangle$ is an element (terminal or nonterminal) of the RHS of the above syntax rule, or it is a nonterminal belonging to the same query (subquery) block as the

Table I. Syntax of SQL Queries

QUERY BLOCK

1 <QUERY> = **SELECT** [**ALL** | **DISTINCT**] <SELECT LIST> <FRCLAUSE> [<WHCLAUSE>] [<GBCLAUSE>] [<HCLAUSE>]
2 <SELECT LIST> = list of <SELECT ELEMENT>
3 <SELECT ELEMENT> = <COL OR VAL> | <FUNC SPEC>
4 <COL OR VAL> = <correlation name> <column name> | <literal>
5 <FUNC SPEC> = <COUNT FUNCTION SPEC> | <AGGR FUNCTION SPEC>
6 <COUNT FUNCTION SPEC> = <DISTINCT COUNT FUNCTION> | **COUNT**(*)
7 <AGGR FUNCTION SPEC> = <DISTINCT AGGR FUNCTION> | <ALL AGGR FUNCTION>
8 <DISTINCT COUNT FUNCTION> = **COUNT** (**DISTINCT** <correlation name> <column name>)
9 <DISTINCT AGGR FUNCTION> = <AGGR FUNCTION NAME> (**DISTINCT** <correlation name> <column name>)
10 <ALL AGGR FUNCTION> = <AGGR FUNCTION NAME> ([**ALL**] <correlation name> <column name>)
11 <AGGR FUNCTION NAME> = **AVG** | **MAX** | **MIN** | **SUM**
12 <FRCLAUSE> =**FROM**<TABLE REFERENCE> [,<TABLE REFERENCE>]
13 <TABLE REFERENCE> = <table name><correlation name>
14 <WHCLAUSE> = **WHERE** <WHERE SEARCH COND>
15 <GBCLAUSE> =**GROUP BY**<correlation name> <column name> [<correlation name> <column name>]
16 <HCLAUSE> =**HAVING**<HAVING SEARCH COND>

WHERE SEARCH CONDITION

17 <WHERE SEARCH COND> = "Boolean Expression of" <WPRED>
18 <WPRED> = <SIMPLE PRED> | <COMPLEX PRED>
19 <SIMPLE PRED> = <COL OR VAL> <comp op> <COL OR VAL>
20 <COMPLEX PRED> = <SOME QUANTIFIED PRED> | <SOME QUANTIFIED AFPRED> | <ALL QUANTIFIED PRED> | | <ALL QUANTIFIED AFPRED> | <COMPLEX IN PRED> | <COMPLEX IN AFPRED> | <COMPLEX NOT IN PRED> | | <COMPLEX NOT IN AFPRED> | <EXISTS PRED> | <COMPLEX COMP PRED> | <COMPLEX COMP AFPRED>
21 <SOME QUANTIFIED PRED> = <COL OR VAL> <comp op> **SOME** <SUBQ>
22 <SOME QUANTIFIED AFPRED> = <COL OR VAL> <comp op> **SOME** <AFSUBQ>
23 <ALL QUANTIFIED PRED> = <COL OR VAL> <comp op> **ALL** <SUBQ>
24 <ALL QUANTIFIED AFPRED>..= <COL OR VAL> <comp op> **ALL** <AFSUBQ>
25 <COMPLEX IN PRED>: = <COL OR VAL> **IN** <SUBQ>
26 <COMPLEX IN AFPRED> = <COL OR VAL> **IN** <AFSUBQ>
27 <COMPLEX NOT IN PRED> . = <COL OR VAL> **NOT IN** <SUBQ>
28 <COMPLEX NOT IN AFPRED> . = <COL OR VAL> **NOT IN** <AFSUBQ>
29 <EXISTS PRED> := **EXISTS** <SUBQ>
30 <COMPLEX COMP PRED> = <COL OR VAL> <comp op> <SUBQ>
31 <COMPLEX COMP AFPRED> = <COL OR VAL> <comp op> <AFSUBQ>

HAVING SEARCH CONDITION

32 <HAVING SEARCH COND> = "Boolean expression of" <HPRED>"
33 <HPRED>' = <HSIMPLE PRED> | <HCOMPLEX PRED> | <HACOL PRED> | <HAFUN PRED> | <HACOMPLEX PRED>
34 <HSIMPLE PRED>: = <SIMPLE PRED>
35 <HCOMPLEX PRED>..= <COMPLEX PRED>
36 <HACOL PRED>''= <FUNC SPEC> <comp op> <COL OR VAL>
37 <HAFUN PRED>. = <FUNC SPEC> <comp op> <FUNC SPEC>
38 <HACOMPLEX PRED> = <AFSOME QUANTIFIED PRED>|<AFSOME QUANTIFIED AFPRED>|<AFALL QUANTIFIED PRED>| | <AFALL QUANTIFIED AFPRED> | <AFCOMPLEX IN PRED> | <AFCOMPLEX IN AFPRED> | <AFCOMPLEX NOT IN PRED> | | <AFCOMPLEX NOT IN AFPRED> | <AFCOMPLEX COMP PRED> | <AFCOMPLEX COMP AFPRED>
39 <AFSOME QUANTIFIED PRED>:.= <FUNC SPEC> <comp op>**SOME**<SUBQ>
40 <AFSOME QUANTIFIED AFPRED>: = <FUNC SPEC> <comp op>**SOME**<AFSUBQ>
41 <AFALL QUANTIFIED PRED>''= <FUNC SPEC> <comp op>**ALL**<SUBQ>
42 <AFALL QUANTIFIED AFPRED> = <FUNC SPEC> <comp op>**ALL**<AFSUBQ>
43 <AFCOMPLEX IN PRED> = <FUNC SPEC>**IN**<SUBQ>
44 <AFCOMPLEX IN AFPRED>..= <FUNC SPEC>**IN**<AFSUBQ>
45 <AFCOMPLEX NOT IN PRED>':= <FUNC SPEC>**NOT IN**<SUBQ>
46 <AFCOMPLEX NOT IN AFPRED>' = <FUNC SPEC>**NOT IN**<AFSUBQ>
47 <AFCOMPLEX COMP PRED>:.= <FUNC SPEC> <comp op> <SUBQ>
48 <AFCOMPLEX COMP AFPRED> = <FUNC SPEC> <comp op> <AFSUBQ>

SUBQUERY BLOCK

49 <SUBQ> = **SELECT** [**ALL** | **DISTINCT**] <COL OR VAL> <FRCLAUSE> [<WHCLAUSE>] [<GBCLAUSE>] [<HCLAUSE>]
50 <AFSUBQ> .= **SELECT** [**ALL** | **DISTINCT**] <FUNC SPEC> <FRCLAUSE> [<WHCLAUSE>] [<GBCLAUSE>] [<HCLAUSE>]

NOTE: the metasymbols (|), (.), and ([]) indicate choice between elements, elements that may be repeated one or more times, and optional elements

nonterminal $\langle X \rangle$ (in the latter case, $TR\langle Y_i \rangle$ is called *inherited translation*). The translation of optionality in the syntax rules is straightforward; the translation rules can use the optional element in the translation, but, if the optional element is not used in the SQL query, then its translation is empty.

