Chapter 9

Correlated Subqueries

A correlated subquery is one in which:

(a) There is a subquery (a main, outer query and an inner query).

(b) The information in the inner subquery is referenced by the outer, main query such that the inner query may be thought of as being executed repeatedly. This point will be clarified by several examples.

In this chapter, we will study correlated subqueries. We will discuss existence queries (WHERE EXISTS) and correlation as well as NOT EXISTS. We will also take a look at SQL's universal and existential qualifiers. Before discussing correlated subqueries in detail, let's make sure we understand what a non-correlated subquery is.

9.1 Non-Correlated Subqueries

A non-correlated subquery is a subquery independent of the outer query. The subquery could be executed on its own. The following is an example of a non-correlated query:

```
SELECT        s.sname
FROM          Student s
WHERE         s.stno IN
  (SELECT     gr.student_number
  FROM        Grade_report gr
  WHERE       gr.grade = 'A');
```

*****Begin Note*****
The part of the query in parentheses is a subquery (also referred to as a *nested query* or *embedded query*). The subquery is an independent entity -- it would work by itself if run as a stand-alone query.
*****End Note*****

We have seen in earlier chapters that Oracle may rearrange queries to gain efficiency. Rearrangement aside, the subquery:

```
(SELECT       gr.student_number
FROM          Grade_report gr
WHERE         gr.grade = 'A');
```

can be *thought* of as being evaluated first, creating the set of student numbers who have A's. The subquery result set is then used to determine which rows in the main query will be SELECTed.

This query:

```
SELECT s.sname                          -- outer
FROM    Student s                       -- outer
WHERE  s.stno IN                        -- outer
    (SELECT gr.student_number           -- inner
    FROM   Grade_report gr              -- inner
    WHERE  gr.grade = 'A')              -- inner
;
```

generates:

```
SNAME
--------------------
Lineas
Mary
Brenda
Richard
Lujack
Donald
Lynette
Susan
Holly
Sadie
Jessica
Steve
Cedric
Jerry

14 rows selected.
```

9.2 Correlated Subqueries

In the beginning of the chapter, we stated that correlated subqueries are subqueries in which there is a subquery (an outer query and an inner subquery), ***and*** the information in the subquery is referenced by the outer query.

Correlated queries present a different execution scenario to the database manipulation language (DML) from ordinary, non-correlated subqueries. The correlated sub-query **cannot** stand alone as it depends on the outer query; completing the subquery prior to execution of the outer query is not an option. The efficiency of the correlated subquery varies; it may be worthwhile to test the efficiency of correlated queries versus joins or sets in production databases.

*****Begin Note*****
One situation in which you cannot avoid correlation is the "for all" query, which we will discuss later in this chapter.
*****End Note*****

The following is an example of a correlated query:


```
SELECT        s.sname
FROM          Student s
WHERE         s.stno IN
(SELECT       gr.student_number
 FROM         Grade_report gr
 WHERE        gr.student_number = s.stno    /*s.stno references outer query  */
 AND          gr.grade = 'B');
```

This produces the following output:

```
SNAME
-------------------
Lineas
Mary
Zelda
Ken
Mario
Brenda
Kelly
Lujack
Reva
Harley
Chris
Lynette
Hillary
Phoebe
Holly
Sadie
Jessica
Steve
Cedric
George
Cramer
Fraiser
Francis
Smithly
Sebastian
Lindsay
Stephanie

27 rows selected.
```

Here, the inner query references the outer one -- observe the use of *s.stno* in the WHERE clause of the inner query. Rather than thinking of this query as creating a set of student numbers with B's, each row from the outer query is SELECTed individually and tested against all rows of the inner query one at a time until it is determined whether a given student number is in the inner set and whether that student earned a B.

This execution scenario is like a nested DO loop in a programming language, where the first row from the **Student** table is SELECTed. Then each outher row is individually selected and tested against rows from the **Grade_report** table. Then the second row from the **Student** table is SELECTed. Then, each outer row is tested against rows from the **Grade_report** table. The following is the DO loop in pseudo-code:

```
LOOP1: For each row in Student s DO
```

```
 LOOP2: For each row in Grade_report gr DO
  IF (gr.student_number = s.stno) then
  IF (gr.grade = 'B') THEN TRUE
  END LOOP2;
 If TRUE, then Student row is SELECTed
END LOOP1;
```

This particular query could have been done without correlation in a manner similar to our first example in this chapter; however, it demonstrates the difference in query execution.

You might think correlated queries are less efficient than doing a simple, uncorrelated subquery because the uncorrelated subquery is done once and the correlated subquery is done once for each outer row. However, the internal handling of how the query executes depends on the SQL and the optimizer for a particular database engine. In Oracle, the database engine is designed so queries containing correlation are actually quite efficient.

