Converting Paradox’s QBE Set Queries Into Access 2000 SQL
Mohammad Dadashzadeh, (Email: mdz123@yahoo.com), Wichita State University

Abstract

One of the most important promises of the move to an SQL-based accounting software package has been that it frees the accountant from the necessity of resorting to a programmer when retrieving information from the organization’s database in response to unanticipated managerial needs. That promise is founded, in part, on the availability of a very high-level, visual relational query language interface known as Query By Example (QBE). Unfortunately, the implementation of QBE in Microsoft Access 2000 fails to support users in formulating complex queries involving set comparison that tend to arise in on-line analytical processing (OLAP) situations. And, while Paradox’s implementation of QBE makes the formulation of such queries quite intuitive, its built-in SQL translation feature fails to provide a clue on how to convert such queries into SQL. This paper presents a systematic approach based on formulating complex set queries in Paradox’s richer QBE notation and translating them into SQL queries that can be handled by Access 2000.

Introduction

Consider the following relational database about suppliers, parts, and jobs. (The primary key of each relation is underlined.)

SUPPLIER( S#, SName, Status, City )
PART( P#, PName, Color, City )
JOB( J#, JName, City )
SHIPMENT( S#, P#, J#, QTY )

The relation SHIPMENT records the quantity of each part being shipped by each supplier to various jobs. An instance of this database is depicted below.

![Database Instance](image)

Readers with comments or questions are encouraged to contact the authors via email.
### Table: Part.db

<table>
<thead>
<tr>
<th>P#</th>
<th>PNAME</th>
<th>COLOR</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P1</td>
<td>Nut</td>
<td>Red</td>
</tr>
<tr>
<td>2</td>
<td>P2</td>
<td>Bolt</td>
<td>Green</td>
</tr>
<tr>
<td>3</td>
<td>P3</td>
<td>Screw</td>
<td>Blue</td>
</tr>
<tr>
<td>4</td>
<td>P4</td>
<td>Screw</td>
<td>Red</td>
</tr>
<tr>
<td>5</td>
<td>P5</td>
<td>Cam</td>
<td>Blue</td>
</tr>
<tr>
<td>6</td>
<td>P6</td>
<td>Cog</td>
<td>Red</td>
</tr>
</tbody>
</table>

### Table: Job.db

<table>
<thead>
<tr>
<th>J#</th>
<th>JNAME</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>J1</td>
<td>Sorter</td>
</tr>
<tr>
<td>2</td>
<td>J2</td>
<td>Punch</td>
</tr>
<tr>
<td>3</td>
<td>J3</td>
<td>Reader</td>
</tr>
<tr>
<td>4</td>
<td>J4</td>
<td>Console</td>
</tr>
<tr>
<td>5</td>
<td>J5</td>
<td>Collator</td>
</tr>
<tr>
<td>6</td>
<td>J6</td>
<td>Terminal</td>
</tr>
<tr>
<td>7</td>
<td>J7</td>
<td>Tape</td>
</tr>
</tbody>
</table>
Now, consider the following queries:

Q1: List the suppliers who ship every red part. (Answer: S5)
Q2: List the suppliers who do not ship to any job located in London. (Answer: S1 and S3)
Q3: List the jobs that are only receiving parts warehoused in London. (Answer: None)
Q4: List the suppliers who are shipping to exactly the same jobs as supplier S1. (Answer: None)

Each of the above queries involves comparison of sets of values in two tables. For example, in Q1, the set of parts (P# values) associated with each supplier (distinct S# value) in the SHIPMENT table must be examined to determine if it contains the set of parts (P# values) in the PART table sharing the value of "Red" for the COLOR attribute.

Despite their innocuous appearances, queries involving set comparison are especially difficult to formulate in relational query languages (Blanning, 1993; Celko, 1997; Dadashzadeh, 2001). Specifically, in SQL such queries must be specified using the complex and error-prone NOT EXISTS function that, for most users, is difficult to comprehend.
and work with.

In contrast, Paradox's QBE provides special set operators (SET, EVERY, NO, ONLY, and EXACTLY) that directly support the formulation of such queries as illustrated below:

**Q1 in Paradox's QBE:** List the suppliers who ship every red part.

![Query: SuppliersShippingEveryRedPart.qbe](image1)

In this QBE formulation, Paradox’s SET operator is used to define a set named XYZ as consisting of the P# of all red parts in the PART table. Then, Paradox’s set comparison operator EVERY is used to indicate that from the SHIPMENT table only those S# values should be printed out that appear with EVERY value in the set XYZ.

**Q2 in Paradox's QBE:** List the suppliers who do not ship to any job located in London.

![Query: SuppliersNotShippingToAnyLondonJob.qbe](image2)

**Q3 in Paradox's QBE:** List the jobs that are only receiving parts warehoused in London.

![Query: JobsReceivingOnlyPartsWarehousedInLondon.qbe](image3)
Q4 in Paradox’s QBE: List the suppliers who are shipping to exactly the same jobs as supplier S1.

