**Subqueries**

SQL supports writing queries within queries, or **nesting queries**

**Outer query:**

-The outermost query is a query whose result set is returned to the caller

**Inner query(Sub-Query):**

-is a query whose result is used by the outer query

- The inner query acts in place of an expression that is based on constants or variables and is evaluated at run time.

- A subquery can be single-valued, multivalued, or table-valued. That is, a subquery can return a single value(Scalar Subqueries), multiple values(Multivalued Subquerie), or a whole table result.

-Unlike the results of expressions that use constants, the result of a subquery can change, because of changes in the queried tables. When you use subqueries, you avoid the need for separate steps in your solutions that store intermediate query results in variables.

**A subquery can be either self-contained or correlated.**

Both self-contained and correlated subqueries can return a scalar or multiple values

**Self-contained subquery:** Subquery has no dependency on tables from the outer query

**Correlated subquery:** Subquery has dependency on tables from the outer query

**Self-contained subqueries**

Every subquery has an outer query that contains it.

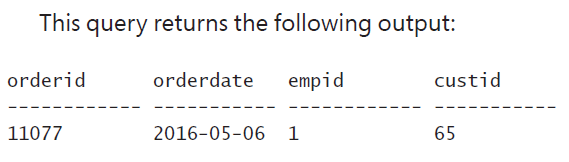
**Self-contained subqueries:** are subqueries that are independent of the tables in the outer query. Self-contained subqueries are convenient to debug, because you can always highlight the inner query, run it, and ensure that it does what it’s supposed to do. Logically, the subquery code is evaluated only once before the outer query is evaluated, and then the outer query uses the result of the subquery.

The following sections take a look at some concrete examples of self-contained subqueries.

**Self-contained scalar subquery**

A scalar subquery is a subquery that returns a single value. Such a subquery can appear anywhere in the outer query where a single-valued expression can appear (such as ***WHERE* or *SELECT***).

For example, suppose you need to query the *Orders* table in the *TSQLV4* database and return information about the order that has the maximum order ID in the table. You could accomplish the task by using a variable. The code could retrieve the maximum order ID from the *Orders* table and store the result in a variable. Then the code could query the *Orders* table and filter the order where the order ID is equal to the value stored in the variable. The following code demonstrates this technique:



DECLARE @maxid AS INT = (SELECT MAX(orderid)

FROM Sales.Orders);

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderid = @maxid;

You can substitute the variable with a scalar self-contained subquery, like so:

SELECT orderid, orderdate, empid, custid

SUBQUERY!!

Because it is contained in where clause

FROM Sales.Orders



WHERE orderid = (SELECT MAX(O.orderid)



FROM Sales.Orders AS O);



For **a scalar subquery** to be valid, it must return no more than one value. If a scalar subquery returns more than one value, it fails at run time. If a scalar subquery returns no value, the empty result is converted to a *NULL*.

Selects the empid from the subquery and SELECTS all the orderid that is associated with that empid

Sub query returns one value



SELECT orderid

FROM Sales.Orders

WHERE empid =

(SELECT E.empid

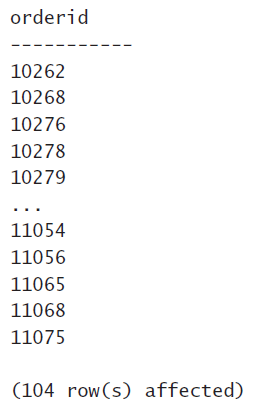


FROM HR.Employees AS E



WHERE E.lastname LIKE N'C%');





The subquery gets the employees lastname that starts with a C and SELECTS that employees empid to return. This is scalar because there is only one employees with last name starting with C



**If the subquery returns more than one value, the query fails.** For example, try running the query with employees whose last names start with *D*:

The query fails because **the subquery returns two values.** And you cannot store two values into empid this can be solved by using the (IN) keyword, which is discussed below

SELECT orderid

FROM Sales.Orders

WHERE empid =

(SELECT E.empid



FROM HR.Employees AS E



WHERE E.lastname LIKE N'D%');

The subquery returns two values for





**Self-contained multivalued subquery**

**IN:** Subquery returns multiple values

A multivalued subquery is a subquery that returns multiple values as a single column. Some predicates, such as the *IN* predicate, operate on a multivalued subquery.

EXAMPLE:

READ AS: WHERE empid FROM Sales.Orders is in the subquery.

Very similar to how a inner join works

SELECT orderid, empid

FROM Sales.Orders

WHERE empid IN

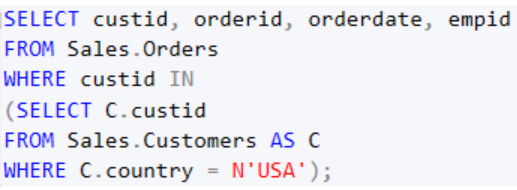
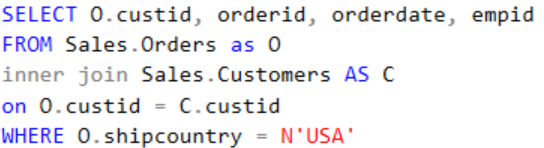


(SELECT E.empid

FROM HR.Employees AS E

WHERE E.lastname LIKE N'D%');

EXAMPLE:

OUTPUTS THE SAME RESULTS

**NOT IN**(Same as a LEFT OUTER JOIN)

As with any other predicate, you can negate the *IN* predicate with the *NOT* operator.

