Subqueries

# Subqueries

## Review of Simple Subqueries

<https://technet.microsoft.com/en-us/library/ms189575(v=sql.105).aspx>

A subquery is a query within a query. Most typically we see this as a select query inside another select query. It allows us to execute an inner query and use those results in the outer query.

We have already had a look at some simple subqueries in our SQL review at the beginning of this course. We used subqueries as the results for IN and EXISTS, which is a fairly common usage. As a refresher, consider the following examples from our previous notes:

SELECT or.\*

FROM Sales.Orders or

WHERE EXISTS ( SELECT 1

FROM Sales.Customers cu

WHERE cu.CustomerName = 'Cosmina Vlad'

AND or.CustomerID = cu.CustomerID );

SELECT \*  
FROM Sales.Orders  
WHERE CustomerID IN ( SELECT CustomerID  
 FROM Sales.Customers  
 WHERE CustomerName = 'Cosmina Vlad' );

In both examples above, we are using the results of one query inside another.

Both of the queries above would be fairly trivial to rewrite as simple joins. Let’s look at a more complex example where that is not true.

## Inline Views

A common use of subqueries is to self-join a table to perform some kind of aggregation, then provide full record details for that aggregated set.

For example, in the WideWorldImporters database, I want to get the full order details from Sales.Order for the most recently edited record(s).

We can easily get the most recent edit:

SELECT MAX(LastEditedWhen)

FROM Sales.Orders;

Now, how do we use this information to get the full details from Sales.Orders?

We could try storing this value in a variable, then using that variable in another query:

DECLARE @MaxDate DATETIME2;

SELECT @MaxDate = MAX(LastEditedWhen)

FROM Sales.Orders;

SELECT \*

FROM Sales.Orders

WHERE LastEditedWhen = @MaxDate;

This works, but it’s a little clunky. Also, what would happen if instead of the most recently edited record for the entire table, we wanted the most recent order by customer? As our need gets more complex, this looks less like a viable way to do this. We know that consolidated code is better. So let’s use a subquery to solve this in a single SELECT statement.

SELECT o.\*

FROM Sales.Orders o

JOIN ( SELECT MAX(LastEditedWhen) AS LastEditedWhen

FROM Sales.Orders ) AS LastEdit

ON o.LastEditedWhen = LastEdit.LastEditedWhen;

Notice that in the example above, I’m using my subquery like a table. I’ve given each column a name and I have given the subquery itself a name. I am then joining to that subquery just like any other table. We have essentially created a temporary view – a named query. When we do that, it is called an **inline view.**

**Inline views** are subqueries in a FROM clause of any DML statement.

Inline views are named with an alias in the FROM clause and can be accessed in SELECT, JOIN, and WHERE clauses (among others) similar to a normal view or table.

We are able to use multiple inline views in a single query. We can even nest an inline view inside another inline view.

# Common Table Expressions

## Using CTEs

<https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql>

Another way to create a temporary view of data for a query is using a common table expression, or CTE. CTEs are defined at the beginning of the query, then can be used throughout the query. Unlike inline views, CTEs can be used multiple times throughout the query, including in other CTEs.

### Example 1

Let’s modify the example above, this time using a CTE. Instead of just getting the last edit for the entire table, let’s find the last edit by each person who has edited this table (i.e. LastEditedBy). We’ll sort by the list to make it a little easier to know that we have it right:

WITH LastEdit AS (

SELECT LastEditedBy,

MAX(LastEditedWhen) AS LastEditedWhen

FROM Sales.Orders

GROUP BY LastEditedBy

)

SELECT o.\*

FROM Sales.Orders o

JOIN LastEdit le

ON o.LastEditedWhen = le.LastEditedWhen

AND o.LastEditedBy = le.LastEditedBy

ORDER BY LastEditedBy;

*NOTE: “WITH” is a commonly used keyword. To differentiate between a CTE and an option on a previous statement, it is mandatory that the previous statement be terminated with a semicolon. In older examples, you may occasionally “;WITH”.*

In this example, we added LastEditedBy to our CTE so that we could get the LastEditWhen for each one. This value can be different for each person, so we want to make sure we get all those values individually. When we join back to the Sales.Orders table, we need to use both columns in our join to avoid a square join.