Here, Paradox’s SET operator is used to define a set named XYZ as consisting of the J# of all jobs receiving a shipment from supplier S1. Then, Paradox’s set comparison operator EXACTLY is used to indicate that from the SHIPMENT table only those S# values (different than S1) should be printed out that appear with EXACTLY the values found in the set XYZ.

The clarity afforded by the use of set operators in Paradox’s QBE is unfortunately absent in Microsoft Access’ implementation of QBE. Therefore, such set comparison queries must necessarily be formulated in Access using SQL. And, even though, Paradox normally does offer to translate the QBE query into SQL, this feature is not available for set comparison queries resulting in the disappointing message shown below:

In this paper, we provide the foundation for a solution to this shortcoming in the form of an algorithm for converting Paradox’s QBE set queries into standard SQL, thus paving the way for much easier formulation of set comparison queries in Microsoft Access.

A Guided Tour of the Conversion Algorithm

We illustrate the algorithm by converting the Q1 query reproduced below.

Q1 in Paradox’s QBE: List the suppliers who ship every red part.
The algorithm consists of two steps. In the first step, the QBE set query is translated to an intermediate SQL-like representation. In the second step, the intermediate SQL-like representation is transformed to the final equivalent standard SQL representation.

The template for the intermediate SQL-like representation of Paradox’s QBE set queries is:

```sql
SELECT source-table-checked-columns
FROM source-table
WHERE source-table-selection-condition
GROUP BY source-table-checked-columns
HAVING SET( source-table-example-element-column )
    set-comparison-operator
    (SELECT set-table-example-element-column
    FROM set-table
    WHERE set-table-selection-condition);
```

where source-table refers to the database table with the QBE set operator (i.e., EVERY, NO, ONLY, or EXACTLY), set-table denotes the database table with the QBE SET operator applied to it, and set-comparison-operator is either CONTAINS (for EVERY), DISJOINT FROM (for NO), CONTAINED IN (for ONLY), or EQUALS (for EXACTLY).

Applying this template to our example query Q1 we arrive at the following intermediate representation:

```sql
SELECT S#
FROM SHIPMENT
GROUP BY S#
HAVING SET( P# )
    CONTAINS
    (SELECT P#
    FROM PART
    WHERE COLOR = "Red");
```

Note that since the rows of the SHIPMENT table are not subject to any selection condition in the QBE query, there is no WHERE clause associated with the outer SELECT statement.

Figures 1-3 depict, respectively, the intermediate representation of queries Q2, Q3, and Q4.

Figure 1.
Intermediate Representation of Q2 (suppliers who do not ship to any job located in London).

```sql
SELECT S#
FROM SHIPMENT
GROUP BY S#
HAVING SET( J# )
    DISJOINT FROM
    (SELECT J#
    FROM JOB
    WHERE CITY = "London");
```
The second step in the algorithm is based on a series of transformation rules depicted in Figures 4-8. Specifically, given an SQL-like query in the format shown in Figure 4, Figures 5-8 give the equivalent standard SQL representations when the set-comparison-operator is, respectively, CONTAINS, DISJOINT FROM, CONTAINED IN, and EQUALS.

Applying the transformation rule from Figure 5 to the intermediate representation of our example query Q1 we get the final equivalent SQL representation:

```sql
SELECT DISTINCT X.S#
FROM SHIPMENT X
WHERE NOT EXISTS
  (SELECT * FROM PART
   WHERE (COLOR = "Red")
   AND P# NOT IN
     (SELECT P# FROM SHIPMENT
      WHERE S# = X.S#));
```

where X is the chosen alias for the outer SHIPMENT table.

The following figures present the above query in Paradox’s SQL Editor and Access 2000 SQL View where column names utilizing special characters such as # symbol must be enclosed, respectively, in quotation marks and square brackets.
Figures 9-11 depict, respectively, the final SQL representation of queries Q2, Q3, and Q4, derived by applying the appropriate transformation rules to the intermediate representation of these queries given in Figures 1-3.
Figure 4.
The General Form of the Intermediate SQL-Like Representation.

```
SELECT grouping-columns
FROM source-table
WHERE source-table-selection-condition
GROUP BY grouping-columns
HAVING SET( set-column )
    set-comparison-operator
    (SELECT set-column
     FROM set-table
     WHERE set-table-selection-condition);
```

Figure 5.
The Equivalent Standard SQL Representation of Figure 4 when set-comparison-operator is CONTAINS.

```
SELECT DISTINCT grouping-columns
FROM source-table ALIAS
WHERE (source-table-selection-condition)
    AND NOT EXISTS
    (SELECT *
     FROM set-table
     WHERE (set-table-selection-condition)
             AND set-column NOT IN
             (SELECT set-column
              FROM source-table
              WHERE (source-table-selection-condition)
                  AND
                  grouping-columns
                  = ALIAS.grouping-columns));
```