In general, there is only one translation rule for each syntax rule and, therefore, the correspondence is indicated by using the same numbers. However, in some cases (rules 49 and 50), it is possible to have more than one translation rule for the same syntax rule, so that different translations are involved in different contexts. This is indicated using a two-level numbering i.j, where i indicates the syntax rule, and j the specific translation rule; the invocation of a specific translation rule is indicated by writing $TRj\langle \ldots \rangle$, instead of just $TR\langle \ldots \rangle$.

In order to make the complete translation, given in Table II, more compact and readable and to concentrate on those aspects that really characterize the semantics of SQL, a set of obvious simplifications have been used.

(1) A translation rule is not stated explicitly when it consists of the pure concatenation of the translations of the elements of the RHS of the corresponding syntax rule in the same order.

(2) Terminals are not translated and appear without modification in E3VPC. (The SQL terminals ⟨correlation name⟩, ⟨column name⟩, ⟨table name⟩, ⟨literal⟩, and ⟨comp op⟩ assume the roles of variable name, attribute name, relation name, constant, and comparison operator in the generated E3VPC expression.)

(3) The SQL nonterminal ⟨COL OR VAL⟩ is treated like a terminal; therefore, in reading the rules of Table II, whenever ⟨COL OR VAL⟩ is encountered the generation of an E3VPC constant or variable attribute must be assumed.

(4) Similarly, whenever *TR*1 ⟨FUNC SPEC⟩ is encountered, the generation of an E3VPC function name followed, except for COUNT(*), by (variable.attribute) must be assumed, with the correct correspondence between SQL functions and E3VPC functions, as follows:

COUNT(*)	(in ⟨COUNT FUNCTION SPEC⟩)	→ COUNT
SUM	(in ⟨ALL AGGR FUNCTION⟩)	→ SUM
AVG	(in ⟨ALL AGGR FUNCTION⟩)	→ AVG
MIN	(in ⟨ALL AGGR FUNCTION⟩)	→ MIN
MAX	(in ⟨ALL AGGR FUNCTION⟩)	→ MAX
COUNT	(in ⟨DISTINCT COUNT FUNCTION⟩)	→ COUNTD
SUM	(in ⟨DISTINCT AGGR FUNCTION⟩)	→ SUMD
AVG	(in ⟨DISTINCT AGGR FUNCTION⟩)	→ AVGD
MIN	(in ⟨DISTINCT AGGR FUNCTION⟩)	→ MIN
MAX	(in ⟨DISTINCT AGGR FUNCTION⟩)	→ MAX

and whenever *TR*2 ⟨FUNC SPEC⟩ is encountered, the generation of an E3VPC constant must be assumed; the value 0 is produced if the function is COUNT and the null value is produced for all the other functions. (These are the values that are returned by applying the functions to an empty set).

Table II. Translation Rules for SQL Queries

QUERY BLOCK

1 $\{ TR$ <FRCLAUSE> : $\| TR$ <WHCLAUSE> $\wedge TR$ <HCLAUSE> $\|^F \}$

13 <correlation name> in <table name>

15 <correlation name> <column name> $\stackrel{\omega}{=}$ <correlation name> $\uparrow$ <column name>
|$\wedge$ <correlation name> <column name> $\stackrel{\omega}{=}$ <correlation name> $\uparrow$ <column name>|

WHERE SEARCH CONDITION

21 $\exists TR1$ <SUBQ> <COL OR VAL> <comp op> $TR2$ <SUBQ>

22 $\exists TR1$<AFSUBQ> <COL OR VAL><comp op> $TR2$<AFSUBQ> $\vee$ ($\neg\exists TR1$<AFSUBQ> $\wedge$ <COL OR VAL><comp op>$TR3$<AFSUBQ>)

23 $\forall TR1$ <SUBQ> <COL OR VAL> <comp op> $TR2$ <SUBQ>

24 $\forall TR1$<AFSUBQ> < COL OR VAL><comp op> $TR2$<AFSUBQ> $\wedge$ ($\exists TR1$<AFSUBQ> $\vee$ <COL OR VAL> <comp op>$TR3$<AFSUBQ>)

25 $\exists TR1$ <SUBQ> <COL OR VAL> $= TR2$ <SUBQ>

26 $\exists TR1$ <AFSUBQ><COL OR VAL> $= TR2$ <AFSUBQ> $\vee$ ($\neg\exists TR1$ <AFSUBQ> $\wedge$ <COL OR VAL><comp op> $TR3$ <AFSUBQ>)

27 $\forall TR1$ <SUBQ><COL OR VAL>$\neq TR2$ <SUBQ>

28 $\forall TR1$ <AFSUBQ> <COL OR VAL> $\neq TR2$ <AFSUBQ> $\wedge$ ($\exists TR1$ <AFSUBQ> $\vee$ <COL OR VAL> <comp op> $TR3$ <AFSUBQ>)

29 $\exists TR1$ <SUBQ>

30 $\exists TR1$ <SUBQ> <COL OR VAL><comp op> $TR2$ <SUBQ> $\vee$ ($\neg\exists TR1$ <SUBQ> $\wedge$ U)

31 <COL OR VAL> <comp op> $TR2$ <AFSUBQ>

HAVING SEARCH CONDITION

34 $\forall \{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\}$ TR <SIMPLE PRED>

35 $\forall \{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\}$ (TR <COMPLEX PRED>)

36 $TR1$ <FUNC SPEC> $\{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\}$ <comp op> <COL OR VAL>

37 $TR1$ <FUNC SPEC> $\{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\}$ <comp op>
$TR1$ <FUNC SPEC> $\{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\}$

