SQL Subqueries: Counterparts in Tutorial D

Hugh Darwen

I present some notes in response to a question on how to transcribe SQL queries that use subqueries into **Tutorial D**. But first let me explain why I don't teach this in CS252. The explanation is quite simply that subqueries are not needed for relational completeness! Put another way, for every SQL query that uses a subquery in its WHERE clause there is an equivalent query that does not use any subqueries. Subqueries do sometimes allow for a more convenient or more intuitive way of expressing something but I am not too concerned with such psychological issues in teaching relational theory. It is more important for you to understand how every query can be expressed using just the relational operators and invocations of WHERE that do not include relational expressions on the right-hand side.

So you don't need to study these notes if you don't want to. But in case you do …

What is a subquery?

SQL uses the term *subquery* to refer to a table expression (commonly known as a query) that is contained within another table expression. Until 1992, as far as the international standard for SQL is concerned, subqueries were permitted only inside conditional expressions (i.e., truth-valued expressions), and conditional expressions were permitted only inside the WHERE clause, and the HAVING clause. Nowadays subqueries are permitted in the FROM and SELECT clauses too (and truth-valued expressions can appear in the SELECT clause, but we don't need to look at examples of those here).

Does Tutorial D support subqueries?

Tutorial D has no restrictions on what type of expression is permitted where, apart from the usual rule that the type of an expression must be appropriate to the context in which it appears. (E.g., both operands of "=" must be of the same type, and both operands of JOIN must be of some relation type.)

You can see examples of **Tutorial D** "subqueries" in some of my lecture slides—for example, Lecture 5, Slide 11 and Lecture 6, Slides 12-16 and 21. These are definitions of "shorthands"—operators like SUMMARIZE and DIVIDEBY that can be defined entirely in terms of existing operators. And of course every constraint declaration in **Tutorial D** (apart from KEY constraints) involves writing relational expressions inside truth-valued expressions such as relation comparisons or invocations of IS_EMPTY.

It follows that whatever can be expressed in SQL using a subquery can be expressed in a similar fashion in **Tutorial D**—if you really want to; often, though, that is not the best way of doing it in **Tutorial D** even if it is the best way in SQL, as we shall see. However, there are some important differences in style to be noted. Understanding these points will help you to make appropriate transcriptions.

SQL uses coercion; Tutorial D doesn't

The term *coercion* refers to the use of an expression *x* of some type *t1* in a place where an operand of a different type *t2* is expected. A suitable "type conversion" operator is implicitly invoked to derive from *x* some corresponding value of type *t2*. For example, some implementations of SQL allow you to assign a character string

such as '0001' to a numeric column. In this example, '0001' is implicitly "converted" to the number 1.

Now consider the following SQL query:

```
SELECT Emp#, Name
FROM   Emp
WHERE  Salary = ( SELECT MAX(Salary) FROM Emp )
```

That WHERE clause appears to be comparing a number, Salary, with a table! But because the table is guaranteed to consist of exactly one column and exactly one row (even when Emp is empty!), SQL is willing to save you the trouble of writing something to tell it to extract the single column value from that single row.

In general coercion is not a good idea and the use of it in SQL has led to severe difficulties in the development of the language since its introduction in 1979. So **Tutorial D** does not use it at all. Here's a faithful transcription into **Tutorial D:**

```
( Emp
  WHERE Salary = Salary FROM TUPLE FROM
                   ( SUMMARIZE Emp PER TABLE_DEE
                      ADD ( MAX(Salary) AS Salary ) ) )
{ Emp#, Name }
```

Alternatively, as the comparison uses "=" rather than "<" or ">", we can convert the scalar to a relation:

```
( Emp
  WHERE RELATION { TUPLE { Salary Salary } }
      = SUMMARIZE Emp PER TABLE_DEE
              ADD ( MAX(Salary) AS Salary ) )
{ Emp#, Name }
```

But exact transcription isn't always the easiest way of solving the same problem in a different language. Unlike SQL, **Tutorial D** has *aggregate operators* (Lecture 5, Slides 13-14) that take relations and yield scalar values. To find the employees with the highest salaries you would be more likely to simply write this:

```
( Emp
  WHERE Salary = MAX ( Emp, Salary ) )
{ Emp#, Name }
```

The second appearance of "Emp" here is what SQL would call a subquery. That's a rather heavy term for a simple relvar reference, but SQL does not allow a query to be a simple table reference, forcing you to write SELECT * FROM Emp instead.

Column references versus attribute references

In SQL, if table operands T1 and T2 each have a column named C, you can use "dot qualification" to tell the system which C you are referring to. This requires the table operands to be named. For example, if they are assigned the names T1 and T2 (in FROM clauses), then T1.C and T2.C can be used to distinguish between the two columns.

Tutorial D does not use dot qualification, because in SQL it sometimes leads to tables having more than one column of the same name. Such tables cannot be used everywhere where you would expect to be able to use a table (e.g., they cannot be

stored in the database!). In **Tutorial D**, therefore, you sometimes have to resort to the RENAME operator to avoid ambiguities. Consider, for example, the SQL query:

```
SELECT StudentId, Name
FROM   Student S
WHERE  NOT EXISTS ( SELECT *
                    FROM IsEnrolledOn E
                    WHERE S.StudentId = C.StudentId )
```

Here's as faithful a transcription into **Tutorial D** as I can come up with:

```
( Student WHERE IS_EMPTY ( ( IsEnrolledOn
                             RENAME ( StudentId AS S ) )
                           WHERE StudentId = S )
{ StudentId, Name }
```

In the invocation of WHERE in the subquery, StudentId refers to the attribute of the Student, permitted because Student is the relation operand of the WHERE in whose condition StudentId appears. Without the RENAME it would refer to the attribute of Enrolment and there would be no way of referring to StudentId of Student.

But of course you wouldn't really do it that way in **Tutorial D** because it's much easier using plain old relational algebra:

```
( Student NOT MATCHING IsEnrolledOn ) { StudentId, Name }
```

Truth-Valued Operators in SQL and Tutorial D

I've already shown that **Tutorial D** has IS_EMPTY (…) corresponding to SQL's NOT EXISTS (…). Obviously NOT (IS_EMPTY (…)) corresponds to SQL's EXISTS (…).

Scalar comparison operators in SQL and **Tutorial D** are pretty well identical, though "=" really does mean "is exactly the same as" in **Tutorial D** whereas in SQL it sometimes means "pretty well the same as, even if not exactly so". Also, much more importantly, all truth-valued operators in **Tutorial D** are guaranteed to yield either TRUE or FALSE when invoked, whereas SQL's (apart from EXISTS) sometimes yield something called UNKNOWN.

Both SQL and **Tutorial D** have an operator for membership testing. In SQL it is IN. In official **Tutorial D** it is ∈ but **Rel** allows you to spell it the same way as in SQL. Don't forget, though, that **Tutorial D**'s counterpart of SQL's

```
(x,y,z) IN ( SELECT a, b, c FROM t )
```

would be

```
TUPLE { a x, b y, c z } IN t { c, b, a }.
```

(I have deliberately written the attribute names in different orders in the two operands, to remind you that Tutorial D doesn't care about the order of the attributes whereas SQL *does* care about the order of columns.)

SQL has no counterpart of **Tutorial D**'s relation comparison operators (is subet of, is superset of, and "equals").

End of paper