Figure 6.
The Equivalent Standard SQL Representation of Figure 4 when set-comparison-operator is DISJOINT FROM.

```
SELECT DISTINCT grouping-columns
FROM source-table ALIAS
WHERE (source-table-selection-condition)
    AND NOT EXISTS
    (SELECT *
     FROM set-table
     WHERE (set-table-selection-condition)
             AND set-column IN
             (SELECT set-column
              FROM source-table
              WHERE (source-table-selection-condition)
                  AND
                  grouping-columns
                  = ALIAS.grouping-columns));
```
Figure 7.
The Equivalent Standard SQL Representation of Figure 4 when set-comparison-operator is CONTAINED IN.

```
SELECT DISTINCT grouping-columns
FROM source-table ALIAS
WHERE (source-table-selection-condition)
AND NOT EXISTS
  (SELECT *
   FROM source-table
   WHERE (source-table-selection-condition)
   AND (grouping-columns = ALIAS.grouping-columns)
   AND set-column NOT IN
  (SELECT set-column
   FROM set-table
   WHERE set-table-selection-condition));
```

Figure 8.
The Equivalent Standard SQL Representation of Figure 4 when set-comparison-operator is EQUALS.

```
SELECT DISTINCT grouping-columns
FROM source-table ALIAS
WHERE (source-table-selection-condition)
AND NOT EXISTS
  (SELECT *
   FROM set-table
   WHERE (set-table-selection-condition)
   AND set-column NOT IN
  (SELECT set-column
   FROM source-table
   WHERE (source-table-selection-condition)
   AND grouping-columns
   = ALIAS.grouping-columns))

AND NOT EXISTS
  (SELECT *
   FROM source-table
   WHERE (source-table-selection-condition)
   AND grouping-columns = ALIAS.grouping-columns
   AND set-column NOT IN
  (SELECT set-column
   FROM set-table
   WHERE set-table-selection-condition));
```
Figure 9.
The Equivalent Standard SQL Representation of Figure 1 (Q2: suppliers who do not ship to any job located in London).

```sql
SELECT DISTINCT S#
FROM SHIPMENT X
WHERE NOT EXISTS
  (SELECT *
   FROM JOB
   WHERE (CITY = "London")
   AND J# IN
     (SELECT J#
      FROM SHIPMENT
      WHERE S# = X.S#));
```

Figure 10.
The Equivalent Standard SQL Representation of Figure 2 (Q3: jobs that are only receiving parts warehoused in London).

```sql
SELECT DISTINCT J#
FROM SHIPMENT X
WHERE NOT EXISTS
  (SELECT *
   FROM SHIPMENT
   WHERE (J# = X.J#)
   AND P# NOT IN
     (SELECT P#
      FROM PART
      WHERE CITY = "London"));
```

Figure 11.
The Equivalent Standard SQL Representation of Figure 3 (Q4: suppliers who are shipping to exactly the same jobs as supplier S1).

```sql
SELECT DISTINCT S#
FROM SHIPMENT X
WHERE (S# <> "S1")
AND NOT EXISTS
  (SELECT *
   FROM SHIPMENT
   WHERE (S# = "S1")
   AND J# NOT IN
     (SELECT J#
      FROM SHIPMENT
      WHERE (S# <> "S1")
      AND S# = X.S#))
AND NOT EXISTS
  (SELECT *
   FROM SHIPMENT
   WHERE (S# <> "S1")
   AND J# = X.J#
   AND J# NOT IN
     (SELECT J#
      FROM SHIPMENT
      WHERE (S# <> "S1")
      AND S# = X.S#));
```

55
Summary

The evolutionary shift from stand-alone accounting software to collaborative, enterprise-wide business applications has irrevocably impacted the accounting profession. One facet that has become important as the value of integrated, DBMS-based applications has risen in modern organizations is the requisite skills of accounting professionals. Along with traditional business skills to interpret data and to know what information is critical in a decision-making scenario, as pointed out by Olsen (2000), “accountants should have considerable database knowledge as well as specific knowledge of the structured query language (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. As pointed out by Rao et al. (1996) “SQL’s syntax is too restricted to express quantified queries. While SQL allows subqueries to form sets, the relationships that can be expressed over sets are limited, and must be written in awkward and complicated ways.” On the other hand, Paradox’s implementation of QBE directly supports set operations making the formulation of set comparison queries quite intuitive. But, although Access 2000—the dominant end-user query/reporting tool—does support QBE, its implementation lacks the set operations of Paradox.

To overcome this shortcoming, this paper has presented an algorithm for converting Paradox’s QBE set queries into standard SQL. The principal contribution to the practicing accountant is learning a simple technique to write complex set comparison queries in any SQL-based system, including Access 2000, by starting with the intuitive Paradox QBE formulation.

References