Technique for Universal Quantification in SQL

Claudio Fratarcangeli
Black Diamond Software, Inc.
127 Wilton Road East
Ridgefield, Connecticut 06877

Abstract

Universal quantification is expressed in ANSI SQL with negated existential quantification because there is no direct support for universal quantification in ANSI SQL. However, the lack of explicit support for universal quantification diminishes the userfriendliness of the language because some queries are expressed more naturally using universal quantification than they are using negated existential quantification. It is the intent of this paper to describe a technique to facilitate the construction of universal quantification queries in ANSI SQL. The technique is based upon a proposed extension to ANSI SQL to incorporate explicit general support for universal quantification.

1. Introduction

SQL is currently accepted as the standard in relational DBMS query languages. It owes much of its success to the fact that it has a strong theoretical basis in tuple relational calculus and relational algebra. (To be precise it is based on a subset of tuple relational calculus called safe tuple relational calculus) [MAIE83, ULLM88]. SQL's relationship to tuple relational calculus is evidenced by the syntactic similarities between the two languages. For example, comparison predicates are essentially identical in syntax. Logical predicates differ only in that the English words, AND, OR, and NOT are used in place of their mathematical symbolic counterparts.

Of course, there are significant differences as well. In particular,there is no direct counter part for the universal quantifier symbol of tuple relational calculus in SQL.The only SQL construct that resembles a universal quantifier symbol is the ALL keyword and it only supports a limited form of universal quantification.

Despite the absence of a direct counterpart for a universal quantifier symbol, there is a direct counterpart for the existential quantifier symbol of tuple relational calculus, i.e. the EXISTS keyword. Consequently, the absence of a universal quantifier in SQL does not limit the expressiveness of the language, because it is possible to express universal quantification with existential quantification using the law:

FORALL (X) == NOT EXISTS (NOT X)

However, it is the author's opinion that the lack of explicit support for universal quantification diminishes the userfriendliness of the language, simply because some queries are expressed more naturally using universal quantification than they are using negated existential quantification. Thus,it is the intent of this article to describe a technique to facilitate the construction of universal quantification queries. The technique is based upon a proposed extension to ANSI SQL to incorporate explicit general support for universal quantification.

The remainder of the article is organized as follows. 1) Basic SQL terminology is reviewed. 2) Quantification in tuple relational calculus is reviewed. 3) Sample tables that will be used in examples throughout the article are presented. 4) A technique to facilitate construction of universal quantification queries in SQL is described. 5) A syntax and semantics for universal quantification in SQL is introduced. 6) A procedure for transforming predicates using the new universal quantification syntax into ANSI SQL is described. 7) A revised syntax and semantics is described to take into account the GROUP BY and HAVING clauses of SQL and a revised transformation algorithm is presented. 8) The relationship between the new universal quantification syntax and the ANSI ALL predicate is discussed. 9) Special considerations concerning NULL values are examined. 10) The article is concluded.

2. SQL Terminology

In this section we present SQL terminology to be used in the remainder of the paper. The terminology used to reference syntactic elements of an SQL statement is borrowed from the SQL ANSI standard [ANSI89]. Thus, a <correlation-name> is the alias that is associated with a table name in a FROM clause. A <search-condition> is a boolean expression associated with a WHERE clause or a HAVING clause. A <value-expression> is either a numeric, date, or character expression.

3. Review of Quantification in Tuple Relational Calculus

It is beyond the scope of this paper to present a full description of safe tuple relational calculus. So the focus of this review is on quantification. A variable, X, is free in a formula of tuple relational calculus, F, if F is a simple comparison predicate, or it is of one of the following forms: F1 AND F2, NOT F1, F1 OR F2, (F1), where F1

and F2 are formulas in which X is also free. In general, a quantification predicate is a formula, Qf, of safe tuple relational calculus of the form:

(QX)(F)

where Q is one of the quantification symbols U or E, F is itself a formula of the tuple relational calculus, and X is a tuple variable. If X occurs as a free variable in F in at least one occurrence, all free occurrences of X in F are said to be bound in the quantification predicate formula, Qf, by the quantifier, (QX). All other variables in F, are free or bound depending upon whether or not they are free or bound in F. As an example, in the formula,

(EX)(X(NAME) = Y(NAME))

X is bound, while Y is free. In the subformula,

(X(NAME) = Y(NAME))

both X and Y are free.