For example, the following query returns customers who did not place any orders:

The outer query returns customers with IDs that do not appear in the result of the subquery—in other words, customers who did not place orders.

SELECT custid, companyname



FROM Sales.Customers



WHERE custid not in (SELECT O.custid FROM Sales.Orders as O)



The subquery returns the IDs of all customers that appear in the *Orders* table. In other words, it returns only the IDs of customers who placed orders.



**Correlated subqueries**(https://www.youtube.com/watch?v=0ETfzlAQqBQ&t=333s)

Correlated subqueries are subqueries that refer to **attributes** from the tables that appear in the **outer query.**

**This means the subquery is dependent on the outer query and cannot be invoked independently.** Logically, the subquery is evaluated separately for each outer row.

**The inner query is executed for every single record that will be returned by the outer query**

OUTER QUERY

**The outer query examines every row in the Sales.Orders table** which has 830 rows. But **only returns if the orderid matches(…...**WHERE orderid = … **) that of the subquery**(MAX orderid), otherwise it is not returned.

SELECT custid, orderid, orderdate, empid

FROM Sales.Orders AS O1

WHERE orderid =

(SELECT MAX(O2.orderid)

FROM Sales.Orders AS O2

WHERE O2.custid = O1.custid);

INNER QUERY

Find the max of the custid where 02.custid=01.custid

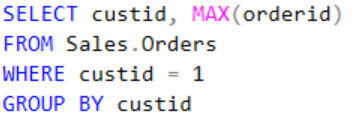
So for each unique custid(1,2,3,4…) it will always return the same orderid

EX: the subquery for custid 1 will always return 11011 because that is the max orderid

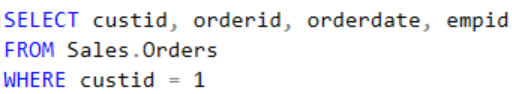


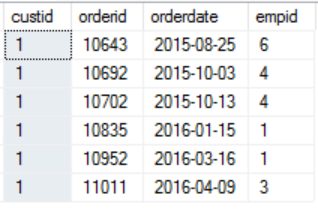
WALK THROUGH

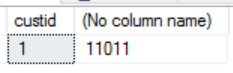
CONSIDER THIS AS THE INNER QUERY



CONSIDER THIS AS THE OUTER QUERY







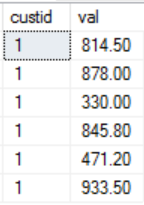


The outer row’s order ID—10643—is compared with the inner one—11011—and because there’s no match in this case, the outer row is filtered out. Then outer row’s orderid 10692 is compared with the inner ones 11011…no match so it is filtered…this happens until there is a match



ANOTHER EXAMPLE

SUBQUERY

SUM

1) 100 \* 814.50 / (4273.00) = 19.06

1) 100 \* 878.00 / (4273.00) = 20.55

. Sum of all value with custid 1

.

2) 100 \* 88.80 / (1402.95) = 6.33

Sum of all value with custid 2

SELECT orderid, custid, val,

CAST(100 \* val / (SELECT SUM(O2.val)



FROM Sales.OrderValues AS O2



WHERE O2.custid = O1.custid)

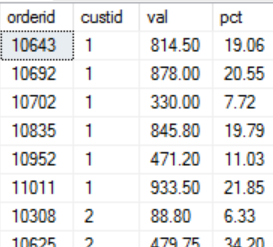




AS NUMERIC(5,2)) AS pct

FROM Sales.OrderValues AS O1

ORDER BY custid, orderid;



The EXISTS predicate: (is a two valued predicate)

T-SQL supports a predicate called EXISTS that accepts a subquery as input and returns TRUE if the subquery returns any rows and FALSE otherwise. For example, the following query returns customers from Spain who placed orders:

The **outer query** against the Customers table filters only customers from Spain for whom the EXISTS predicate returns TRUE.

The EXISTS predicate returns TRUE if the current customer has related orders in the Orders table.

SELECT custid, companyname

FROM Sales.Customers AS C

WHERE country = N'Spain'

AND EXISTS (SELECT \* FROM Sales.Orders AS O



WHERE O.custid = C.custid);



One of the benefits of using the EXISTS predicate is that you can intuitively phrase queries that sound like English. For example, this query can be read just as you would say it in ordinary English: Return customers from Spain if they have any orders where the order’s customer ID is the same as the customer’s customer ID.