39 $\exists TR1$<SUBQ> $TR1$<FUNC SPEC> $\{ TR$<FRCLAUSE> $\|TR$<WHCLAUSE> $\wedge TR$<GBCLAUSE>$\|^F\}$ <comp op>$TR2$<SUBQ>

40 $\exists TR1$<AFSUBQ> $TR1$<FUNC SPEC>$\{TR$<FRCLAUSE> $\|TR$<WHCLAUSE> $\wedge TR$<GBCLAUSE>$\|^F\}$<comp op> $TR2$<AFSUBQ>
$\vee$ ($\neg\exists TR1$ <AFSUBQ> $\wedge$ <COL OR VAL> <comp op> $TR3$ <AFSUBQ>)

41 $\forall TR1$<SUBQ> $TR1$<FUNC SPEC> $\{ TR$<FRCLAUSE> $\|TR$<WHCLAUSE> $\wedge TR$<GBCLAUSE>$\|^F\}$ <comp op> $TR2$<SUBQ>

42 $\forall TR1$<AFSUBQ> $TR1$<FUNC SPEC>$\{TR$<FRCLAUSE> $\|TR$<WHCLAUSE> $\wedge TR$<GBCLAUSE>$\|^F\}$<comp op>$TR2$<AFSUBQ>
$\wedge$ ($\exists TR1$<AFSUBQ> $\vee$ <COL OR VAL><comp op> $TR3$<AFSUBQ>)

43 $\exists TR1$ <SUBQ> $TR1$ <FUNC SPEC> $\{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\} = TR2$ <SUBQ>

44 $\exists TR1$<AFSUBQ> $TR1$<FUNC SPEC>$\{ TR$<FRCLAUSE > $\|TR$ <WHCLAUSE> $\wedge TR$<GBCLAUSE> $\|^F\} = TR2$<AFSUBQ>
$\vee$ ($\neg\exists TR1$ <AFSUBQ> $\wedge$ <COL OR VAL> <comp op> $TR3$ <AFSUBQ>)

45 $\forall TR1$ <SUBQ> $TR1$ <FUNC SPEC> $\{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\} \neq TR2$ <SUBQ>

46 $\forall TR1$<AFSUBQ> $TR1$<FUNC SPEC>$\{ TR$<FRCLAUSE> $\|TR$ <WHCLAUSE> $\wedge TR$<GBCLAUSE> $\|^F\} \neq TR2$<AFSUBQ>
$\wedge$ ($\exists TR1$ <AFSUBQ> $\vee$ <COL OR VAL> <comp op> $TR3$ <AFSUBQ>)

47 $\exists TR1$<SUBQ>$TR1$<FUNC SPEC>$\{TR$<FRCLAUSE> $\|TR$<WHCLAUSE> $\wedge TR$<GBCLAUSE>$\|^F\}$<comp op> $TR2$<SUBQ>
$\vee$ ($\neg\exists TR1$<SUBQ> $\wedge$ U)

48 $TR1$ <FUNC SPEC> $\{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\}$ <comp op> $TR2$ <AFSUBQ>

SUBQUERY BLOCK

49 1 $\{ TR$ <FRCLAUSE> $\| TR$ <WHCLAUSE> $\wedge TR$ <HCLAUSE> $\|^F\}$

49 2 <COL OR VAL>

50 1 $\{ TR$ <FRCLAUSE> $\| TR$ <WHCLAUSE> $\wedge TR$ <HCLAUSE> $\|^F\}$

50 2 $TR1$ <FUNC SPEC> $\{ TR$ < FRCLAUSE > $\|TR$ < WHCLAUSE> $\wedge TR$ <GBCLAUSE> $\|^F\}$

50 3 $TR2$ <FUNC SPEC>

NOTE: the ellipses (. . .), and the square brackets (| |) are metasymbols with the meaning given in Table I

(5) The translation of the rules having the form "Boolean expression of" $\langle P \rangle$ is "Boolean expression of" $TR\langle P \rangle$ (that is, the same Boolean expression in E3VPC as in SQL with the predicates substituted by their translations). Parentheses are not explicitly indicated in Table II, since the places where they should be generated are obvious.

Finally, the select list of the (external) query block is not translated, since it only states which attribute values are extracted from the tuples that constitute the query result.

The rationale of the translations is now briefly discussed. An extensive discussion of the correspondence between the formal semantics expressed by Table II and the semantics of SQL, as defined in [3], can be found in [21].

(1) *General structure of the translation of the query and subquery blocks.* The E3VPC formula, which constitutes the translation of the basic query (subquery) block, which can be contained in the outer query or in a subquery, is generated by translation rules 1, 49.1, and 50.1. All these translations have the same general structure:

$$\{TR\ \langle\text{FRCLAUSE}\rangle : \| TR\ \langle\text{WHCLAUSE}\rangle \wedge TR\ \langle\text{HCLAUSE}\rangle \|^{F}\}$$

This formula defines the set of tuples that are returned by the query or subquery. The following comments explain the rationale of this formula (in the sequel we call Selection Expression, the conjunction of the translations of the *where* and *having* clauses):

1.1 *The Selection Expression* is false-interpreted because the SQL semantics does not include in the result those tuples that produce an Unknown as a result of the evaluation of the where clause and those groups of tuples that produce an Unknown as a result of the evaluation of the *having* clause. The false-interpretation of the Selection Expression in the above E3VPC formula produces exactly the same result. Of course, this does not invalidate the three-valued logic approach, since three-valuedness is applied in the evaluation of the predicates of the Selection Expression, and interpreting the whole expression as false is not the same as interpreting each predicate as false, which would reduce SQL to two-valued logic.

1.2 *The group-by clause* does not appear in the above formula because it is not required to express the set of tuples constituting the result. The translation of the group-by clause is invoked (inherited translation) in the translation of the having clause and of predicates involving aggregate functions, because grouping affects the evaluation of the having clause and of functions. The translation of the group-by clause is essentially a join predicate, which correlates each tuple of the set produced by the from clause, with the tuples of the same set having the same grouping value.

1.3 *Tuple identifier.* Since the tuples are assumed to contain a unique tuple identifier, the set defined by the above translation contains all the tuples that satisfy the Selection Expression, without any elimination of duplicates. This corresponds to an SQL query containing the ALL keyword in the select list, which is also the default. The treatment of the DISTINCT keyword is discussed at point 3.