The quantification symbol, E, is the existential quantifier.The interpretation of the predicate formula, Qf, when the quantifier symbol is E is the following. The predicate evaluates to true if there exists a value that can be substituted for X that causes the formula F to evaluate to true. Since X is a tuple variable of a specified arity (i.e. It has a specified number of components) a value for X must come from the domain of all tuples with the arity of X.

When Q is the universal quantifier, U, the interpretation is the following. The predicate evaluates to true if for all possible values of X, F evaluates to true. Again the domain of X is all possible tuples with the same arity as X.

4. Sample Tables

All examples presented below refer to the following set of tables.

Table: CUSTOMERS

NAME	CUST_ID	CITY	STATE
LISA	C1	TRUMBULL	CT
JOHN	C2	POMONA	NY
PAUL	C3	FAIRFIELD	CT
JACOB	C4	TAMPA	FL

Table: SALESMAN

NAME	SLM_ID	CITY	STATE	AGE
JIM	S1	SHELTON	CT	30
JACK	S2	NEW YORK	NY	22
GEORGE	S3	BERKELEY	CA	40

Table: PRODUCTS

PROD_ID	DESCRIPTION	COLOR
P1	PANTS	BLUE
P2	SHIRT	RED
P3	HAT	RED
P4	SOCKS	YELLOW
P5	HAMMER	

Table: ORDERS
Columns: ORDER#, DATE, MONTH, CUST_ID, PROD_ID, SLM_ID, AMOUNT, COMMIS

ORD#	DATE	MONTH	CUST_ID	PROD_ID	SLM_ID	AMT	COM
01	1/1/89	1	C1	P1	S1	23	
02	1/1/89	1	C1	P2	S2	23	1
03	11/11/89	11	C1	P3	S3	23	42
04	12/12/89	12	C2	P1	S3	23	53
05	11/13/89	11	C2	P4	S2	23	
06	9/13/89	9	C2	P5	S1	23	55
07	8/13/89	8	C3	P2	S2	23	63
08	8/14/89	8	C3	P5	S3	23	22
09	8/14/89	8	C4	P1	S2	23	11
010	8/14/89	8	C4	P4	S3	23	55

5. Technique to Facilitate Universal Quantification

The technique to facilitate construction of universal quantification queries in SQL is based upon a proposed extension to SQL to provide user-friendly general support for universal quantification. The technique involves the following steps:

- Construction of the SQL query using the proposed extension.
- Transforming the SQL with the proposed extension into ANSI SQL using a simple transformation procedure.

The utility of the technique derives from the fact that construction of universal quantification queries using the proposed extension is relatively straightforward. Once the SQL is constructed, generation of ANSI SQL involves the application of a simple transformation procedure. The proposed extension and the transformation procedure to ANSI SQL will be discussed in detail in subsequent sections.

6. SQL Syntax for Universal Quantification

The extension that the author proposes is a new type of predicate called a FORALL predicate in the WHERE clause of SQL. The BNF for the FORALL predicate is:

```
<forall-predicate> ::=
        FORALL ( <table-reference> [ , ... ]
                [ <where-clause> ]
                <forall-search-condition> )
```

```
<table-reference> ::=
        <table-name> [ <correlation-name> ]
```

The <where-clause> has the form of a standard SQL WHERE clause and the <forall-search-condition> has the form of a standard SQL <search-condition>.

The semantics of the FORALL predicate are as follows. The cross product of the tables referenced in the list of <table-reference>'s is formed. The predicate evaluates to true if for each row in the cross product that satisfies the <where-clause>, the <forall-search-condition> evaluates to true. If there is no <where-clause>, the FORALL predicate evaluates to true only if all rows of the cross product satisfy the <forall-search-condition>. Otherwise it evaluates to false.