9.3 Existence Queries and Correlation

Correlated subqueries are often written so the question in the inner query is one of existence. For example, assume we want to find the names of students who have taken a computer science (COSC) class and have earned a grade of B in that course. This query, like most queries, can be written in several ways. For example, we can use a non-correlated subquery as follows:

```
SELECT      s.sname
FROM        Student s
WHERE       s.stno IN
  (SELECT   gr.student_number
  FROM      Grade_report gr, Section
  WHERE     Section.section_id = gr.section_id   /* join condition Grade_report-Section */
  AND       Section.course_num LIKE 'COSC____'
  AND       gr.grade = 'B');
```

This query would produce the following output:

```
SNAME
------------------------
Holly
Lynette
Stephanie
Lindsay
Fraiser
Hillary
George
Lineas
Lujack
Cramer
Chris
Brenda
Mary
Francis
Phoebe
Reva
Harley

17 rows selected.
```

Since this query is non-correlated, we can think of it as first forming the set of student numbers of students who have earned B's in COSC courses -- the inner query result set. In the inner query, we must have both the **Grade_report** and the **Section** tables because the course numbers are in the **Section** table and the grades are in the **Grade_report** table. Once we form this set of student numbers (once we complete the inner query), the outer query looks through the **Student** table and SELECTs only those students who are in the inner query result set.

*****Begin Note*****
This query could also be done by creating a double-nested subquery containing two INs, or it could be written using a three-table join.
*****End Note*****

Had we chosen to write the query with an unnecessary correlation, it might look like this:

```
SELECT      s.sname
FROM        Student s
WHERE       s.stno IN
  (SELECT   gr.student_number
   FROM     Grade_report gr, Section
   WHERE    Section.section_id = gr.section_id  /* join condition Grade_report-Section */
   AND      Section.course_num LIKE 'COSC____'                /*correlation     */
   AND      gr.student_number = s.stno
   AND      gr.grade = 'B');
```

The final result of this query would be the same as the previous one. In this case, using the **Student** table in the subquery is unnecessary. Next, we will look at situations in which correlation is necessary, and, in particular, introduce a new predicate -- EXISTS.

9.3.1 EXISTS

As noted earlier, there will be situations in which the correlation of a subquery *is* necessary. Another way to write the correlated query is with the EXISTS predicate, which looks like this:

```
SELECT          s.sname
FROM            Student s
WHERE EXISTS
   (SELECT      1
   FROM         Grade_report gr, Section
   WHERE        Section.section_id = gr.section_id   /* join condition Grade_report-Section */
   AND          Section.course_num like 'COSC____'
   AND          gr.student_number = s.stno      /*  correlation         */
   AND          gr.grade = 'B');
```

This correlated query produces the same output (17 rows) as both of the previous queries. Let us dissect this version.

The EXISTS predicate says, "Choose the row from the **Student** table in the outer query if the subquery is TRUE" -- if a row in the subquery exists and satisfies the condition in the subquery WHERE clause. Since no actual result set for the inner query is formed, "SELECT 1" is used as a "dummy" result set to indicate the subquery is TRUE (1 is returned) or FALSE (no rows are returned). In the non-correlated case, we tied the student number in the **Student** table to the inner query by the IN predicate as follows:

```
SELECT      s.stno
FROM        Student s
WHERE       s.stno IN
   (SELECT "student number"...)
```

When using the EXISTS predicate, we use the **Student** table in the subquery (i.e., it's correlated). Hence, we are seeking only to find whether the subquery WHERE clause can be satisfied.

What is the "SELECT 1" doing in the subquery? Using the EXISTS predicate, the subquery does not form a result set *per se*, but rather returns TRUE or FALSE. SELECT * in the subquery may be used; however, from an "internal" standpoint, SELECT * causes the SQL engine to check the Data Dictionary unnecessarily. Because the actual result of the inner query is only TRUE or FALSE, it is suggested that SELECT 'X' (or SELECT 1) ... instead of SELECT * be used so a constant is SELECTed instead of some "sensible" entry. The SELECT 'X'… or (SELECT 1...) is simply more efficient.

The EXISTS predicate forces us to correlate the query. To illustrate where correlation is usually necessary with EXISTS, consider the following query:

```
SELECT        s.sname        /* exists-uncorrelated  */
FROM          Student s
WHERE EXISTS
              (SELECT       'X'
              FROM          Grade_report gr, Section t
              WHERE         t.section_id = gr.section_id        /* join  Grade_report-Section */
              AND           t.course_num like 'COSC____'
              AND           gr.grade = 'B');
```

This produces the following output:

```
SNAME
--------------------
Lineas
Mary
Brenda
Richard
Kelly
.
.
.
Romona
Ken
Smith
Jake

48 rows selected.
```

This query uses EXISTS, but has no correlation. This syntax infers that for each student row, we test the joined the **Grade_report** and **Section** tables to see whether there is a course number like COSC and a grade of B (which, of course, there is). We unnecessarily ask the subquery question over and over again. The result from this latter, uncorrelated EXISTS query is the same as:

```
SELECT          s.sname
FROM            Student s;
```

The point is the correlation is necessary when we use EXISTS.