1.4 The reason there is only one translation rule associated with syntax rule 1 for the external query block, but several translation rules associated with syntax rules 49 and 50 for subqueries having a column, or an aggregate function in their select-list, respectively, is that the translation of a subquery is used not only for generating the result set, but also for generating the translation of the complex predicate involving the subquery (see following point 2).

(2) *Translation of complex predicates.* A complex predicate is a predicate involving a subquery. The syntax of SQL has been rearranged in Table I so

that complex predicates are classified into (normal) ⟨COMPLEX PRED⟩, appearing in the where clause (rule 20) and, in some cases, the having clause (rule 35), and ⟨HACOMPLEX PRED⟩, appearing in the having clause and including a function in their expression (rule 38). Rules 20 and 38 are symmetrical, since the same predicates can appear in the where and having clauses. Each specific type of complex predicate, except the ⟨EXISTS PRED⟩ (rule 29) which does not admit functions in the select list of the subquery, has been subdivided into two subclasses: those predicates involving a subquery having a simple ⟨COL OR VAL⟩ in its select list (rule 49, ⟨SUBQ⟩) and those predicates involving a subquery having a function in its select list (rule 50, ⟨AFSUBQ⟩). Based on this classification, the translation rules for complex predicates, except for the complex comparison predicate (rules 30, 31, 47, 48), discussed at point 5, follow four slightly different schemes (translation rules 21–29, and 39–46):

2.1 *Complex predicate in a where clause, simple subquery (rules* 21, 23, 25, 27*).* The structure of the translation is

*quantifier TR*1 ⟨*SUBQ*⟩ ⟨*COL OR VAL*⟩⟨*compop*⟩ *TR*2 ⟨*SUBQ*⟩

where *TR*2 ⟨SUBQ⟩ is given by rule 49.2. The rationale of this structure consists in applying the quantifier to the set of tuples satisfying the Selection Expression, inasmuch as *TR*1 ⟨SUBQ⟩ is the set discussed at the above point 1, and for each tuple the simple ⟨COL OR VAL⟩ is compared with *TR*2 ⟨SUBQ⟩, which produces (rule 49.2), the column name contained in the select list of the subquery. The translation of the ⟨EXISTS PRED⟩ (rule 29) is obviously simpler.

2.2 *Complex predicate in a where clause,* ⟨*AFSUBQ*⟩ *(rules* 22, 24, 26, 28*).* The structure of the translation is essentially the same as the above; however, there are two differences:

(a) since the ⟨AFSUBQ⟩ contains a function in its select list, *TR*2 ⟨AFSUBQ⟩, given by translation rule 50.2, expresses the application of the function to the set obtained through the translation of the from clause, the where clause, and the group-by clause of the subquery. The translation of the having clause is not necessary, it is already included in *TR*1 ⟨AFSUBQ⟩, and therefore, the function is evaluated only for those groups which satisfy the having clause. (If there is not a group-by clause, the whole set *TR*1 ⟨AFSUBQ⟩ constitutes one group that has been either eliminated, or maintained as a whole by the having clause).

(b) An additional expression in these rules is necessary for dealing correctly with the cases in which the set produced by *TR*1 ⟨AFSUBQ⟩ is empty (see point 4).

2.3 ⟨*HACOMPLEX PRED*⟩ *in a having clause, simple subquery (rules* 39, 41, 43, 45*).* These translation rules are similar to point 2.1 with respect to the subquery, but they include the translation of a function applied to the set obtained through the translation of the from clause, the where clause, and the group-by clause of the outer query (subquery) block in their translation.

2.4 ⟨*HACOMPLEX PRED*⟩ *in a having clause* ⟨*AFSUBQ*⟩ *(rules* 40, 42, 44, 46*)*. The explanation of these rules is the combination of points 2.2 and 2.3.

(3) *Translation of the DISTINCT keyword.* The DISTINCT keyword can appear in the following situations in SQL, which are analyzed separately:

3.1 *After an aggregate function.* In this case, the capability of computing the right function is in the function itself, since E3VPC functions that do not include duplicates in their computation are defined. The rearrangement of the SQL syntax in rules 5–11, and the translation rules *TR*1 ⟨FUNC SPEC⟩ and *TR*2 ⟨FUNC SPEC⟩, defined at point 4 of the general description of the translation rules, produce the required result.

3.2 *In the select list of a subquery.* In this case, we do not need a specific translation for the DISTINCT keyword, since a subquery is always translated using a quantifier (notice that in Table II, *TR*1 ⟨SUBQ⟩, and *TR*1 ⟨AFSUBQ⟩, which are the real translations of a subquery, are always preceded by a quantifier, while *TR*2 ⟨SUBQ⟩ and *TR*2 ⟨AFSUBQ⟩ are irrelevant from this viewpoint, as they are used only to complete the join predicate).

The DISTINCT keyword can appear also in the select list of the main query, but this is not considered, as this select list is not translated.

(4) *Empty Sets produced by a subquery.* In translating a ⟨COMPLEX PRED⟩ or a ⟨HACOMPLEX PRED⟩ the E3VPC formula must be correct also in the case that *TR*1 ⟨SUBQ⟩, or *TR*1 ⟨AFSUBQ⟩ is empty. Inasmuch as a predicate having the structure $\exists TR1$ ⟨SUBQ⟩ or $\exists TR1$ ⟨AFSUBQ⟩ is always FALSE, if *TR*1 ⟨SUBQ⟩ or *TR*1 ⟨AFSUBQ⟩ is empty, and a predicate $\forall TR1$ ⟨SUBQ⟩ or $\forall TR1$ ⟨AFSUBQ⟩ is always TRUE if *TR*1 ⟨SUBQ⟩ or *TR*1 ⟨AFSUBQ⟩ is empty, some of the translations need an additional term to reflect the correct SQL semantics. In fact, the translations of ⟨COMPLEX PRED⟩ or ⟨HACOMPLEX PRED⟩ involving a simple ⟨SUBQ⟩ are already correct in the form stated at points 2.1 and 2.3, because the result of evaluating a predicate on a column of the result of a subquery which is empty, corresponds exactly to the semantics of the given E3VPC formula. However, if the subquery contains a function in its select list, thus being a ⟨AFSUBQ⟩, then the evaluation of the function produces a result, even if the set is empty (COUNT produces 0, the other functions produce a null value). For this reason, the translation rules referred to at points 2.2 and 2.4 need an additional term. The additional term is expressed by two different E3VPC expressions, for the cases of universal and existential quantification, respectively.