As an example, consider the query, "What customers have bought all the red products". A logical expansion of this query is "Show customers such that for all products that are red there exists an order for the product by the customer." The SQL for this query using the proposed extension is:

```
SELECT * FROM CUSTOMERS CU
  WHERE
    FORALL (PRODUCTS PR
      WHERE COLOR = 'RED'
      EXISTS ( SELECT 1 FROM ORDERS ORD
          WHERE ORD.PROD_ID = PR.PROD_ID
            AND ORD.CUST_ID = CU.CUST_ID))
```

As can be seen the SQL follows the logical expansion of the query fairly closely. The semantics of the above query are the following. For each customer row the FORALL predicate is evaluated. If it evaluates to true the customer row is returned. Otherwise it is not returned. The evaluation of the FORALL predicate proceeds as follows. For each row in the PRODUCTS table the WHERE clause is evaluated. If the WHERE clause doesn't evaluate to true the row is skipped and the next one is processed. If the WHERE clause does evaluate to true, then the EXISTS is evaluated. If the EXISTS evaluates to false, the FORALL evaluates to false terminating evaluation of the FORALL predicate. If the EXISTS evaluates to true, then the next row in the PRODUCTS table is processed. If all rows of the PRODUCTS table are processed successfully, the FORALL predicate evaluates to true.

To appreciate the userfriendliness of the FORALL predicate, compare the above query with the following semantically equivalent ANSI SQL query.

```
SELECT * FROM CUSTOMERS CU
WHERE
  NOT EXISTS (SELECT PRODUCTS PR
     WHERE COLOR = 'RED' AND
       NOT EXISTS ( SELECT 1
          FROM ORDERS ORD
          WHERE ORD.PROD_ID = PR.PROD_ID
            AND ORD.CUST_ID = CU.CUST_ID))
```

The author believes that the ANSI SQL form is significantly less intuitive than the FORALL form.

As the BNF indicates the <where-clause> is optional in the <forall-predicate>. A variation on the original example SQL that is missing a <where-clause> follows.

```
SELECT * FROM CUSTOMERS CU
WHERE
  FORALL (PRODUCTS PR
    EXISTS ( SELECT 1 FROM ORDERS ORD
       WHERE ORD.PROD_ID = PR.PROD_ID
         AND ORD.CUST_ID = CU.CUST_ID))
```

It answers the query, "What customers bought all products", or rewording it, "Show customers such that for all products there exists an order for the product by the customer."

7. Transforming the FORALL Predicate to ANSI SQL

It was stated earlier that the lack of explicit support for the universal quantifier does not limit the expressiveness of SQL because universal quantification can be expressed with existential quantification. It is therefore appropriate to demonstrate how a query containing a FORALL predicate can be transformed into ANSI SQL.

In general, a FORALL predicate represented by the following formula

```
FORALL (T WHERE A B)
```

(where T is a list of <table-reference>'s separated by commas, A is a standard WHERE clause <search-condition>, and B is a<forall-search-condition>) is semantically equivalent to the following ANSI SQL

```
NOT EXISTS (SELECT 1 FROM T WHERE A AND
        NOT B)
```

The following shows why this is so. It was stated previously that the FORALL predicate evaluates to true if for each row in the cross product of the <table-reference>'s the following is true: If the row satisfies the criterion in the <where-clause> then it must also satisfy the criterion in the <forall-search-condition>. Thus, it is semantically equivalent to the following tuple relational calculus-like formula in which keywords with the obvious meaning are

used in place of the mathematical symbols for quantifiers, boolean operators and comparison operators.

(FORALL T) (A IMPLIES B)

The formula evaluates to true if for all rows in T, if A evaluates to true then B must evaluate to true. Otherwise, it evaluates to false. Propositional logic tells us that this is in turn equivalent to.

(FORALL T) (NOT A OR B)

The law mentioned above that showed the equivalence between a formula containing universal quantification and a formula containing existential quantification, can be used to construct the following semantically equivalent form.

NOT (EXISTS T) (NOT (NOT A OR B))

This formula evaluates to true only if there does not exist a row in T such that the predicate in parentheses evaluates to true. Reducing the scope of the NOT yields.