Consider another example in which a correlation could be used. Suppose we want to find the names of all students who have three or more B's. A first pass at a query might be something like this:

```
SELECT          s.sname
FROM            Student s WHERE "something" IN
  (SELECT       "something"
  FROM          Grade_report
  WHERE         "count of grade = 'B'" > 2);
```

This query can be done with a HAVING clause as we saw previously, but we want to show how to do this in yet another way. Suppose we arrange the subquery to use the student number from the **Student** table as a filter and count in the subquery only when a row in the **Grade_report** table correlates to that particular student. The query looks like this:

```
SELECT          s.sname
FROM            Student s
WHERE 2 < (SELECT COUNT(*) -- EXISTS implied
                FROM      Grade_report gr
                WHERE     gr.student_number = s.stno
                AND       gr.grade = 'B');
```

This results in the following output:

```
SNAME
--------------------
Lineas
Mary
Lujack
Reva
Chris
Hillary
Phoebe
Holly

8 rows selected.
```

Although there is no EXISTS in the query, it is implied. The syntax of the query does not allow an EXISTS, but the sense of the query is "WHERE EXISTS a COUNT OF 2 WHICH IS LESS THAN…" In this correlated query, we must examine the **Grade_report** table for each member of the **Student** table to see whether the student has two B's (correlation). We test the entire **Grade_report** table for each student row in the outer query.

If it were possible, a subquery without the correlation would be more desirable. The overall query might start out like this:

```
SELECT          s.sname
FROM            Student s
WHERE           s.stno in ...
```

We might attempt to write the following subquery:

```
SELECT          s.sname
FROM            Student s
WHERE           s.stno IN
  (SELECT       gr.student_number
  FROM          Grade_report gr
  WHERE         gr.grade = 'B');
```

However, this would give us only students who had made at *least* one B, as seen in the following output:

```
SNAME
------------------------
Lineas
Mary
Zelda
Ken
Mario
Brenda
Kelly
Lujack
Reva
Harley
Chris
Lynette
Hillary
Phoebe
Holly
Sadie
Jessica
Steve
Cedric
George
Cramer
Fraiser
Francis
Smithly
Sebastian
Lindsay
Stephanie

27 rows selected.
```

To get students who have earned three B's, we could try the following query:

```
SELECT         s.sname
FROM           Student s
WHERE          s.stno IN -- this query will not work!
   (SELECT     gr.student_number, COUNT(*)
   FROM        Grade_report gr
   WHERE       gr.grade = 'B'
   GROUP BY    gr.student_number
   HAVING      COUNT(*) > 2);
```

However, this will not work because the subquery cannot have two attributes in its result set unless the main query has two attributes in the WHERE .. IN. Here, the subquery must have only *gr.student_number* to match *s.stno*. We might then construct an inline view as with the following query:

```
SELECT         s.sname
FROM           Student s
WHERE          s.stno IN
   (SELECT     student_number
   FROM        (SELECT student_number, COUNT(*)
   FROM        Grade_report gr
   WHERE       gr.grade = 'B'
   GROUP BY    student_number having COUNT(*) > 2));
```

This succeeds in Oracle but may fail in other versions of SQL. The output of this query would be:

```
SNAME
----------------------------
Holly
Hillary
Lineas
Lujack
Chris
Mary
Reva
Phoebe

8 rows selected.
```

As you can see, we can query the database using various methods with SQL. In this case, the correlated query may be the easiest to see and perhaps the most efficient.

9.3.2 From IN to EXISTS

A simple example of converting from IN to EXISTS or from uncorrelated to correlated queries (or vice versa), would be to move the set-test in the WHERE .. IN of the uncorrelated query to the WHERE of the EXISTS in the correlated query. For example, note the placement of the set-test in the following uncorrelated query:

```
SELECT          *
FROM            Student s
WHERE           s.stno IN -- link s.stno to the subquery
  (SELECT       g.student_number
  FROM          Grade_report g
  WHERE         grade = 'B');
```

Now, note the placement of the set test in the following correlated query:

```
SELECT          *
FROM            Student s
WHERE EXISTS  -- replace IN with EXISTS
  (SELECT       1  -- change the result set to 1
  FROM          Grade_report g
  WHERE         grade = 'B'
 AND            s.stno = g.student_number) -- move link here to correlate the subquery
;
```

These two queries produce the following output:

```
  STNO      SNAME        MAJOR   CLASS       BDATE
--------  ----------   --------  -------   ----------
     2    Lineas       ENGL         1      15-APR-01
     3    Mary         COSC         4      16-JUL-98
     5    Zelda        COSC                12-FEB-98
     6    Ken          POLY                15-JUL-01
     7    Mario        MATH                12-AUG-01
     8    Brenda       COSC         2      13-AUG-97
     3    Kelly        MATH         4      12-AUG-01
    14    Lujack       COSC         1      12-FEB-97
    15    Reva         MATH         2       10-JUN-01
    19    Harley       POLY         2       16-APR-02
    24    Chris        ACCT         4      12-FEB-98
   .
   .
   .

   148    Sebastian    ACCT         2      14-OCT-96
   155    Lindsay      UNKN         1      15-OCT-98
   157    Stephanie    MATH                16-APR-02

27 rows selected.
```

This example gives us a pattern to move from one kind of query to the other and test the efficiency of both kinds of queries. In the EXISTS version, we changed the result set for the subquery to 1 by removing the original result set of *g_student_number*s.