ANOTHER EXAMPLE:

SELECT custid, companyname

FROM Sales.Customers AS C

WHERE EXISTS

(SELECT \*



FROM Sales.Orders AS O



WHERE O.custid = C.custid



AND EXISTS

(SELECT \*



FROM Sales.OrderDetails AS OD



WHERE OD.orderid = O.orderid



AND OD.ProductID = 12));



NOT EXISTS

SELECT custid, companyname

FROM Sales.Customers AS C

WHERE country = N'Spain'

AND NOT EXISTS (SELECT \*

FROM Sales.Orders AS O

WHERE O.custid = C.custid);

**Returning previous or next values**

Suppose you need to query the *Orders* table in the *TSQLV4* database and return, for each order, information about the current order and also the previous order ID. The tricky part is that the concept of “previous” implies order, and rows in a table have no order. One way to achieve this objective is with a T-SQL expression that means “the maximum value that is smaller than the current value.” You could use the following T-SQL expression, which is based on a correlated subquery, for this:

Read as:

From Sales.Orders as O2, Where O2.orderid is less than O1.orderid, Select the max orderid from O2. Then give it an alias name prevorderid

SELECT orderid, orderdate, empid, custid,

(SELECT MAX(O2.orderid)



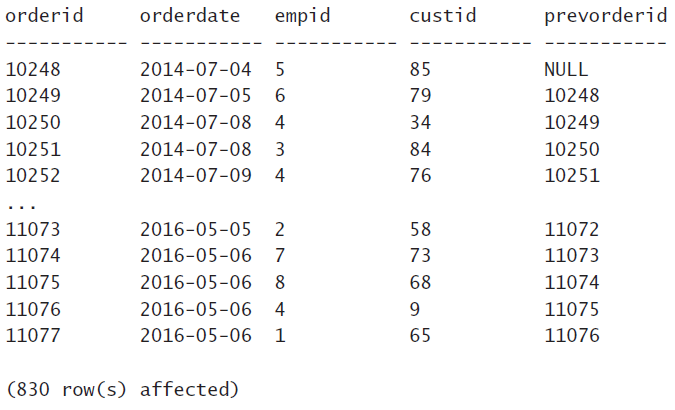
FROM Sales.Orders AS O2



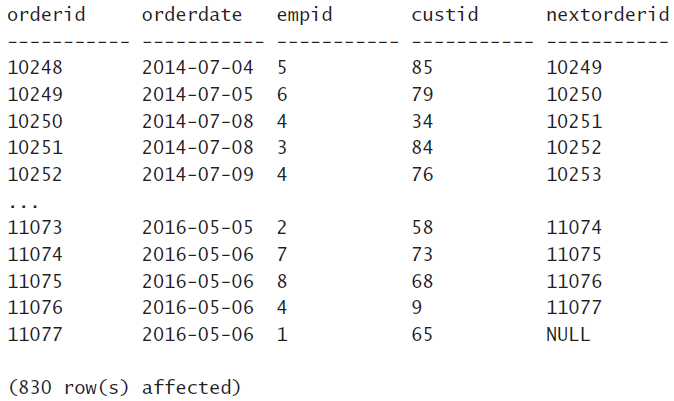
WHERE O2.orderid < O1.orderid) AS prevorderid



FROM Sales.Orders AS O1;



because there’s no order before the first order, the subquery returned a *NULL* for the first order.



Returns the next orderid

SELECT orderid, orderdate, empid, custid,

(SELECT MIN(O2.orderid)

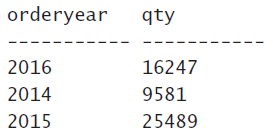
FROM Sales.Orders AS O2

WHERE O2.orderid > O1.orderid) AS nextorderid

FROM Sales.Orders AS O1;

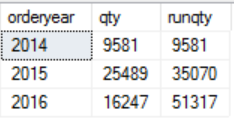
**Using running aggregates**

*Running aggregates* are aggregates that accumulate values based on some order. In this section, I use the *Sales.OrderTotalsByYear* view to demonstrate a technique that calculates those. The view has total order quantities by year. Query the view to examine its contents:



SELECT orderyear, qty

FROM Sales.OrderTotalsByYear;



SELECT orderyear, qty,

(SELECT SUM(O2.qty)



FROM Sales.OrderTotalsByYear AS O2



WHERE O2.orderyear <= O1.orderyear) AS runqty



FROM Sales.OrderTotalsByYear AS O1

ORDER BY orderyear;







Sums everything that is less than or equal 2014 ( 9581 )

SELECT SUM(O2.qty) as SUMQTY

FROM Sales.OrderTotalsByYear AS O2

WHERE O2.orderyear <= '2014'

Sums everything that is less than or equal to 2015 ( 9581 + 25489 )

SELECT SUM(O2.qty) as SUMQTY

FROM Sales.OrderTotalsByYear AS O2

WHERE O2.orderyear <= '2015'

Sums everything that is less than or equal to 2016 ( 9581 + 25489 + 16247 )

SELECT SUM(O2.qty) as SUMQTY

FROM Sales.OrderTotalsByYear AS O2

WHERE O2.orderyear <= '2016'