NOT (EXISTS T) (A AND NOT B)

Transforming the tuple relational calculus-like notation to ANSI SQL yields the following,

NOT EXISTS (SELECT 1 FROM T WHERE A AND NOT B)

The above discussion did not take into account the case where the <where-clause> is absent. In the absence of an explicit <where-clause> an implicit <where-clause> that always evaluates to true is assumed. This can be written as

FORALL (T WHERE true B)

Following the logic presented above this is equivalent to the following tuple relational calculus-like formula.

(FORALL T) (NOT true OR B)

This is in turn equivalent to

(FORALL T) (false OR B)

which is the same as

(FORALL T) (B)

Using the law of equivalence between universal quantification and negated existential quantification this becomes

NOT (EXISTS T) (NOT B)

The ANSI SQL equivalent is

NOT EXISTS (SELECT 1 FROM T WHERE NOT B)

We are now in a position to present a formula for transforming a FORALL predicate into ANSI SQL. The procedure is as follows. Assume the following FORALL predicate.

FORALL (T [WHERE A] B)

Depending upon whether or not the <where-clause> is present, the ANSI SQL equivalent form varies as follows.

- The <where-clause> is present:

NOT EXISTS (SELECT 1 FROM T
WHERE (A) AND NOT (B))

- The <where-clause> is absent:

NOT EXISTS (SELECT 1 FROM T
WHERE NOT (B))

8. Group By and Having

Although the GROUP BY and HAVING clauses do not have a counterpart in tuple relational calculus, they are part of SQL and a place should be found for them in the FORALL predicate. This can be accomplished by revising the BNF for the FORALL predicate as follows.

<forall-predicate> ::=

FORALL (<table-reference> [, ...]
[<where-clause>]
[<group-by-clause>]
[<having-clause>]
<forall-search-condition>)

The <group-by-clause> and <having-clause> are identical in syntax to their counterparts in ANSI SQL. The revised semantics can be stated informally as follows. If the <group-by-clause> and <having-clause> are not present the semantics are identical to those stated for the original BNF. If either clause is present the semantics are as follows. The cross product of the table references in the <table-reference> list is formed. If a <group-by-clause> is present the rows of the cross product are grouped in the usual fashion to form a new table with a single row replacing all rows within each group. In the absence of a <group-by-clause>, if either a <having-clause> is present or a <forall-search-condition> is present that contains references to grouping functions like MIN or MAX, then there is an implicit grouping operation that creates a single grouped row out of the entire cross product. For each row in the resulting table the <having-clause> is evaluated if present. If the <having-clause> evaluates to false, the row is processed successfully and the next one is processed. If the <having-clause> is absent or, if present, it evaluates to

true, the <forall-search-condition> is evaluated. If it evaluates to true, the row is processed successfully and evaluation continues with the next row. If it does not evaluate to true, the FORALL predicate evaluates to false and evaluation ceases. If all rows are evaluated successfully the FORALL predicate evaluates to true.

It should be noted that if a <group-by-clause> or <having-clause> is present, the <forall-search-condition> is subject to the same restrictions that the <search-condition> of the <having-clause> is subject to. This is because under these circumstances the <forall-search-condition> applies to the rows of the table resulting from the grouping operation as does the <having-clause>.

As an example, consider the query, "Show salesmen whose total sales for each month in 1989 exceeded $50." The query can be reworded as, "Show salesmen such that for all months in 1989, the salesman's total monthly sales amount exceeded $50." Assume for the sake of simplicity that each salesman made at least one sale each month of 1989. The SQL is

```
SELECT * FROM SALESMAN SM
  WHERE
    FORALL (ORDERS ORD
      WHERE DATE BETWEEN '1/1/89' AND
         '12/31/89' AND
        SM.SLM_ID = ORD.SLM_ID
      GROUP BY MONTH
      SUM(AMOUNT) > 50)
```

Although the SQL does not flow quite as naturally from the English reworded query as did prior examples, it is still significantly more intuitive than the following ANSI SQL equivalent which follows.

```
SELECT * FROM SALESMAN SM
  WHERE
    NOT EXISTS (SELECT 1 FROM ORDERS ORD
       WHERE DATE BETWEEN '1/1/89' AND
          '12/31/89' AND
         SM.SLM_ID = ORD.SLM_ID
       GROUP BY MONTH
       HAVING NOT SUM(AMOUNT) > 50)
```

