Lesson 6: coding subqueries

Objectives:

- Learn what a subquery is
- When to use a subquery
- Nested subqueries
- USE IN, NOT IN
- Use the ALL, ANY SOME, and EXISTS keywords
- Use Inline queries in Subqueries
- Use subqueries in HAVING, SELECT, and FROM clauses

In the Book:

Read ***Chapter 7: How to code subqueries***

Lecture Notes:

A subquery is a SELECT statement coded within another SQL statement.
Can be used as a search condition in a WHERE or HAVING clause
CAN be used as a table specification in a FROM clause
Can be used as a column specification in a SELECT clause

Can return a single value, a list of values retrieved from one column, or a table (rows and columns).

Example:

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_total >
(SELECT AVG(invoice_total)
FROM invoices)
ORDER BY invoice_total.
```

In this example the inner(subquery) is evaluated first returning a value using the aggregate AVG function. The value 1879.741316 replaces the subquery. The outer query then executes in this manner.

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_total > 1879.741316
```

```
FROM invoices
ORDER BY invoice_total
```

Most subqueries can be replaced by JOINS and most JOINS can be replaced by subqueries.

The above example is an instance where a JOIN of any type could not be substituted for the subquery.

Respective advantages of subqueries and joins.

Subqueries

Can include result sets containing columns of multiple tables.

When a direct relationship exists easier to read and follow.

Test against an aggregate such as MIN, MAX or AVG

Easier to read and follow if the relationship between two tables is ad-hoc

Long complex queries can be easier to code using subqueries

IN (NOT IN) phrase in a subquery. When a subquery returns a list of values from a single column use the IN/NOT IN phrase in the outer query to test to test against the list.

```
SELECT vendor_id, vendor_name, vendor_state

FROM vendors

WHERE vendor_id NOT IN

 (SELECT DISTINCT vendor_id from invoices)

ORDER BY vendor_id
```

Explanation: the subquery will return a list of distinct vendor ids from the invoices table. It will then test against these vendor_ids to display the names, states, and IDs of vendors that *don't* have invoices.

COMPARISON Operators. Use in a WHERE clause to compare an expression with the results of a subquery.

In the example below the result set returned would be a list of invoices whose "balance due" is less than the average "balance due" of all invoices. Since some invoices are paid in full you would want to exclude the invoices that have a balance due of zero.

First get the query would analyze the subquery and return the value to compare against.

```
SELECT AVG(invoice_total – payment_total – credit_total )

 FROM invoices
```

WHERE invoice_total – payment_total- credit_total >0)

This will return a value of 2910.947273.

Now the outer query compares all invoices with a "balance due" against this average.

SELECT invoice_number, invoice_date, invoice_total – payment_total- credit_total AS balance_due

FROM invoices

WHERE invoice_total – payment_total- credit_total > 0

AND WHERE invoice_total – payment_total- credit_total <

(SELECT AVG(invoice_total – payment_total – credit_total)

FROM invoices

WHERE invoice_total – payment_total- credit_total >0)

ORDER BY invoice_total DESC

THE ALL keyword

When a subquery returns a list the outer query with this comparison operator will test every value in the list. Often this **type of comparison can be replaced with a MAX function, avoiding making the query easier to read.**

WHERE invoice_total >

(SELECT MAX(invoice_total) FROM invoices WHERE vendor_id =34)

Produces the same result as

WHERE invoice_total> ALL

(SELECT invoice_total FROM invoices WHERE vendor_id =34)

SOME and ANY keywords. Identical – compares a list generated by the subquery and tests in the outer query if any value matches the comparison.

Correlated subquery. Similar to a loop, the subquery executes once for each row in the main query.

SELECT vendor_id, invoice_number, invoice_total

FROM invoices i

WHERE invoice_total >

(SELECT AVG(invoice_total)

FROM invoices

WHERE vendor_id = i.vendor_id)

This query will execute the each row of the outer query and compare or test the invoice_total it against the average invoice total for each distinct vendor. Why would you do this? You would be looking to see how many invoices have a total greater than the average for each vendor.

EXISTS/NOT EXISTS operator.

Doesn't return a value, instead returns a true or false condition. For example if you wanted a list of vendors that have no associated invoices first test to see in the subquery whether the vendor_id is present in the invoices table. If it isn't print the row from the vendor. THIS would test for false (NOT EXISTS)

Conversely if you wanted a list of vendors that *have* invoices use the EXISTS operator to test for true. In other words if the vendor_id is present in the invoices tables then return the vendor information.

Subqueries in SELECT, HAVING and FROM clauses.

SELECT clause subqueries. – usually the same result can be accomplished by a join.

SELECT vendor_name,

(SELECT MAX(invoice_date) FROM invoices

WHERE vendor_id = vendors.vendor.vendor_id) AS latest_invoice

Same result using a join.

SELECT vendor_name, MAX(invoice_date) as latest_inv

FROM vendors

LEFT JOIN invoice ON vendors.vendor_id = invoices.vendor_id

GROUP BY vendor_name

Using subqueries in HAVING clauses function syntactically the exact same way as subqueries in WHERE clauses.

Using the subqueries in a FROM clause.

Often referred to as an inline view. This type of subquery returns a set of rows and columns from which the outer query uses as its data source.

In this example we first add up the invoice amounts for each vendor grouped by state and produce an inline view of the result in the subquery.

```
SELECT vendor_state, vendor_name, SUM(invoice_total) AS sum_of_ invoices

FROM vendors JOIN invoices

 ON vendors.vendor_id = invoices.vendor_id

GROUP BY vendor_state, vendor_name
```

Then the outer query further summarizes the invoice totals by state. Notice that the subquery functions as table name where in fact no table exists. This method requires a table alias – in this case the t.

```
SELECT vendor_state, MAX(sum_of_invoices)

FROM

        (SELECT vendor_state, vendor_name, SUM(invoice_total) AS sum_of_ invoices

        FROM vendors JOIN invoices

 ON vendors.vendor_id = invoices.vendor_id

        GROUP BY vendor_state, vendor_name) t

GROUP by vendor_state
```

So what does this query accomplish? It gets the largest invoice for the top vendor in each state.

Procedure for building complex queries.

1. **State the problem in English and seek agreement from the requestor and the query author of what the expected result should be.**

2. **Use pseudocode to outline the steps to build the query.**

3. **Run subqueries or nested joins separately to ensure they work before embedding them in the query.**

4. **Code and test the final query – preferably with multiple and indicative test rows.**