9.3.3 NOT EXISTS

There are some situations in which the EXISTS and NOT EXISTS predicates are necessary. For example, if we ask a "for all" question, it must be answered by "existence" (actually, the lack thereof [that is, "not existence"]). In logic, the statement "find x for all y" is logically equivalent to "do not find x where there does not exist a y." In SQL, there is no "for all" predicate. Instead, SQL uses the idea of "for all" logic with NOT EXISTS. (A word of caution -- SQL is not simply a logic exercise, as we will see.) We will first see how EXISTS and NOT EXISTS work in SQL, and then tackle the "for all" problem. Consider the following correlated, existence query to find students who have made a C in some course:

```
SELECT      s.sname, s.stno, s.major
FROM        Student s
WHERE EXISTS
  (SELECT       'X'
  FROM          Grade_report gr
  WHERE         s.stno = gr.student_number
    /* return TRUE if a student has made a C */
  AND           gr.grade = 'C')
ORDER BY        s.sname;
```

This produces the following output:

```
SNAME                 STNO       MAJO
--------------------  ------------  ----------
Alan                   130       COSC
Benny                  161       CHEM
Bill                    70       POLY
Brenda                   8       COSC
Donald                  20       ACCT
Genevieve              153       UNKN
Gus                    160        ART
Jake                    31       COSC
Jessica                126       POLY
Ken                      6       POLY
Lionel                 163
Losmith                151       CHEM
Mario                    7       MATH
Monica                  62       MATH
Rachel                 131       ENGL
Reva                    15       MATH
Richard                 10       ENGL
Sadie                  125       MATH
Sebastian              148       ACCT
Smithly                147       ENGL
Steve                  127       ENGL
Susan                   49       ENGL
Thornton               158
Zelda                    5       COSC

24 rows selected.
```

The ORDER BY was added for comparison purposes. For this correlated query, "student names" are SELECTed when:

(a) The student is enrolled in a section (WHERE *s.stno = gr.student_number*), and

(b) The same student has a grade of C.

In the EXISTS version of this query, both (a) and (b) must be TRUE for the student row to be SELECTed. We use SELECT 1 or SELECT 'X' in our inner query because we want the subquery to return something if the subquery is TRUE. Therefore, SELECT .. EXISTS "says" SELECT .. WHERE TRUE, and the inner query is TRUE if any row is SELECTed in the inner query.

Here is the join version for comparison:

```
SELECT       DISTINCT s.sname, s.stno, s.major, g.grade
FROM         student s, grade_report g
WHERE        s.stno = g.student_number
   AND       g.grade = 'C'
   ORDER BY  s.sname;
```

Which gives:

```
SNAME                STNO     MAJOR    GRA
-------------------- -------- -------- --------
Alan                  130     COSC     C
Benny                 161     CHEM     C
Bill                   70     POLY     C
Brenda                  8     COSC     C
Donald                 20     ACCT     C
Genevieve             153     UNKN     C
Gus                   160     ART      C
Jake                   31     COSC     C
Jessica               126     POLY     C
Ken                     6     POLY     C
Lionel                163              C
Losmith               151     CHEM     C
Mario                   7     MATH     C
Monica                 62     MATH     C
Rachel                131     ENGL     C
Reva                   15     MATH     C
Richard                10     ENGL     C
Sadie                 125     MATH     C
Sebastian             148     ACCT     C
Smithly               147     ENGL     C
Steve                 127     ENGL     C
Susan                  49     ENGL     C
Thornton              158              C
Zelda                   5     COSC     C

24 rows selected.
```

Now consider the following query, where we change EXISTS to NOT EXISTS:

```
SELECT        s.sname
FROM          Student s
WHERE NOT EXISTS
   (SELECT    'X'
   FROM       Grade_report gr
   WHERE      s.stno = gr.student_number
   AND        gr.grade = 'C')
ORDER BY      s.sname;
```

This produces the following output:

```
SNAME
----------------------------------
Brad
Cedric
Chris
Cramer
Elainie
Fraiser
Francis
George
Harley
Harrison
Hillary
Holly
Jake
Jerry
Kelly
Lindsay
Lineas
Lujack
Lynette
Mary
Phoebe
Romona
Smith
Stephanie

24 rows selected.
```

In this query, we are still SELECTing with the pattern SELECT .. WHERE TRUE because all SELECTs with EXISTS work that way. However, the twist is the subquery must be FALSE to be SELECTed with NOT EXISTS. If the subquery is FALSE, then NOT EXISTS is TRUE and the outer row is SELECTed.

Now, logic implies if either (a) *s.stno* <> *gr.student_number* or (b) *gr.grade* <> 'C', then the subquery "fails" -- it is FALSE for that student row. Because the subquery is FALSE, the NOT EXISTS would return a TRUE for that row. Unfortunately, this logic is not quite what happens. Recall, we characterized the correlated query as follows:

```
LOOP1: For each row in Student s DO
          LOOP2: For each row in Grade_report DO
                    IF (gr.student_number = s.stno) THEN
                            IF (gr.grade = 'C') THEN TRUE
          END LOOP2;
     if TRUE, then student row is SELECTed
END LOOP1;
```

LOOP2 is completed before the next student is tested. In other words, just because there is a student number inequality, this will not cause the subquery to be FALSE. Rather, the entire subquery table is parsed, and the logic is more like this:

For the case *"EXISTS WHERE s.stno = gr.student_number…,"* is there a gr.grade = 'C'? If, when the student numbers are equal, no C can be found; then the subquery *fails* and is FALSE for that outer student row. So with NOT EXISTS we will SELECT students with student numbers equal in the **Grade_report** and **Student** tables, but who have no 'C' in the **Grade_report** table. The point about "no C in the **Grade_report** table" can only be answered TRUE by looking at all the rows in the inner query.

