Teaching Tip

A Simpler Approach to Set Comparison Queries in SQL

Mohammad Dadashzadeh
Decision and Information Sciences
Oakland University
Rochester, MI 48309, USA
dadashza@oakland.edu

ABSTRACT

The current specification of the SQL standard fails to support users adequately in formulating complex queries involving set comparison that tend to arise in on-line analytical processing (OLAP) situations. Such queries must be formulated using correlated subqueries and the NOT EXISTS function which present an overwhelming challenge to both casual as well as everyday SQL users. This paper presents a simpler approach for teaching users how to formulate in SQL complex set comparison queries encountered in ad-hoc decision making scenarios.

Keywords: Database Management Systems, On-Line Analytical Processing, SQL, Set Comparison, Relational Algebra, Division Operator, Generalized Division, Human Factors

1. INTRODUCTION

One of the most important promises of the relational data model has been that it frees the decision maker, the manager, from the necessity of resorting to an intermediary, the programmer, in retrieving information from the organization's database in response to unanticipated needs. That promise is founded on the availability of very high-level relational query languages such as SQL. Unfortunately, the current specification of the SQL standard fails to support users adequately in formulating complex queries involving set comparison that tend to arise in on-line analytical processing (OLAP) situations.

Consider the following relational database about suppliers, parts, and jobs. (The primary key of each relation is underlined.)
SUPPLIER(<u>S#</u>, Supplier-Name, Supplier-City)
PART(<u>P#</u>, Part-Name, Part-Color)
JOB(<u>J#</u>, Job-Description, Job-City)
SHIPMENT(<u>S#</u>, <u>J#</u>, <u>P#</u>, QTY)
SUPPLY(<u>S#</u>, <u>P#</u>)

The relation SHIPMENT records information on what parts are *currently* shipped by each supplier to each job, while the relation SUPPLY indicates what parts can be supplied, *in the future*, by each supplier.

Now, consider the following queries:
Q1: Which suppliers are shipping *at least one* red part?
Q2: Which suppliers are shipping *every* red part?
Q3: Which suppliers are shipping *only* red parts?
Q4: Which suppliers will be able to supply *all* the parts that they are currently shipping?
Q5: Which suppliers are shipping *exactly* the same parts as supplier S1?

Of the queries listed, Q2-Q5 are considered **set comparison queries** since their result sets (i.e., the desired supplier numbers) can only be determined by comparing two sets (e.g., the set of part numbers shipped by each supplier against the set of part numbers for red parts). In contrast, the result set for Q1 can be obtained by merely matching (i.e., joining) the part number from a SHIPMENT row with that of a "red" PART row as shown below:
Q1: Which suppliers are shipping *at least one* red part?

```
SELECT DISTINCT S#
FROM   SHIPMENT, PART
WHERE  (SHIPMENT.P# = PART.P#) AND
       (Part-Color = 'RED');
```

Despite their simple appearances, queries involving set comparison are very difficult to formulate in relational query languages (Blanning 1993; Dadashzadeh 1992; Dadashzadeh 2001; Matos 2002; Rao 1996). In SQL, such queries must be specified using the complex and error-prone EXISTS function. In relational algebra, the algebraic operation of division used for this purpose is difficult for most users to comprehend and work with, and is incapable of expressing queries (such as Q4) that demand the comparison of sets of values associated with matching groups of rows in two tables. To fix ideas, consider the following formulation for Q2:

Q2: Which suppliers are shipping *every* red part?

```
SELECT DISTINCT S#
FROM   SHIPMENT X
WHERE  NOT EXISTS
       (SELECT*
       FROM   PART
       WHERE  Part-Color = 'RED'
              AND
              P# NOT IN
              (SELECT      P#
              FROM         SHIPMENT
              WHERE        S# = X.S#));
```

The use of double negation (NOT EXISTS and NOT IN) combined with a correlated subquery proves to be especially troublesome in teaching students how to formulate set comparison queries in SQL. Matos and Grasser (2002) have presented an alternative solution that is more intuitive and easier to deliver in the classroom. Their solution addresses only set comparison queries such as Q2 that can be expressed in relational algebra using the division operator. In this paper, we extend their approach to set comparison queries such as Q3-Q5 that must be expressed in relational algebra using the Generalized Division operator (Dadashzadeh 1989).