### Example 2

In the above example, we used an implicit column list. We can also explicitly define them. Let’s look at another example, and provide an explicit column list. Let’s also chain a few CTEs together.

In this example, we would like to get the most recent order for each customer. When a customer has placed multiple orders on the same day, we will use the one with the lowest OrderID to narrow it down to one record per customer. This is similar to our query above, but we’re aggregating our set twice instead of once.

WITH LastOrderByCust ( CustomerID, OrderDate) AS (

-- Get the most recent order date by customer

SELECT CustomerID, MAX(OrderDate)

FROM Sales.Orders

GROUP BY CustomerID ),

MinOrder ( CustomerID, OrderID ) AS (

-- For all of the most recent customer orders, get the lowest order ID by customer

SELECT o.CustomerID, MIN(o.OrderID)

FROM Sales.Orders o

JOIN LastOrderByCust lobc

ON o.CustomerID = lobc.CustomerID

AND o.OrderDate = lobc.OrderDate

GROUP BY o.CustomerID

)

SELECT o.\*

FROM Sales.Orders o

JOIN MinOrder mo

ON o.OrderID = mo.OrderID;

Note the explicit column lists immediately following our CTE names. Also notice that our second CTE can reference the first. There’s a lot going on in that query, so let’s walk through it.

1. **Create LastOrderByCust**Here we are doing a simple aggregation. For each customer, we are getting the maximum order date.
2. **Create MinOrder**

Similar to our first CTE example, we want to join on both CustomerID and OrderDate to be sure that we’re getting the right order date by customer. Then, instead of returning all the records, we’re going to aggregate our set again, this time by minimum order id.

1. **Join to Sales.Orders**

Since we know that OrderID must be distinct, we can join just on OrderID. If we were using a value that was not guaranteed to be unique, we would have had to include CustomerID similar to our join in MinOrder.

## Recursive CTEs

It is possible to reference a CTE in its own definition. Doing so, allows us to create a recursive CTE. A common use for this technique is to query a hierarchical list.

When we create a recursive CTE, we must create at least one **anchor member** and at least one **recursive member**. An anchor member is a query that does not reference the CTE and a recursive member is a query that does. All anchor members must be defined before the first recursive member.

To see what that means, let’s look at an example.

**Business Concept (or Business RULE!!.. ah ha…) inherent in this WideWorldImporters Database Structure :**

**In Sales.Orders, some of the orders may be backorders. When that happens, the original order is stored as BackorderOrderID, which references another order in the Sales.Orders table. Some of those original orders may be backorders themselves, so we have a multi-level hierarchical structure.**

We’re worried that some of our customer orders may be getting backordered too frequently, so we’d like to see these deeply nested backorders. We don’t know how many levels deep this may go, so we’ll just keep selecting backorders until we reach records that don’t have a BackorderOrderID. To illustrate the effect, we will keep a list of all orders in the chain.

WITH Backorder ( OrderID, OrderChain, BackorderOrderID ) AS

(

-- Anchor member. Does not reference our CTE

SELECT OrderID,

CAST(OrderID AS VARCHAR(MAX)) AS OrderChain,

BackorderOrderID

FROM Sales.Orders

WHERE BackorderOrderID IS NULL

-- Recursive members. Must be joined in using UNION ALL. References our CTE.

UNION ALL

SELECT o.OrderID,

b.OrderChain + ', ' + CAST(o.OrderID AS VARCHAR(MAX)) AS OrderChain,

o.BackorderOrderID

FROM Sales.Orders o

JOIN Backorder b

ON o.BackorderOrderID = b.OrderID

)

SELECT \*

FROM Backorder

WHERE BackorderOrderID IS NOT NULL;

# Homework

## Understand

1. Using Sales.SpecialDeals, get the full deal details of the deal with the largest discount percentage.
2. Using Sales.OrderLines, write a query that will get the full order line details for the most expensive item on each order (i.e. highest UnitPrice \* Quantity value). Try doing it both with an inline view and a CTE.
3. Using Sales.Orders and Sales.OrderLines, get the full order line details for the most expensive line on the most recent order (i.e. max order date) for each customer. When multiple orders exist for a customer, get the most expensive order line from any one of them (i.e not one line per order).
4. Using the recursive CTE example above, modify the query to track how many levels deep the recursion is (i.e. start with 0 for an order with no backorder, then add 1 for each level above that).