Consider this join version:

```
SELECT DISTINCT s.sname, g.grade
FROM Student s, Grade_report g
WHERE s.stno = g.student_number
  AND g.grade <> 'C';
```

This gives:

```
SNAME                          GRA
----------------------------   -------
Lineas                           D
Lujack                           B
Reva                             F
Lynette                          A
Brad                             F
George                           B
Lineas                           A
Mary                             B
Lynette                          B
Harrison                         F
Lindsay                          B
.
.
.
Reva                             B
Susan                            A
Fraiser                          B

47 rows selected.
```

This join returns 47 rows because it is telling us all the **Student-Grade_report** combinations where there is no C. Remember, a join is a Cartesian product restricted by *s.stno = g.grade_report*. So a student could have a C and also have some other grade. The query would return that student and the non-C grade. The NOT EXISTS only returns students with no C at all.

There is one other point to be made here. The two results from above (EXISTS vs. NOT EXISTS) have the same number of rows, but this is just a coincidence. If the two result sets are examined, you will notice the people in the two sets are all different. Two people in the second result set show up because they took no courses and hence had no rows in the inner query of the NOT EXISTS (<Smith,88..> and <Jake,191 ..>). One has to be careful to account for nulls situations.

An extra query to check for this situation could be:

```
SELECT     s.sname, s.stno, s.major
FROM       Student s
WHERE      s.stno NOT IN -- students who have taken no courses
(SELECT    g.student_number
FROM       Grade_report g);
```

Giving:

```
SNAME                STNO       MAJOR
-------------------- ---------- -----------
Smith                88
Jake                 191        MATH
```

9.4 SQL Universal and Existential Qualifiers -- the "for all" Query

The terms "for all," "for each," and "by all" are called "universal qualifiers," and "there exists" is the "existential qualifier." SQL has an existential predicate with the EXISTS predicate. As we mentioned above, SQL does not have a "for all" predicate; however, logically, the following relationship exists:

For all x, WHERE P(x) is true

is logically the same as:

There does not exist an x, WHERE P(x) is not true.

A "for all" type SQL query is less straightforward than the other queries we have studied and used. The "for all" type SQL query involves a double-nested, correlated query using the NOT EXISTS predicate. The next section shows an example.

Example 1

To show a "for all" type SQL query, we will use tables other than our student records. We have created a table called **Languages**. This table has names of students who have multiple foreign-language capabilities. We begin by looking at the table by typing the following query:

```
SELECT      *
FROM        Languages
ORDER BY    name;
```

This produces the following output:

```
NAME                LANGU
------------------- --------------
BRENDA              FRENCH
BRENDA              CHINESE
BRENDA              SPANISH
JOE                 CHINESE
KENT                CHINESE
LUJACK              SPANISH
LUJACK              FRENCH
LUJACK              CHINESE
LUJACK              GERMAN
MARY JO             CHINESE
MARY JO             FRENCH
MARY JO             GERMAN
MELANIE             CHINESE
MELANIE             FRENCH
RICHARD             FRENCH
RICHARD             SPANISH
RICHARD             GERMAN
RICHARD             CHINESE

18 rows selected.
```

Notice, for all languages in this table, RICHARD and LUJACK speak all four.

Another view of this table ordered by language is:

```
NAME                    LANGU
--------------------    --------------------
JOE                     CHINESE
MARY JO                 CHINESE
KENT                    CHINESE
LUJACK                  CHINESE
MELANIE                 CHINESE
RICHARD                 CHINESE
BRENDA                  CHINESE
MELANIE                 FRENCH
LUJACK                  FRENCH
MARY JO                 FRENCH
BRENDA                  FRENCH
RICHARD                 FRENCH
LUJACK                  GERMAN
RICHARD                 GERMAN
MARY JO                 GERMAN
BRENDA                  SPANISH
LUJACK                  SPANISH
RICHARD                 SPANISH

18 rows selected.
```

In this sort order, notice CHINESE occurs for all names.

Suppose we want to find out which languages are spoken by all students using SQL. This is a universal qualifier question. Although this manual exercise would be very difficult for a large table, for our practice table, we can answer the question by looking at the table sorted two ways as above.

To see how to answer a question of this type for a much larger table where sorting and examining the result would be tedious, we will construct a query. We will show the query and then dissect the result. The query to answer our question, "*Which language(s) are spoken by all students?*," looks like this:

```
SELECT name, langu
FROM    Languages x
WHERE NOT EXISTS
        (SELECT 'X'
        FROM Languages y
        WHERE NOT EXISTS
                (SELECT 'X'
                FROM Languages z
                WHERE x.langu =z.langu
                AND y.name=z.name));
```

As you will see, all of the "for all/for each/by all" questions follow this double-nested, correlated NOT EXISTS pattern. It is convenient to use the table aliases (x, y, and z) here for the three instances of the table, **Languages**.