2. A GENERALIZED APPROACH TO SET COMPARISON QUERIES IN SQL

A general set comparison query can be modeled in the following ***intermediate*** SQL-like representation:

```
SELECT       desired-columns
FROM         desired-table(s)
WHERE        ( desired-non-set-comparisons )
GROUP BY     desired-columns
HAVING       SET( desired source set of values )
             set-comparison-operator
             ( target set of values subquery );
```

where, (*target set of values subquery*) may or may not be correlated.

For example, consider the following ***intermediate*** representation:

```
SELECT       S#, Supplier-Name
FROM         Supplier X, SHIPMENT
WHERE        (Supplier-City = 'LONDON') AND
             (X.S# = SHIPMENT.S#)
GROUP BY     S#, Supplier-Name
HAVING       SET( P# )
             CONTAINS
             (SELECT      P#
             FROM         PART
             WHERE        Part-Color =
                          'RED')
```

This query is intended to list S# and Supplier-Name for those suppliers located in London whose set of part shipments contains *every* red part. Here, the following correspondence with the general template can be established:

```
desired-columns:
        S#, Supplier-Name
Desired-table(s):
        Supplier X, SHIPMENT
( desired-non-set-comparisons ):
        (Supplier-City = 'LONDON')
( desired source set of values ):
        SET( P# )
Set-comparison-operator:
        CONTAINS
```

(target set of values subquery) *non-correlated subquery:*

```
        (SELECT      P#
        FROM         PART
        WHERE        Part-Color = 'RED')
```

Converting the above ***intermediate*** SQL-like representation to standard SQL is guided by the following theorem:

Theorem 1. Set A *CONTAINS* set B if | A ∩ B | = | B |.

In other words, set A *CONTAINS* set B if after restricting set A to elements that are also in set B, the number of elements (i.e., cardinality) in the *restricted* set A is identical to the number of elements in set B. Applying this observation to the above intermediate representation, we obtain the following standard SQL formulation for the query:

```
SELECT       S#, Supplier-Name
FROM         Supplier X, SHIPMENT
WHERE        (Supplier-City = 'LONDON') AND
             (X.S# = SHIPMENT.S#) AND
             (P# IN
                  (SELECTP#
                  FROM   PART
                  WHERE  Part-Color =
                         'RED'))
GROUP BY     S#, Supplier-Name
HAVING       COUNT( DISTINCT  P# )
             =
             (SELECTCOUNT( DISTINCT P# )
             FROM   PART
             WHERE  Part-Color = 'RED');
```

The following theorems help establish a similar approach for translating set comparison queries in the ***intermediate*** SQL-like representation to standard SQL when the set comparison operator is IN and is EQUAL TO:

Theorem 2. Set A *IS IN* set B if | A | = | B ∩ A |.

Theorem 3. Set A *IS EQUAL TO* set B if | A ∩ B | = | B | as well as | A | = | B ∩ A |.

For example, using the above generalized approach, Q3 is first represented in the ***intermediate*** representation by:

Q3 Intermediate Representation: Which suppliers are shipping *only* red parts?

```
SELECT      DISTINCT S#
FROM        SHIPMENT
GROUP BY    S#
HAVING      SET( P# )
            IS IN
            (SELECT     P#
            FROM        PART
            WHERE       Part-Color =
                        'RED')
```

And, using the transformation implied by Theorem 2, it can then be converted to standard SQL as:

Q3 Standard SQL: Which suppliers are shipping *only* red parts?

```
SELECT      DISTINCT S#
FROM        SHIPMENT X
GROUP BY    S#
HAVING      COUNT( DISTINCT P# )
            =
            (SELECTCOUNT( DISTINCT P# )
            FROM    PART
            WHERE   (Part-Color = 'RED') AND
                    (P# IN
                    (SELECTP#
                    FROM    SHIPMENT
                    WHERE   S# = X.S#));
```

3. CONCLUDING REMARKS

SQL does not provide direct support for comparing two sets. In fact, standard SQL does not provide operators to perform set intersection or set difference operations where it is required to compare two union-compatible tables for rows that are common to both or that are in one and not in the other. In order to formulate set intersection or set difference operations, the SQL user is expected to construct a query using two of the more difficult concepts in SQL: correlated subquery and the EXISTS function.