As another example consider the query, "Show salesmen who for all months in 1989 in which their total sales exceeded $50, their total commissions for the month exceed $40." This can be reworded as, "Show salesman such that for all months in 1989 in which their monthly sales total exceeded $50, their total commission for the month exceeded $40." The SQL is

```
SELECT * FROM SALESMAN SM
  WHERE
    FORALL (ORDERS ORD
      WHERE
        DATE BETWEEN '1/1/89' AND '12/31/89'
        AND SM.SLM_ID = ORD.SLM_ID
      GROUP BY MONTH
      HAVING SUM(AMOUNT) > 50
      SUM(COMMIS) > 40)
```

The ANSI SQL equivalent is

```
SELECT * FROM SALESMAN SM
WHERE
  NOT EXISTS (SELECT 1 FROM ORDERS ORD
     WHERE
       DATE BETWEEN '1/1/89' AND '12/31/89'
       AND SM.SLM_ID = ORD.SLM_ID
     GROUP BY MONTH
     HAVING SUM(AMOUNT) > 50 AND
       NOT SUM(COMMIS) > 40)
```

As a last example, consider the query, "Show salesmen whose total sales in 1989 exceeded $1000." The SQL for this query is

```
SELECT * FROM SALESMAN SM
WHERE
  FORALL (ORDERS ORD
     WHERE
       DATE BETWEEN '1/1/89' AND '12/31/89'
       AND SM.SLM_ID = ORD.SLM_ID
     SUM(AMOUNT) > 1000)
```

The equivalent query using the NOT EXISTS construct is

```
SELECT * FROM SALESMAN SM
WHERE
  NOT EXISTS (SELECT 1 FROM ORDERS ORD
     WHERE
       DATE BETWEEN '1/1/89' AND '12/31/89'
       AND SM.ID = ORD.SALESMAN_ID
     HAVING NOT SUM(AMOUNT) > 1000)
```

Although in practice the same query would probably be answered using a more intuitive construct than either FORALL or NOT EXISTS, it is presented to illustrate that the <forall-search-condition> can contain a group function like SUM even in the absence of a <group-by-clause> or a <having-clause>. In such cases, there is an implicit grouping of all rows into a single group and the <forall-search-condition> applies to the single resulting row.

9. Revised Algorithm for Transforming FORALL to ANSI SQL

The logic used to derive the modified procedure for

transforming an SQL query using the revised FORALL syntax to ANSI SQL is similar to that used to derive the first version of the procedure. So the modified procedure will be presented without including the actual details of the derivation. Assume the following FORALL predicate.

```
FORALL (T [ WHERE A ] [ GROUP BY B ]
        [ HAVING C ] D)
```

The ANSI SQL equivalent form varies depending upon which one of a number of conditions applies. Each condition is listed along with the ANSI SQL form that applies if the condition is met.

- If both the <group-by-clause> and the <having-clause> are absent and D does not contain references to group functions, the original version of the transformation procedure is used to obtain the ANSI SQL equivalent. Otherwise, one of the other conditions apply.

- The <having-clause> is present:

```
NOT EXISTS (SELECT 1 FROM T [ WHERE A ]
        [ GROUP BY B ]
          HAVING (C) AND NOT (D))
```

- The <having-clause> is absent:

```
NOT EXISTS (SELECT 1 FROM T [ WHERE A ]
        [ GROUP BY B ] HAVING NOT (D))
```

10. ALL Predicate in ANSI SQL

ANSI SQL has limited direct support for universal quantification in the form of the ALL predicate. The syntax for the ALL predicate is

```
<value-expression> <comparison-operator>
     ALL <subquery>
```

The predicate evaluates to true only if "<value-expression> <comparison-operator> X" evaluates to true for all possible values of X returned by the <subquery>.