The result set for this query will be:

```
NAME            LANGU
---------       ------------
BRENDA          CHINESE
RICHARD         CHINESE
LUJACK          CHINESE
MARY JO         CHINESE
MELANIE         CHINESE
JOE             CHINESE
KENT            CHINESE

7 rows selected.
```

The Way the Query Works

To SELECT a language spoken by all students, the query proceeds as follows:

a. SELECT a row in **Languages**(x) (outer query).

b. For that row, begin SELECTing each row again in **Languages**(y) (middle query).

c. For each of the middle query rows, we want the inner query (**Languages**(z)) to be TRUE for all cases of the middle query (TRUE is translated to FALSE by the NOT EXISTS). As each inner query is satisfied (it is TRUE), it forces the middle query to continue looking for a match -- to look at all cases and eventually conclude FALSE (evaluate to FALSE overall). If the middle query is FALSE, the outer query sees TRUE because of its NOT EXISTS.

To make the middle query (y) find FALSE, all of the inner query (z) occurrences must be TRUE (i.e., the languages from the outer query have to exist with all names from the middle one (y) in the inner one (z)). For an eventual match, every row in the middle query for an outer query row must be FALSE (i.e., every row in the inner query is TRUE).

These steps are explained in further detail in the next example where we used a smaller table, **Languages1** (so it will be easier to understand the explanation).

Example 2

Suppose we had this simpler table, **Languages1**, as shown below:

NAME	LANGU
Joe	Spanish
Mary	Spanish
Mary	French

*****Begin Note*****
Note, this table, **Languages1**, does not exist. You will have to create it. The attribute names and types are the same as the **Languages** table.
*****End Note*****

Using this smaller table, **Languages1**, let's now look at how we can answer the same question, "*Which language(s) are spoken by all students?*." We can see the answer is Spanish.

The query will be similar to the one used in Example 1:

```
SELECT          name, langu
FROM            Languages1 x
WHERE NOT EXISTS
        (SELECT  'X'
        FROM     Languages1 y
        WHERE NOT EXISTS
        (SELECT  'X'
                FROM     Languages1 z
                WHERE    x.langu = z.langu -- x and z .. so what languages occurs
                AND      y.name = z.name)) -- for all names
ORDER BY langu;
```

The output for this query will be:

```
NAME                LANGU
------------------- -----------
Mary                Spanish
Joe                 Spanish
```

The result set tells us Spanish is spoken by all students in the **Language1** table.

How this query works:

Here is the **Languages1** table again:

```
NAME        LANGU
Joe         Spanish
Mary        Spanish
Mary        French
```

1. The row <Joe, Spanish> is SELECTed by the outer query (x).
2. The row <Joe, Spanish> is SELECTed by the middle query (y).
3. The row <Joe, Spanish> is SELECTed by the inner query (z).
4. The inner query is TRUE:

```
X.LANGU = Spanish
Z.LANGU = Spanish
Y.NAME = Joe
Z.NAME = Joe
```

5. Because the inner query is TRUE, the NOT EXISTS of the middle query translates this to FALSE and continues with the next row in the middle query. The middle query SELECTs <Mary, Spanish> and the inner query begins again with <Joe, Spanish> seeing:

```
X.LANGU = Spanish
Z.LANGU = Spanish
Y.NAME = Mary
Z.NAME = Joe
```

This is FALSE, so the inner query SELECTs a second row <Mary, Spanish>:

```
X.LANGU = Spanish
Z.LANGU = Spanish
Y.NAME = Mary
Z.NAME = Mary
```

This is TRUE, so the inner query is TRUE. (Notice, the X.LANGU has not changed, yet the outer query (X) is still on the first row.)

6. Because the inner query is TRUE, the "NOT EXISTS" of the middle query translates this to FALSE and continues with the next row in the middle query. The middle query now SELECTs <Mary, French> and the inner query begins again with <Joe, Spanish> seeing:

```
X.LANGU = Spanish
Y.NAME = Mary
Z.NAME = Joe
```

This is FALSE, so the inner query SELECTs a second row <Mary, Spanish>:

```
X.LANGU = Spanish
Z.LANGU = Spanish
Y.NAME = Mary
Z.NAME = Mary
```

This is TRUE, so the inner query is TRUE.

7. Because the inner query is TRUE, the NOT EXISTS of the middle query again converts this TRUE to FALSE and wants to continue, but the middle query is out of rows. This means the middle query is FALSE.

8. Because the middle query is FALSE, and because we are testing this query:

```
"SELECT distinct name, language
FROM Languages1 x
WHERE NOT EXISTS
(SELECT 'X' FROM Languages1 y ...),"
```

the FALSE from the middle query is translated to TRUE for the outer query and the row <Joe,Spanish> is SELECTed for the final result set. Note, "Spanish" occurs with both "Joe" and "Mary."

