Subqueries

Introduction

At times, the final answer to a request requires multiple steps. For instance, say we wanted to know which customers have a shipping cost greater than the average shipping cost of customers in zip code 78710. This requires a two-step approach. First, we need to figure out what the average shipping cost of customers in zip code 78710. Once we derive this answer, we can use the answer to figure out which customers have a shipping cost greater than that. We could write and execute two separate queries to get our answer, or we could “nest” the inner query (average shipping cost of customers in zip code 78710) inside the outer query (customers with a shipping cost greater than).

Using two separate queries it would be: SELECT AVG(ShipCost) FROM Orders WHERE ShipZip = ‘78710’; Assuming this average returns a value of 5, we could code the second query as: SELECT Customer# FROM Orders WHERE ShipCost > 5;. This has an inherent flaw in that the second query would need to be changed before each run because the actual average will constantly be changing as rows are added, changed and deleted from the Orders table.

To solve this problem, we can use a subquery, using the inner query on the WHERE clause:
SELECT Customer# FROM Orders WHERE ShipCost > (SELECT AVG(ShipCost) FROM Orders WHERE ShipZip = ‘78710’);. This query requires no maintenance because the inner query is executed first, then the answer used in the comparison.

Types of Subqueries

There are several different types of subqueries. A *single-row subquery* (also called a single value subquery) produces a single value that gets returned to the outer query. A *multiple-row subquery* produces more than one result that gets returned to the outer query. A *multiple-column subquery* produces a result that containing more than one column, then returns it to the outer query. A *correlated subquery* executes the subquery once for each row in the outer query, based on a specified column. Exactly which type of subquery to use is based on the result to be achieved. Sometimes, more than one type of subquery can be used to solve the request.

A subquery can be used in the WHERE, HAVING, SELECT or FROM clause of a query. Generally, it is more common to see it used on the WHERE or HAVING clause. All of the subquery types listed above can be used on the WHERE, HAVING and FROM clauses, but only a single-row subquery can be used on the SELECT clause.

Single-row Subqueries

You should use a single-row subquery when the outer query’s results are conditioned on a single, unknown value. The average ship cost subquery from above is an example of a single-row (single value) subquery. Using a subquery on the WHERE clause is one of the most common scenarios that requires the use of a subquery. Using subqueries to determine and return a value, instead of “hard-coding” the value, is more efficient and requires less maintenance.

Single-row subqueries use the following single-row operators: =, <, >, >=, <=, and <>. The IN operator is allowed, but ONLY if a single value is used.

Subqueries can also be coded on the HAVING clause to restrict group results, much the same way as rows are restricted by the WHERE clause. Keep in mind that the result returned from the subquery in this case will be from a group function. As an example, say we want to know which categories return a higher average profit than the average profit of books in the Literature category:

```
SELECT Category, AVG(Retail - Cost)
FROM Books
GROUP BY Category
HAVING AVG(Retail - Cost) > (SELECT AVG(Retail – Cost) FROM Books WHERE Category = ‘Literature’);
```

Subqueries used on the SELECT clause return a value that can be used for every row of output in the final query results, either as a display column and/or in a calculation. If we wanted to show each book’s price and how it compares to the average price of all books (over/under), we would need:

```
SELECT Title, Retail, (SELECT AVG(Retail) FROM Books) “Overall Average”, Retail - (SELECT AVG(Retail)
FROM Books) “Difference +/-” FROM
Books;
```

Multiple-row Subqueries

As stated before, multiple-row subqueries return more than one value (row) back to the outer query. The WHERE and HAVING clauses are most commonly used. Keep in mind that a multiple-row operator, (IN, ALL, ANY), must be used in the comparison, since the subquery results are a set, not a single value.

For instance, if we wanted to know which books have a price matching the highest price of any book category, we would code:

```
SELECT Title, Retail, Category
FROM Books WHERE Retail IN (SELECT MAX(Retail) FROM Books GROUP BY Category);
```

In this case, each books retail price is “looked up” in the subquery results. The ALL and ANY operators can be used when the subquery results need to be treated as a set of values, not individually. If we wanted to show which books have a price greater than the most expensive book in the Cooking category:

```
SELECT Title, Retail
FROM Books
WHERE Retail >ALL (SELECT Retail FROM Books WHERE Category = ‘COOKING”);
```

The HAVING clause can be used for multiple-row subqueries, but keep in mind the subquery result will be a group function value for each group.

Multiple-Column Subqueries

So far, the subqueries we have discussed all return a single value, basically one row of one column. Subqueries may return more than one column of data, and can be passed to the outer query on the FROM, WHERE or HAVING clauses.

When used on the FROM clause, Oracle treats the subquery result as a temporary table, also referred to as an inline view. This temporary table can be used just like any other table that physically exists in the database, even if the rows in the subquery are grouped data (formed by a GROUP BY). For an example, if we wanted those books which have a higher than average selling price compared to books in the same category, we could code:

```
SELECT B.Title, B.Retail, A.Category, A.CatAvg
FROM Books B, (SELECT Category, AVG(Retail) CatAvg FROM Books GROUP BY Category) A
WHERE B.Category = A.Category AND B.Retail > A.CatAvg
```

The subquery returns the group data, which can then be “joined” back to the Book table and the Retail amounts compared.

We can also use a subquery on the WHERE and HAVING clauses. The use of the IN operator is required since multiple columns will be returned. If necessary, refer back to the earlier unit that discussed the IN operator. Using a similar scenario, say we want to know the most expensive books in each category. We would then code:

```
SELECT Title, Retail, Category
FROM Books
WHERE (Category, Retail) IN (SELECT Category, MAX(Retail) FROM Books GROUP BY Category)
```

NULL Values

Special consideration must be given to NULL values that are returned from a subquery. The use of the NVL function and the IS NULL comparison operator are a must when working with subqueries that may return a NULL.

Subqueries on the UPDATE and DELETE Statements

Subqueries can return a value that can be used in an action query (UPDATE, DELETE). If used on the UPDATE statement, it can return a value that gets assigned to another column in the SET clause, or can be used as a condition on the WHERE clause. When used on a DELETE statement, it is used as a condition on the WHERE clause.