The complexity in formulating set difference and set intersection operations in SQL becomes much more pronounced when dealing with queries such as Q4 which involve set comparison for matching groups of rows in tables, rather than entire tables, and especially when considering set comparison operations such as equality or containment. As the SQL standard continues to become more widely used, and as end-users begin to rely on SQL for ad hoc database access, the difficulty in formulating set comparison queries in SQL is apt to become a common end-user complaint.

The undue complexity in formulating queries involving set comparison was avoided, to a large extent, in SEQUEL2 (Chamberlin 1976), the forerunner of SQL. In SEQUEL2, the EXISTS function is non-existent. Instead, SEQUEL2 provides explicit support for set comparison in two ways. First, SEQUEL2 provides direct support for set intersection and set difference operations in terms of INTERSECT and MINUS operations. Second, the built-in function SET in SEQUEL2 can be used in conjunction with the GROUP BY and HAVING operators to compare a set of values associated with a group of rows with the set of values derived from another table. The set comparison operators supported consist of: IS EQUAL TO; IS NOT EQUAL TO; CONTAINS; DOES NOT CONTAIN; IS IN; and IS NOT IN. In an unfortunate affront to human factor engineering, the current SQL standard expects the user to re-invent these set comparison operators using the complex and error-prone EXISTS function.

In this paper, we have presented a simpler approach to formulating set comparison queries in SQL that avoids the EXIST function. The approach is based on emulating SEQUEL2's built-in SET function and set comparison operators using the much more limited COUNT function. Matos and Grasser (2002) report positive results from human factor studies on such an approach that re-affirm earlier studies (Dadashzadeh 1993) indicating student preference for SEQUEL2's approach to set comparison queries. Nevertheless, the COUNT function is hardly a match for set comparison operators such as CONTAINS especially when one considers that a popular DBMS such as Microsoft Access dos not even support COUNT DISTINCT. As educators, our most appropriate response is to train users on when to use EXISTS and when not to use EXISTS. As IT professionals, however, our most appropriate recourse should be to press for re-examining the SQL standard for the possibility of introducing SEQUEL2's more user-friendly approach to set comparisons queries into SQL.

4. REFERENCES

Blanning, R. W. [1993]. "Relational Division in Information Management," Decision Support Systems, 9(4), pp. 313-324.

Chamberlin, D.D., et al. [1976]. "SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control," IBM Journal of Research & Development, 20(6), pp. 560-575.

Dadashzadeh, Mohammad [1989]. "An Improved Division Operator for Relational Algebra," Information Systems, 14(5), pp. 431-437.

Dadashzadeh, Mohammad [1992], "A Proposed Change to the SQL Standard." In Handbook of Systems Management: Development and Support, Edited by Paul C. Tinnirello, pp. 465-472. Auerbach Publications, Boston, MA.

Dadashzadeh, Mohammad [1993], "A Human Factor Study of Set Comparison Constructs in SQL.," TIMS/ORSA Joint National Meeting, May 16-19, 1993.

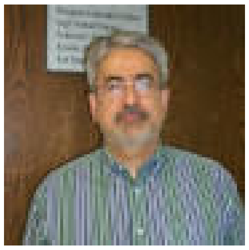
Dadashzadeh, Mohammad [2001], "Set Comparison Queries in SQL." In Developing Quality Complex Database Systems: Practices, Techniques, and

Technologies, Edited by Shirley Becker, pp. 303-316. Idea Group Publishing, Harrisburg, PA.

Matos, Victor M. and Rebecca Grasser [2002], "A Simpler (and Better) SQL Approach to Relational Division," Journal of Information Systems Education, 13(2), pp. 85-87.

Rao, S. G., A. Badia, and D. Van Gucht [1996], "Providing Better Support for a Class of Decision Support Queries." In Proceedings of the 1996 SIGMOD International Conference on Management of Data, pp. 217-227. Association for Computing Machinery, New York, NY.

AUTHOR BIOGRAPHY



Mohammad Dadashzadeh holds a bachelor in electrical engineering, a master in computer science, both from MIT, an MBA, and a Ph.D. in computer and information science from University of Massachusetts. He has been affiliated with University of Detroit (1984-1989) and Wichita State University (1989-2003) where he served as the W. Frank Barton Endowed Chair in MIS. He is now serving as Professor of MIS and Director of the Applied Technology in Business (ATiB) Program at Oakland University. Dadashzadeh has authored 4 books and more than 40 articles on information systems and has served as the editor-in-chief of *Journal of Database Management*.

ISSN 1055-3096