9. The second row in the outer query will repeat the steps from above for <Mary, Spanish>. The value "Spanish" will be seen to occur with both "Joe" and "Mary" as <Mary, Spanish> is added to the result set.

10. The third row in the outer query begins with <Mary, French>. The middle query SELECTs <Joe, Spanish> and the inner query SELECTs <Joe, Spanish>. The inner query sees:

```
X.LANGU = French
Z.LANGU = Spanish
Y.NAME = Joe
Z.NAME = Mary
```

This is FALSE, so the inner query SELECTs a second row, <Mary, Spanish>:

```
X.LANGU = French
Z.LANGU = Spanish
Y.NAME = Joe
Z.NAME = Mary
```

This is FALSE, so the inner query SELECTs a third row, <Mary, French>:

```
X.LANGU = French
Z.LANGU = French
Y.NAME = Joe
Z.NAME = Mary
```

This is also FALSE. The inner query fails. The inner query evaluates to FALSE, which causes the middle query to see TRUE because of the NOT EXISTS. Because the middle query sees TRUE, it is finished and evaluated to TRUE. Because the middle query evaluates to TRUE, the NOT EXISTS in the outer query changes this to FALSE, and "X.LANGU = French" fails. It failed because *X.LANGU* = French did not occur with all values of the attribute, *name*.

Consider again the "for all" query we have presented:

```
SELECT          name, langu
FROM            Languages1 x
WHERE NOT EXISTS
        (SELECT  'X'
        FROM     Languages1 y
        WHERE NOT EXISTS
        (SELECT  'X'
              FROM      Languages1 z
              WHERE     x.langu = z.langu -- x and z .. what language occurs
              AND       y.name = z.name)) -- for all names
ORDER BY langu;
```

The tip-off of what a query of this kind means can be found in the inner-most query. You will find a phrase that says, "WHERE **x.langu** = **z.langu**…" The *x.langu* is where the query is testing ***which language*** occurs for all names. Using our x, y, z notation, the inner query tests x and z.

This query is a SQL realization of a relational division exercise. Relational division is a "for all" operation just like that which we have illustrated above. In relational algebra, the query must be set up into a divisor, dividend, and quotient in this pattern:

Quotient (B) ←Dividend(A, B) divided by Divisor (A).

If the question is *"What language for all names,"* then the Divisor, A, is names, and the quotient, B, is language. It is most prudent to set up SQL like relational algebra with a two column table (like **Languages** or **Languages1**) for the Dividend and then treat the Divisor and the Quotient appropriately. Our query will have the attribute for language, *x.langu*, in the inner query; *langu* will be the quotient. We have chosen to also report the name attribute in the result set.

Example 3

Note, the preceding query is completely different from the following one which asks, "*Which students speak all languages?*":

```
SELECT          DISTINCT name, langu
FROM            Languages1 x
WHERE NOT EXISTS
         (SELECT 'X'
        FROM     Languages1 y
        WHERE NOT EXISTS
                (SELECT  'X'
                FROM     Languages1 z
                WHERE    x.name = z.name  -- x and z: What names occur for all languages?
                AND      y.langu =  z.langu))  -- for all languages
ORDER BY langu;
```

This would produce the following output:

```
NAME            LANGU
----------      ------------
Mary            French
Mary            Spanish
```

Note the phraseology, "find the *name* for all languages," which infers *x.name* will occur in the WHERE of the inner query. If you look back at the previous example, "find **languages** for all names" means *x.langu* is in the inner query.

Using the table **Languages**, the following query:

```
SELECT          DISTINCT name, langu
FROM            Languages x
WHERE NOT EXISTS
        (SELECT 'X'
        FROM     Languages y
        WHERE NOT EXISTS
                (SELECT  'X'
                FROM     Languages z
                WHERE    x.name = z.name -- x and z .. what names occur
                AND      y.langu = z.langu))  -- for all languages?
ORDER BY langu;
```

Would give:

```
NAME            LANGU
-----------     -----------
LUJACK          CHINESE
RICHARD         CHINESE
LUJACK           FRENCH
RICHARD          FRENCH
LUJACK          GERMAN
RICHARD         GERMAN
LUJACK          SPANISH
RICHARD         SPANISH

8 rows selected.
```

The inner query contains *x.name*, which means the question was *"Which names occur for all languages?"* or, put another way, *"Which students speak all languages?"* The "all" goes with languages for *x.name*.

Exercises for Chapter 9

As you do the exercises, unless it is stated otherwise, you will be using the tables from our standard **Student-Course** database. Also, as you do the exercises, it will be a good idea to copy/paste your query as well as your query result into a word processor.

9-1. List the names of students who have received C's. Do this in three ways: (a) as a *join*, (b) as an *uncorrelated* subquery, and (c) as a *correlated* subquery. Show all the results and account for any differences.

9-2. In the section, "Existence Queries and Correlation," we were asked to find the names of students who have taken a computer science class and earned a grade of B. We noted this could be done in several ways. One query could look like the following:

```
SELECT      s.sname
FROM        Student s
WHERE s.stno IN
    (SELECT gr.student_number
    FROM    Grade_report gr, Section
    WHERE   Section.section_id = gr.section_id   /* join condition Grade_report-Section */
    AND     Section.course_num LIKE 'COSC____'
    AND     gr.grade = 'B');
```

Re-do this query putting the finding of the COSC course in a correlated subquery. The query should be: The **Student** table uncorrelated subquery to the **Grade_report** table correlated EXISTS to the **Section** table.