The ANSI SQL ALL predicate is semantically equivalent to the restricted form of the FORALL predicate that follows

```
FORALL ( T [ WHERE A ]
       [ GROUP BY B ] [ HAVING C ]
       FV <comparison-operator> BV )
```

where FV and BV are<value-expression>'s. In addition, one of FV or BV (let's say FV for purposes of discussion) must not contain references to any tuple variables that are bound in the tuple relational calculus formula implied by the FORALL predicate. The tuple variables implied by the correlation names in T are implicitly universally quantified and are thus bound in the formula. Thus, another way of stating the requirement for FV is that it not contain column references that are qualified by any correlation name in T. The equivalent SQL form using the ALL predicate is

```
FV <comparison-operator> ALL
     ( SELECT BV FROM T [ WHERE A ]
     [ GROUP BY B] [ HAVING C ] )
```

As an example consider the query, "Show any salesman who is younger than all other salesmen".

```
SELECT NAME FROM SALESMAN SM1
  WHERE
    FORALL (SALESMAN SM2
       WHERE SM2.SLM_ID != SM1.SLM_ID
       SM1.AGE < SM2.AGE)
```

Using the equivalence rule just defined this is the same as

```
SELECT NAME FROM SALESMAN SM1
  WHERE SM1.AGE < ALL (SELECT SM2.AGE
     FROM SALESMAN SM2
     WHERE SM2.ID != SM1.ID)
```

11. Null Values

In describing the equivalence between a FORALL predicate and a NOT EXISTS predicate, a 2-valued logic was assumed. However, SQL actually supports 3-valued logic in which a predicate evaluates to either true, false, or NULL, which is the same as unknown. If we assume that the FORALL predicate fully supports 3-valued logic, the equivalence demonstrated above between a FORALL predicate and a NOT EXISTS predicate does not always hold. For example, consider the query, "Show customers that did not buy any red products". The SQL for this query follows naturally from the reworded form, "Show customers such that for all orders made by the customer, the product was not red".

```
SELECT NAME FROM CUSTOMERS CU
WHERE
  FORALL (PRODUCTS PR, ORDERS ORD
     WHERE PR.PROD_ID = ORD.PROD_ID AND
       ORD.CUST_ID = CU.CUST_ID
     PR.COLOR != 'RED')
```

For the sample tables, only customer JACOB is returned by this query, despite the fact that customer JOHN also appears to have bought no red products. In order for JOHN to be selected, the predicate, "PR.COLOR != 'RED'", must evaluate to true for all rows in the join of ORDERS and PRODUCTS where the value of ORD.CUST_ID indicates that the customer is JOHN. The

predicate certainly evaluates to true for the ORDERS rows where the product is P1 or P4, but in the case of ORDER# O6, where the product is P5 and the color is NULL, the predicate evaluates to NULL. (i.e. The predicate "NULL != 'RED'" evaluates to NULL or unknown). Consequently, the FORALL predicate also evaluates to NULL and the CUSTOMERS row for JOHN is not returned. Coincidentally, in a 2-valued logic the FORALL predicate evaluates to false rather than NULL, but the result is the same. JOHN is not returned.

Now let's examine the ANSI SQL query that results from the transformation procedure described above.

```
SELECT NAME FROM CUSTOMERS CU
WHERE
  NOT EXISTS (SELECT 1
     FROM PRODUCTS PR, ORDERS ORD
     WHERE PR.PROD_ID = ORD.PROD_ID AND
       ORD.CUST_ID = CU.CUST_ID AND
       PR.COLOR = 'RED')
```

Unlike the FORALL version of the query, this query returns JOHN, along with JACOB,as a result. It does this despite the fact that the predicate "PR.COLOR = 'RED'" evaluates to NULL or unknown for product P5. As explained in a recent paper by Date [DATE89], the EXISTS predicate only evaluates to either true or false and in the case where it should really evaluate to unknown, as in this example, it evaluates to false. Consequently, the NOT converts the false to true and the entire predicate evaluates to true, causing CUSTOMER "JOHN" to be returned. If the EXISTS did support 3-valued logic,in this example it would evaluate to unknown rather than false. Since NOT unknown evaluates to unknown, JOHN would not be returned by the query.