(5) *Complex comparison predicate (rules* 30, 31, 47, 48*)*. The translation rules of the complex comparison predicate with simple subquery (rules 30, 47) differ from points 2.1 and 2.3 only because they include an additional term for dealing with empty sets (in SQL, a complex comparison predicate, unlike other complex predicates, returns Unknown with an empty set). The translation rules for this predicate with ⟨AFSUBQ⟩ (rules 31, 48) are based on the

fact that in SQL the ⟨AFSUBQ⟩ in this predicate cannot contain a group-by clause, and therefore, the result of this subquery contains only one value.

Example. Consider a database containing two relations,

DEPT(d#, nofemp, location, manager)
EMP(e#, d#, residence)

and the query: Select the names of the managers of departments whose employees all have their residences in the same place as the department location, and such that the average number of employees of the selected departments, managed by the same manager, is greater than 500. This query can be expressed in SQL as

SELECT *d.manager*
FROM DEPT d
WHERE d.location = **ALL**
  SELECT e.residence
  FROM EMP e
  WHERE e.d# = d.d#
GROUP BY d.manager
HAVING AVG(d.nofemp) > 500

By applying translation rules 1–15, corresponding to the translation of the query block, we obtain the (incomplete) translation

(a) { d in DEPT: ‖ *TR*⟨WHERE SEARCH COND⟩
∧*TR*⟨HAVING SEARCH COND⟩ ‖ ^{F}}

A sequence of obvious rules and rule 23 allow substitution of *TR*⟨WHERE SEARCH COND⟩ with

(b) ∀*TR*1⟨SUBQ⟩ d.location = *TR*2⟨SUBQ⟩,

and the translations *TR*1 and *TR*2 of ⟨SUBQ⟩, stated by rules 49.1 and 49.2, produce, respectively,

(c) { e in EMP: ‖e.d# = d.d# ‖ ^{F}}

and

(d) e.residence.

Therefore, the whole structure of *TR*⟨WHERE SEARCH COND⟩ is (by combining (b), (c), and (d):

(e) ∀{ e in EMP: ‖e.d# = d.d# ‖ ^{F}} d.location = e.residence

The translation of ⟨HAVING SEARCH COND⟩ can be reduced by the obvious rules and rule 36 to

(f) AVG(d.nofemp){*TR*⟨FRCLAUSE⟩: ‖ *TR*⟨WHCLAUSE⟩
∧*TR*⟨GBCLAUSE⟩ ‖ ^{F}} > 500

and to

(g) AVG(d.nofemp){d in DEPT: ‖∀{e in EMP: ‖e.d# = d.d# ‖ ^{F}}d.location =
e.residence ∧ *TR*⟨GBCLAUSE⟩ ‖ ^{F}} > 500

Finally, by rule 15 we obtain the translation of the ⟨GB CLAUSE⟩:

$$\text{(h)}\quad \text{d.manager} \overset{\omega}{=} \text{d}\uparrow\text{.manager}$$

The complete translation of ⟨HAVING SEARCH COND⟩ is obtained by inserting expression (h) into expression (g), in place of TR⟨GBCLAUSE⟩.

Finally, the result of the complete translation of the original query is

$$\{\text{d in DEPT}:\|\forall\{\text{e in EMP}:\|\text{e.d\#} = \text{d.d\#}\|^F\}\ \text{d.location} = \text{e.residence} \wedge \wedge\text{AVG(d.nofemp)}\{\text{d in DEPT}:\|\forall\{\text{e in EMP}:\|\text{e.d\#} = \text{d.d\#}\|^F\}\text{d.location} = \text{e.residence} \wedge \wedge\text{d.manager} \overset{\omega}{=} \text{d}\uparrow\text{.manager}\|^F\} > 500\|^F\}$$

This is a correct E3VPC expression and defines the semantics of the query.

4. EQUIVALENCE OF SQL QUERIES

The equivalence of two SQL queries can be analyzed by applying the equivalence transformations of E3VPC to their translations. The equivalence transformation of E3VPC are those defined for three-valued predicate calculus in [8, 21], and some additional transformations reflecting the properties of the interpretation operator and compact forms with interpretation.

In Subsection 4.1 the additional rules of E3VPC are defined, and it is shown that they can be used for transforming a general E3VPC expression into a canonical form that can be manipulated using the usual rules of two-valued predicate calculus; since two-valued predicate calculus is well understood, we can say that the transformations given in Subsection 4.1 are a complete solution of the problem of deciding whether two SQL queries are equivalent. In Subsection 4.2 the effect of the specific approach taken by the SQL semantics with respect to three-valued logic and the interpretation of unknowns is analyzed using the rules given in Subsection 4.1; the goal of this analysis is to show that the SQL approach has an effect on the equivalence of queries (that is, two queries that would be equivalent in a two-valued interpretation are not equivalent in the real SQL) and to identify the SQL constructs responsible for this fact.

4.1 Equivalence Transformations on E3VPC Expressions

Definition. a *Canonical E3VPC expression* is an E3VPC expression where

(a) an interpretation operator is associated to each atomic predicate;
(b) no other interpretation operators are contained in the expression;
(c) the expression does not contain "compact forms."

A canonical E3VPC expression can be manipulated using the classical rules of two-valued predicate calculus, provided that each atomic predicate is always moved together with its associated interpretation operator, since an interpreted three-valued predicate is a two-valued predicate.

In order to transform a general E3VPC expression to a canonical form, we can use most of the usual properties of two-valued predicate calculus [15, 16,

Table III.

1.	$\| P(x) \vee Q(x) \|^{\alpha} \Leftrightarrow \| P(x) \|^{\alpha} \vee \| Q(x) \|^{\alpha}$
2	$\| P(x) \wedge Q(x) \|^{\alpha} \Leftrightarrow \| P(x) \|^{\alpha} \wedge \| Q(x) \|^{\alpha}$
3	$\| \neg P(x) \|^{\alpha} \Leftrightarrow \neg \| P(x) \|^{\alpha}$
4.	$\| \| P(x) \|^{\alpha} \|^{\beta} \Leftrightarrow \| P(x) \|^{\alpha}$
5.	$\| \exists x \text{ in } R: P(x) \|^{\alpha} \Leftrightarrow \exists x \text{ in } R: \| P(x) \|^{\alpha}$
6.	$\| \forall x \text{ in } R: P(x) \|^{\alpha} \Leftrightarrow \forall x \text{ in } R: \| P(x) \|^{\alpha}$

21, 26]; the only rules that are not valid when P is three-valued are the tautology $P \vee \neg P \Leftrightarrow 1$, and the contradiction $P \wedge \neg P \Leftrightarrow 0$.