9-3. In the section "SQL Universal and Existential Qualifiers," we illustrated an existence query:

```
SELECT s.sname
FROM        Student s
WHERE EXISTS
    (SELECT 'X'
    FROM    Grade_report gr
    WHERE   s.stno = gr.student_number
    AND     gr.grade = 'C');
```

and a NOT EXISTS version:

```
SELECT s.sname
FROM        Student s
WHERE NOT EXISTS
    (SELECT 'X'
    FROM    Grade_report gr
    WHERE s.stno = gr.student_number
    AND      gr.grade = 'C');
```

Show that the EXISTS version is the opposite of the NOT EXISTS version -- count the rows in the EXISTS result, the rows in the NOT EXISTS result, and the rows in the **Student** table. Also, devise a query to test the opposite with IN and NOT..IN.

9-4. a. Discover whether all students take courses by counting the students, then count those students whose student numbers are in the **Grade_report** table and those who are not. Use IN and then NOT..IN, and then use EXISTS and NOT EXISTS. How many students take courses and how many students do not?

b. Find out which students have taken courses but have not taken COSC courses. Create a set of student names and courses from the **Student**, **Grade_report**, and **Section** tables (use the prefix COSC to indicate COSC courses). Then use NOT..IN to subtract from that set another set of student names where students (who take courses) have taken COSC courses. For this set difference, use NOT..IN.

c. Change NOT..IN to NOT EXISTS (with other appropriate changes) and explain the result. The "other appropriate changes" include adding the correlation and the change of the result attribute in the subquery set.

9-5. There is a table called **Plants** in our **Student-Course** database. Display the table contents and determine which company or companies have plants in all cities. Verify your result manually. Note: If you are having trouble finding **Plants**, ask yourself who owns the table **Plants**?

9-6. a. Run the following query and print the result:

```
SELECT distinct name, langu
FROM Languages x
WHERE NOT EXISTS
     (SELECT 'X'
     FROM Languages y
     WHERE NOT EXISTS
               (SELECT 'X'
               FROM Languages z
               WHERE x.langu =z.langu
               AND y.name=z.name));
```

Save the query (e.g., save forall) and hand in the result.

b. Re-create the **Languages** table under your account number (call it some other name such as **LANG1**). To do this, first create the table and then use the INSERT statement with the subselect option (INSERT INTO LANG1 AS SELECT * FROM Languages;).

c. Add a new person to your table who speaks only BENGALI.

d. Recall your stored SELECT from above (get forall).

e. CHANGE the table from **LANGUAGES** to **LANG1** (for all occurrences use CHANGE/ Languages/lang1/ repeatedly, assuming you called your table **LANG1**).

f. Start the new query (the one you just created with **LANG1** in it).

g. How is this result different from the situation when "Newperson" was not in **LANG1**? Provide an explanation of why the query did what it did.

9-7. (Refer to Exercise 7-7 in Chapter 7) The **D2M** table is a list of four-letter department codes with the department names. In Exercise 7-7, we created a table called **Secretary**, which should now have data like this:

Secretary

dCode	**Name**
ACCT	Sally
COSC	Chris
ENGL	Maria

In Exercise 7-7, we did the following:

a. Create a query to list the names of departments that have secretaries (use IN and the **Secretary** table in a subquery with the **Department_to_major** table in the outer query). Save this query as q77a.

b. Create a query to list the names of departments without secretaries (use NOT..IN). Save this query as q77b.

c. Add one more row to the **Secretary** table containing <null, ‘Brenda’>. (This could be a situation in which we have hired Brenda but have not yet assigned her to a department.)

d. Recall q77a and re-run it.

e. Recall q77b and re-run it.

We remarked in Exercise 7-7 that the NOT..IN predicate has problems with nulls. The behavior of NOT..IN when nulls exist may surprise you. If nulls may exist in the subquery, then NOT..IN should not be used. If you use NOT..IN in a subquery, you must ensure nulls will not occur in the subquery or you must use some other predicate (such as NOT EXISTS). Perhaps the best solution is to avoid NOT..IN.

Here, we repeat Exercise 7-7 using NOT EXISTS:

a. Re-word query q77a to use EXISTS. You will have to correlate the inner and outer queries. Save this query as q99a.

b. Re-word query q77b to use NOT EXISTS. You will have to correlate the inner and outer queries. Save this query as q99b. You should not have a phrase “IS NOT NULL” in your NOT EXISTS query.

c. Re-run q99a with and without <null, Brenda>.

d. Re-run q99b with and without <null, Brenda>.

Note the difference in behavior versus the original question. List the names of those departments that have/do not have secretaries. The point here is to encourage you to use NOT EXISTS in a correlated query rather than NOT..IN.

References

Earp, R., & Bagui, S. (2001). “An In-Depth Look at Oracle’s Correlated Subqueries,” *Oracle Internals*, Vol. 3(4), 2–8.