It was stated earlier that in this particular example the FORALL query would return the same result even if like the EXISTS predicate it were incapable of evaluating to unknown. An example of a query where the result depends upon whether or not 3-valued logic is fully supported is"Show customers such that it is not true that all products ordered by the customer are red". Such a query should return any customer who ordered a non-red product. The SQL using the FORALL predicate is

```
SELECT NAME FROM CUSTOMERS CU
WHERE
  NOT FORALL (PRODUCTS PR, ORDERS ORD
     WHERE PR.PROD_ID = ORD.PROD_ID AND
       ORD.CUST_ID = CU.CUST_ID
     PR.COLOR = 'RED')
```

If FORALL, like EXISTS,could only evaluate to true or false, the query would return customers LISA, JOHN, PAUL, and JACOB. It is clear that LISA, JOHN and JACOB should be returned.In the case of PAUL, the predicate "PR.COLOR = 'RED'" evaluates to unknown or NULL for product P5 whose color is NULL. Since unknown is not true, the FORALL predicate evaluates to false, causing PAUL to be returned.If, on the other hand, FORALL could evaluate to unknown,then only LISA, JOHN and JACOB would be returned. The reason for this is that in the case of PAUL, the predicate "PR.COLOR = 'RED'" evaluates to unknown. Consequently, the FORALL predicate evaluates to unknown. NOT unknown also evaluates to unknown and PAUL is not returned. Following Date's logic [DATE89], the latter is the correct result. The essence of his argument is that all predicates in ANSI SQL are capable of evaluating to unknown. So why should there be an exception for the EXISTS predicate. The same argument applies to the FORALL predicate.

Incidentally, the SQL constructed by using the transformation procedure described above is

```
SELECT NAME FROM CUSTOMERS CU
WHERE
  EXISTS (SELECT 1
     FROM PRODUCTS PR, ORDERS ORD
     WHERE PR.PROD_ID = ORD.PROD_ID AND
       ORD.CUST_ID = CU.CUST_ID AND
       PR.COLOR != 'RED')
```

If it is assumed that FORALL supports 3-valued logic, this query is semantically identical to the source FORALL query. It returns LISA, JOHN and JACOB, but not PAUL. Thus, in this case the transformation procedure yields a correct result.

The above discussion makes it clear that in the presence of NULL's, the previously stated equivalence between a FORALL predicate and a NOT EXISTS predicate does not always hold. This is true regardless of whether or not the semantics of the FORALL predicate are defined to support 3-valued logic. The only way the semantics of the FORALL predicate and the a corresponding NOT EXISTS predicate would be guaranteed to match, is if both the EXISTS predicate of ANSI SQL and the proposed FORALL predicate supported 3 valued logic. This can be verified by reexamining the examples presented above.

12. ALL Predicate and Nulls

Unlike the EXISTS predicate the <comparison> ALL predicate does support 3 valued logic. Although this implies that if the FORALL predicate supported a 3-valued logic, the above stated equivalence between a FORALL predicate and a <comparison> ALL predicate is valid, it turns out not to be. To illustrate let's revisit a prior example, which happens to be of the form that is translatable directly into an ALL predicate.

```
SELECT NAME FROM CUSTOMERS CU
WHERE
  FORALL (PRODUCTS PR, ORDERS ORD
     WHERE PR.PROD_ID = ORD.PROD_ID AND
       ORD.CUST_ID = CU.CUST_ID
     PR.COLOR != 'RED')
```

The equivalent ALL predicate form is

```
SELECT NAME FROM CUSTOMERS CU
WHERE
  'RED' != ALL (SELECT PR.COLOR
     FROM PRODUCTS PR, ORDERS ORD
     WHERE PR.PROD_ID = ORD.PROD_ID AND
       ORD.CUST_ID = CU.CUST_ID)
```

If we assume that FORALL supports 3-valued logic, both queries return the same result. JACOB is returned.

Now let's reexamine the example assuming the presence of the following additional row in the ORDERS table.