These rules however are not sufficient, because they do not define how to move the interpretation operator from a whole subexpression to each single predicate.

Table III shows, therefore, the following additional rules:

—the distributive law of interpretation over logic operations OR and AND (rules 1 and 2);

—the movement of a negation operator over the boundary of an interpretation operator (rule 3);

—the idempotent law of an interpreted predicate (rule 4);

—the distributive law of interpretation over quantifiers (rules 5 and 6).

The proof of these rules is based on the simple application of three-valued truth tables, and is therefore omitted.

By applying these rules and the general rules of two-valued predicate calculus that are still valid, it is possible to transform any E3VPC expression into a canonical E3VPC expression. However, since the E3VPC expressions produced by an SQL query contain compact forms, and the elimination of compact forms requires a lengthy sequence of steps, the following two theorems are proved, which make the transformation straightforward.

THEOREM 1

$$\| \exists \{ x \text{ in } R: \| P(x) \|^{\alpha} \} Q(x) \|^{\beta} \Leftrightarrow \exists x \text{ in } R: \| P(x) \|^{\alpha} \wedge \| Q(x) \|^{\beta}$$

PROOF

$\| \exists \{ x \text{ in } R: \| P(x) \|^{\alpha} \} Q(x) \|^{\beta}$
$\Leftrightarrow \| \exists x \text{ in } R: (\| P(x) \|^{\alpha} \wedge Q(x)) \|^{\beta}$ (classical rule of predicate calculus)
$\Leftrightarrow \exists x \text{ in } R: \| \| P(x) \|^{\alpha} \wedge Q(x) \|^{\beta}$ (rule number 5 in Table III)
$\Leftrightarrow \exists x \text{ in } R: \| \| P(x) \|^{\alpha} \|^{\beta} \wedge \| Q(x) \|^{\beta}$ (rule number 2 in Table III)
$\Leftrightarrow \exists x \text{ in } R: \| P(x) \|^{\alpha} \wedge \| Q(x) \|^{\beta}$ (rule number 4 in Table III) □

THEOREM 2

$$\| \forall \{ x \text{ in } R: \| P(x) \|^{\alpha} \} Q(x) \|^{\beta} \Leftrightarrow \forall x \text{ in } R: \| \neg P(x) \|^{\neg \alpha} \vee \| Q(x) \|^{\beta}$$

PROOF

$\| \forall\{ x \text{ in } R: \| P(x) \|^{\alpha}\} Q(x) \|^{\beta}$
$\Leftrightarrow \| \forall x \text{ in } R: (\neg \| P(x) \|^{\alpha} \vee Q(x) \|^{\beta})$ (classical rule of predicate calculus)
$\Leftrightarrow \forall x \text{ in } R: \| \neg \| P(x) \|^{\alpha} \vee Q(x) \|^{\beta}$ (rule number 6 in Table III)
$\Leftrightarrow \forall x \text{ in } R: \| \neg \| P(x) \|^{\alpha} \|^{\beta} \vee \| Q(x) \|^{\beta}$ (rule number 1 in Table III)
$\Leftrightarrow \forall x \text{ in } R: \| \| \neg P(x) \|^{\neg \alpha} \|^{\beta} \vee \| Q(x) \|^{\beta}$ (rule number 3 in Table III)
$\Leftrightarrow \forall x \text{ in } R: \| \neg P(x) \|^{\neg \alpha} \vee \| Q(x) \|^{\beta}$ (rule number 4 in Table III) □

4.2 The Effect of Three-Valuedness on the Equivalence of SQL Queries

The equivalence transformations on E3VPC expressions and the translation rules constitute a complete solution to the problem of analyzing the equivalence of SQL queries. However, it is interesting to investigate whether the way in which SQL interprets the unknown value has an effect on the equivalence of SQL queries.

The effect of the interpretation of an unknown is characterized by the existence of SQL queries that can be translated and transformed to canonical E3VPC expressions having an identical structure but differing only in the type of interpretation associated to some elementary predicates. In the sequel, we call a set of queries having this property a *Critical Equivalence Set*.

For example, suppose that the translations of two SQL queries Q_1 and Q_2 can be transformed so that we obtain two E3VPC expressions such as:

$$TR\langle Q_1 \rangle = \{ x \text{ in } R: \| P(x) \|^{F} \}$$
$$TR\langle Q_2 \rangle = \{ x \text{ in } R: \| P(x) \|^{T} \}$$

These two queries produce identical results if they are executed on a database without null-values, but they are not equivalent on a general database, since a tuple x′ such that $P(x') = \mathrm{U}$ is included in the result of Q_2 but not of Q_1. Q_1 and Q_2 therefore constitute a critical equivalence set.

It is interesting to understand which kind of SQL constructs can produce this effect. In the following analysis, we consider two elementary predicates to be identical only if they are *syntactically* identical, and assume that no logical relationship can exist between two different predicates $P(x)$ and $Q(x)$. This assumption means that $A = B$, and $A \neq B$ are considered different if the latter is not transformed to $\neg (A = B)$.

Notice that, since the translation rules given in Section 3 produces only the interpretation to false ($\| \ldots \|^{F}$), situations like that of Q_1 and Q_2 cannot be generated directly by the translation rules; however, since the translations are not directly in canonical form, we must investigate whether the transformations that must be applied to the translations to produce the canonical forms can create a critical equivalence set.

In fact, there are two equivalence transformations that can potentially produce this situation; they are expressed by rules 3 and 4 of Table III that transform, respectively, a $\| \ldots \|^{\alpha}$ into a $\| \ldots \|^{\neg \alpha}$, and a $\| \| \ldots \|^{\alpha} \|^{\beta}$ into a $\| \ldots \|^{\alpha}$. The reason why rule 3 can potentially produce a critical equivalence

set is obvious; concerning rule 4, its potential for generating a critical situation is due to a possible transformation of the kind, $\| \| P \|^T \|^F \Rightarrow \| P \|^T$, and, therefore, to the existence of a true-interpreted predicate.

We can now investigate whether there are types of queries in SQL that belong to critical equivalence sets.

Simple queries (without nesting and quantification). It is easy to show that if two queries Q_1 and Q_2 are simple, they cannot constitute a critical equivalence set. In order to produce a critical equivalence set by rule 3 or rule 4, the queries must be such that one of the following situations arise

$$\left.\begin{aligned} TR\langle Q_1\rangle &= \{x \text{ in } X : \| \neg\ P(x) \|^F\} \\ TR\langle Q_2\rangle &= \{x \text{ in } R : \neg\ \| P(x) \|^F\} \Rightarrow \{x \text{ in } R : \| \neg\ P(x) \|^T\} \end{aligned}\right\} \textit{rule } 3$$

or

$$\left.\begin{aligned} TR\langle Q_1\rangle &= \{x \text{ in } R : \| P(x) \|^F\} \\ TR\langle Q_2\rangle &= \{x \text{ in } R : \| \| P(x) \|^T \|^F\} \Rightarrow \{x \text{ in } R : \| P(x) \|^T\} \end{aligned}\right\} \textit{rule } 4$$

However, there is no query in SQL like Q_2, and therefore none of these situations can be produced.

Universal quantification. A universal quantified predicate can be expressed in five different basic ways in SQL, which are all equivalent if two-valued logic interpretation is considered; the translation of these five forms, as obtained through the rules of Section 3, are

(1) $\| \neg\ \exists \{ y \text{ in } S : \| Q(x, y) \wedge P(y) \|^\alpha \} \|^\beta$
(2) $\| \neg\ \exists \{ y \text{ in } S : \| Q(x, y) \|^\alpha \} P(y) \|^\beta$
(3) $\| \neg\ \exists \{ y \text{ in } S : \| P(y) \|^\alpha \} Q(x, y) \|^\beta$
(4) $\| \forall \{ y \text{ in } S : \| Q(x, y) \|^\alpha \} \neg\ P(y) \|^\beta$
(5) $\| \forall \{ y \text{ in } S : \| P(y) \|^\alpha \} \neg\ Q(x, y) \|^\beta$

where $\alpha = \mathrm{F}$ always, since in SQL, a search condition is interpreted to false, and β depends on the general structure of the query containing the predicate.

These five E3VPC expressions can be transformed, using the theorems of the previous subsection, into the following three equivalent canonical E3VPC expressions:

(1′) $\forall y \text{ in } S : \| \neg\ Q(x, y) \|^{\neg \alpha} \vee \| \neg\ P(y) \|^{\neg \alpha}$
(2′) $\forall y \text{ in } S : \| \neg\ Q(x, y) \|^{\neg \alpha} \vee \| \neg\ P(y) \|^{\beta}$
(3′) $\forall y \text{ in } S : \| \neg\ Q(x, y) \|^{\beta} \vee \| \neg\ P(y) \|^{\neg \alpha}$
(4′) = 2′.
(5′) = 3′.

These expressions are equivalent only if $\beta = \neg\ \alpha$. However, in general this is not true; for example, if the quantified predicate is the only complex predicate of an otherwise simple query, then $\alpha = \beta = \mathrm{F}$. This is shown by the five

Consider a database schema containing two relations

$R(A_1, .., A_i, .., A_n)$

$S(B_1, .., B_i, .., B_j, .., B_m)$

and the 5 SQL queries

Q_1 **SELECT** *
FROM $R\ x$
WHERE NOT EXISTS
SELECT *
FROM $S\ y$
WHERE $x.A_i = y.B_i \wedge y.B_j \neq 'P'$

Q_2 **SELECT** *
FROM $R\ x$
WHERE NOT $'P' \neq$ **SOME**
SELECT $y.B_j$
FROM $S\ y$
WHERE $x.A_i = y.B_i$

Figure 1

Q_3 **SELECT** *
FROM $R\ x$
WHERE NOT $x.A_i$ **IN**
SELECT $y.B_i$
FROM $S\ y$
WHERE $x.B_j \neq 'P'$

Q_4 **SELECT** *
FROM $R\ x$
WHERE $'P' =$ **ALL**
SELECT $y.B_j$
FROM $S\ y$
WHERE $x.A_i = y.B_i$

Q_5 **SELECT** *
FROM $R\ x$
WHERE $x.A_i$ **NOT IN**
SELECT $y.B_i$
FROM $S\ y$
WHERE $y.B_j \neq 'P'$

queries of Figure 1; the translations of the five, where search conditions correspond to the five E3VPC expressions defined above, where α and β are both false, $Q(x, y)$ is $x.A_i = y.B_i$ and $P(y)$ is $y.B_j \neq 'P'$.

Existential quantification. Existential quantification can be expressed in SQL in five different basic ways, which produce the following translations:

(1) $\|\exists\{y \text{ in } S: \| Q(x, y) \wedge P(y) \|^{\alpha}\}\|^{\beta}$
(2) $\|\exists\{y \text{ in } S: \| Q(x, y) \|^{\alpha}\} P(y) \|^{\beta}$
(3) $\|\exists\{y \text{ in } S: \| P(y) \|^{\alpha}\} Q(x, y) \|^{\beta}$
(4) $\| \neg\ \forall\{y \text{ in } S: \| Q(x, y) \|^{\alpha}\} \neg\ P(y) \|^{\beta}$
(5) $\| \neg\ \forall\{y \text{ in } S: \| P(y) \|^{\alpha}\} \neg\ Q(x, y) \|^{\beta}$

and the same considerations apply here to α and β as for universal quantification.

These five E3VPC expressions can be transformed into the following three equivalent canonical E3VPC expressions:

(1′) $\exists y \text{ in } S: \| Q(x, y) \|^{\alpha} \wedge \| P(y) \|^{\alpha}$
(2′) $\exists y \text{ in } S: \| Q(x, y) \|^{\alpha} \wedge \| P(y) \|^{\beta}$
(3′) $\exists y \text{ in } S: \| Q(x, y) \|^{\beta} \wedge \| P(y) \|^{\alpha}$
(4′) = 2′.
(5′) = 3′.

Unlike the case of universal quantification, these forms are equivalent only if $\alpha = \beta$. Since $\alpha = \mathrm{F}$ always, a critical equivalence set can be generated only if the existentially quantified predicate is contained in a query structure that produces a translation with $\beta = \mathrm{T}$; this is possible only if the existentially quantified predicate is one of the constituent predicates (P or Q) of a universally quantified predicate, since, as shown above, the universal quantification can generate the interpretation to true of some of its internal predicates.