ORD#	DATE	MON	CUST_ID	PROD_ID	SLM_ID	AMT	COM
O11	8/14/89	8		P2	S4	23	3

This time the FORALL form of the query returns no rows, while the ALL form still returns JACOB. The FORALL form returns no rows because when the CUSTOMERS row for JACOB is processed in the outer query block, the predicate, "ORD.CUST_ID = CU.CUST_ID", evaluates to NULL for the O11 ORDERS row. Consequently, the entire WHERE predicate evaluates to unknown. At the same time the predicate, "PR.COLOR != 'RED'", evaluates to false. This is the same as "unknown implies false", which evaluates to unknown. Thus, the FORALL predicate evaluates to unknown and JACOB is not returned.

The ALL predicate returns JACOB, despite the fact that the ORD.CUST_ID = CU.CUST_ID evaluates to unknown for the ORDERS row with the NULL CUST_ID value. If the unknown CUST_ID value happens to be C4 which is the customer corresponding to JACOB, the subquery would return RED as a value, and JACOB would not be returned as a result. Thus, the ALL predicate should evaluate to unknown for customer JACOB. This illustrates that even though the ANSI ALL predicate is capable of evaluating to unknown, there are circumstances when it evaluates to either true or false when it should really evaluate to unknown. For this reason, its semantics are not identical to those of the FORALL predicate.

There is a modified semantics of the FORALL predicate that would make the previously stated equivalence between it and the ALL predicate valid. The modified semantics require that the WHERE and HAVING clauses of the FORALL predicate support a 2-valued logic rather than a 3-valued logic. The <forall-search-condition>, however,would still support a 3-valued logic. In our example, the ORDERS row with a NULL CUST_ID would cause the WHERE clause to evaluate to false rather than unknown.Consequently, the FORALL clause would evaluate to true causing JACOB to be returned as a result.

13. Conclusion

An extension to ANSI SQL has been presented to provide more general support for universal quantification than is currently provided by the ALL predicate. The main benefit of the extension is to allow universal quantification queries to be expressed more naturally than the current syntax allows.

Using the law that states the equivalence between universal quantification and negated existential quantification, a procedure for transforming a query expressed in the new proposed universal quantification syntax into an ANSI standard equivalent form was derived. The extension and the transformation procedure are useful as aides in formulating quantification queries.A user may formulate a query using the proposed syntax and then mechanically transform it into the equivalent less intuitive ANSI syntax using the transformation procedure.

It was then shown that because the EXISTS predicate of SQL does not support 3-valued logic, the SQL generated from the transformation procedure may not be semantically equivalent to the original FORALL form. This is true regardless of whether or not the FORALL predicate semantics is defined to support 3 valued logic. Although this is a problem, the ANSI SQL generated by the transformation procedure is probably what the user would construct in the absence of a FORALL predicate. Thus, the FORALL predicate does not make matters worse. Of course, the real solution is to change the semantics of the EXISTS predicate to support 3-valued logic.

Unlike the ANSI EXISTS predicate the ANSI ALL predicate does support 3 valued logic. However, it was demonstrated that the support for 3-valued logic in the ALL predicate is not quite complete. For this reason, there are circumstances where the semantics of the ALL predicate differ from those of the FORALL predicate.

A modified semantics for the FORALL predicate was then introduced that provided for only partial support for 3-valued logic. It was shown that the modified semantics of a restricted form of the FORALL predicate are identical to those of the ANSI ALL predicate.

Acknowledgements

I thank Tim Berney and Al Gobel for a thorough review of this article and many useful suggestions.

References

[ANSI89] ANSI. Publication No. X3.135-1989, Database Language SQL.

[CHAM76] Chamberlain, D.D.,et al., "SEQUEL 2: a unified approach to data definition, manipulation, and control," IBM J. Research and Development 20:6, 1976, pp. 560-575.

[CODD88] Codd, E.F., "Fatal Flaws in SQL", Datamation, Aug. 15, 1988, pp. 45-48.

[DATE89] Date, C.J., "Be Careful Exists", Database Programming and Design, Sept. 1989, pp. 50-52.

[MAIE83] Maier, D., "The Theory of Relational Databases", Computer Science Press, Rockville, Maryland, 1983.

[ULLM88] Ullman, J. D., "Principles of Database and Knowledge Base Systems, Volume 1", Computer Science Press, Rockville, Maryland, 1988.