Complex comparison predicate. A complex comparison predicate can be expressed only in one way in SQL, and therefore, its translation produces only one equivalence class.

General SQL queries. We can summarize the above investigation of the effect of the interpretation of unknowns on the equivalence of SQL queries in the following way:

(1) SQL queries not containing universal quantification never belong to a critical equivalence set;
(2) SQL queries containing universal quantification always produce critical equivalence sets;
(3) the number of different interpretations of predicates of queries having the same canonical structure and at least one universally quantified predicate, is increased by the existence of other (existentially and universally) quantified predicates contained in the universally quantified predicate.

5. CONCLUSIONS

The semantics of SQL queries has been formally defined by a set of translation rules which produce equivalent expressions of an extended predicate calculus, called E3VPC. SQL is usually considered a nonprocedural language; however, the specification of its semantics given in [3] is procedural, because it states the order in which different parts of a query are evaluated (e.g., the *having* clause is evaluated after the *where* clause and the *group-by* clause). This paper has presented a completely non-procedural specification, which is formal and much more compact than the procedural specification of [3], and which constitutes a powerful tool for understanding the deep semantics of queries, and of the rules that can be applied for query transformation.

The equivalence rules that allow determination whether two SQL queries are equivalent have also been stated. It has been shown that the equivalence

of two SQL queries containing quantified predicates cannot be analyzed by simply using two-valued logic, because the three-valuedness and the interpretation of unknowns of the SQL semantics produce different equivalence classes than a pure two-valued predicate calculus. Therefore, the design of a transformer of SQL queries (typically, a precompiler which reduces queries to standard forms *before* optimization) should be safely based on the rules given here.

Finally, the analysis of the translation rules from SQL to a formal model, could be used as a method for ameliorating the SQL language itself, as some of the complexities of the given translations are due to the procedural nature of the SQL definition, which has no straightforward counterpart in logic. E3VPC, which has been shown to possess at least the same expressive power of SQL, could be used as a basis for the definition of a more rigorous language having the same power of SQL; however, this has been left as a subject for further work.

REFERENCES

1. AHO, A. V., SAGIV, Y., AND ULLMAN, J. D. Efficient optimization of a class of relational expressions. *ACM Trans. Database Syst. 4* (1979), 435–454.
2. AHO, A. V., SAGIV, Y., AND ULLMAN, J. D. Equivalence of relational expressions. *SIAM J. Comput. 8* (1979), 218–246.
3. AMERICAN NATIONAL STANDARD INSTITUTE. *American National Standard Database Language SQL.* ISO/TC97/SC21/WG5-15, n. 90, Washington D.C., 1985.
4. BÜLTZINGWLOEWEN, G. V. Translating and optimizing SQL queries having aggregates. In *Proceedings of the 13th Conference on Very Large Data Bases* (Brighton, 1987).
5. CERI, S., AND GOTTLOB, G. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL Queries. *IEEE Trans. Softw. Eng. SE 11, 4* (1985), 324–345.
6. CHAMBERLIN, D. D. ET AL. Sequel 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev. 20* (1976), 560–575.
7. CODD, E. F. Understanding relations (Installment 7) *FDT Bull. ACM-SIGMOD 7* (1975), 23–28.
8. CODD, E. F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4* (1979), 397–434.
9. DATE, C. J. *A Guide to the SQL Standard.* Addison-Wesley, Reading, Mass., 1987.
10. DAYAL, U. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates and quantifiers. In *Proceedings of the 13th Conference on Very Large Data Bases* (Brighton, 1987).
11. GALLAIRE, H. Impacts of logic on data bases. In *Proceedings of the 7th Conference on Very Large Data Bases* (Cannes, 1981).
12. GANSKY, R. A., AND WONG, H. K. T. Optimization of nested SQL queries revisited. In *Proceedings of the International Conference ACM SIGMOD* (San Francisco, 1987), 23–33.
13. GOTTLOB, G., AND ZICARI, R. Closed world databases opened through null values. In *Proceedings of the 14th Conference on Very Large Data Bases* (Los Angeles, 1988).
14. GRANT, J. Null values in a relational database. *Inf. Process. Lett. 5* (1977), 156–157.
15. JARKE, M., AND SCHMIDT, J. W. Query processing strategies in the PASCAL/R relational database management system. In *Proceedings of the International Conference ACM SIGMOD* (Orlando, Fla., 1982), 256–263.
16. JARKE, M., AND KOCH, J. Query optimization in database systems. *ACM Comput. Surv.* (1984), 111–152.
17. KIM, W. On optimizing an SQL-like nested query. *ACM Trans. Database Syst. 7,* 3 (1982), 443–469.

18. KLUG, A. On inequality tableaux. Computer Science Tech Rep. 403, Univ. of Wisconsin, Madison, 1980.
19. KLUG, A. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM 29,* 3 (1982), 699–717.
20. LIPSKI, W., JR. On semantic issues connected with incomplete information databases. *ACM Trans. Database Syst. 4,* 3 (1979), 262–296.
21. NEGRI, M., PELAGATTI, G., AND SBATTELLA, L. The effect of three-valued predicates on the semantics and equivalence of SQL queries Tech Rep. 85-27, Dip di Elettronica, Politecnico di Milano, 1985.
22. ROSENTHAL, A., AND REINER, D. An architecture for query optimization. In *Proceedings of the International Conference ACM SIGMOD* (Orlando, Fla., 1982), 246–255.
23 SAGIV, Y., AND YANNAKAKIS, M. Equivalence among relational expressions with the union and difference operators. *J. ACM 27* (1980), 633–655.
24. SELINGER, P. ET AL. Access path selection in a relational database system. In *Proceedings of the International Conference ACM SIGMOD* (Boston, Mass., 1979), 23–24.
25. SMITH, J. M., AND CHANG, P. Y. T. Optimizing the performance of a relational algebra database interface. *Commun. ACM 18* (1975), 568–579.
26 STANAT, D. F., AND MCALLISTER, D F *Discrete Mathematics in Computer Science.* Prentice Hall, Englewood Cliffs, N.J., 1977.
27. VASSILIOU, Y. Null values in database management: A denotational semantics approach. In *Proceedings of the International Conference ACM SIGMOD* (Boston, Mass., 1979).
28. WONG, E , AND YOUSSEFI, K. Decomposition: A strategy for query processing. *ACM Trans. Database Syst. 1,* 3 (1976), 223–241.

Received November 1985; final revision February 1990; accepted March 1990