Table expressions

T-SQL supports four types of table expressions

1) Derived Tables
2) Common Table Expression (CTE)
3) Views
4) Inline table valued functions

Table expressions are not physically materialized anywhere—they are virtual. When you query a table expression, the inner query gets unnested. In other words, the outer query and the inner query are merged into one query directly against the underlying objects. The benefits of using table expressions are typically related to logical aspects of your code and not to performance. For example, you can use table expressions to simplify your solutions by using a modular approach. Table expressions also help you circumvent certain restrictions in the language, such as the inability to refer to column aliases assigned in the *SELECT* clause in query clauses that are logically processed before the *SELECT* clause.

Derived tables (Table Subqueries)

-Are defined in the `FROM` clause of an outer query. Their scope of existence is the outer query. As soon as the outer query is finished the derived table is gone.

-You specify the query that defines the derived table within parentheses, followed by the *AS* clause and the derived table name. For example, the following code defines a derived table called *USACusts* based on a query that returns all customers from the United States, and the outer query selects all rows from the derived table:

all types of table expressions, a query must meet <u>THREE REQUIREMENTS</u> to be a valid inner query in a table-expression definition

Order is not guaranteed. A table expression is supposed to represent a relational table, and the rows in a relational table have no guaranteed order. Recall that this aspect of a relation stems from set theory. For this reason, standard SQL disallows an *ORDER BY* clause in queries that are used to define table expressions, unless the *ORDER BY* serves a purpose other than presentation. An example for such an exception is when the query uses the *OFFSET-FETCH* filter. T-SQL enforces similar restrictions, with similar exceptions—when *TOP* or *OFFSET-FETCH* is also specified. In the context of a query with the *TOP* or *OFFSET-FETCH* filter, the *ORDER BY* clause serves as part of the specification of the filter. If you use a query with *TOP* or *OFFSET- FETCH* and *ORDER BY* to define a table expression, *ORDER BY* is guaranteed to serve only the filtering-related purpose and not the usual presentation purpose. If the outer query against the table expression does not have a presentation *ORDER BY*, the output is not guaranteed to be returned in any particular order. See the "Views and the *ORDER BY* clause" section later in this chapter for more detail on this item (which applies to all types of table expressions).

All columns must have names. All columns in a table must have names; therefore, you must assign column aliases to all expressions in the *SELECT* list of the query that is used to define a table expression.

All column names must be unique. All column names in a table must be unique; therefore, a table expression that has multiple columns with the same name is invalid. Having multiple columns with the same name might happen when the query defining the table expression joins two tables that have a column with the same name. If you need to incorporate both columns in your table expression, they must have different column names. You can resolve this issue by assigning different column aliases to the two columns.

Assigning column aliases

One of the benefits of using table expressions is that, in any clause of the outer query, you can refer to column aliases that were assigned in the *SELECT* clause of the inner query. This behavior helps you get around the fact that you can't refer to column aliases assigned in the *SELECT* clause in query clauses that are logically processed prior to the *SELECT* clause (for example, *WHERE* or *GROUP BY*).

```
SELECT
  YEAR(orderdate) AS orderyear,
  COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY orderyear;
```

```
Invalid column name 'orderyear'.

Group By has a higher logical order so the alias orderyear is unknown at that moment
```

Solution

```
SELECT orderyear, COUNT (DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders) AS D
GROUP BY orderyear
```

Here we have orderyear defined in the FROM clause so we are able to use it now in the GROUP BY clause, then finally we can define it in the SELECT clause

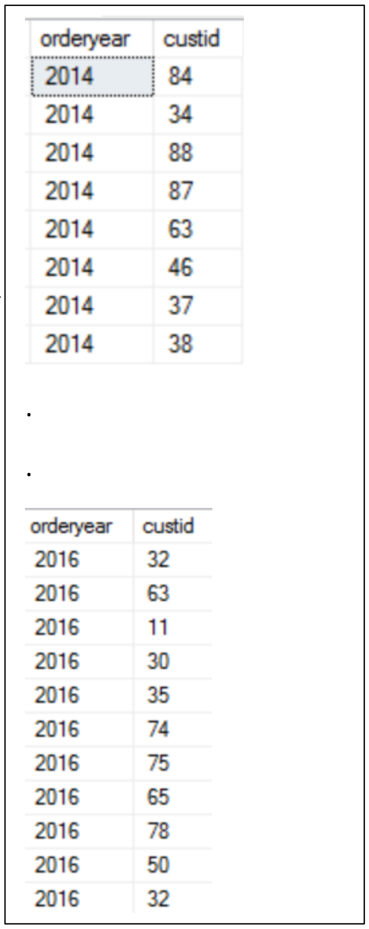
This code defines a derived table called *D* based on a query against the *Orders* table that returns the order year and customer ID from all rows. The *SELECT* list of the inner query uses the inline aliasing form to assign the alias *orderyear* to the expression *YEAR(orderdate)*. The outer query can refer to the *orderyear* column alias in both the *GROUP BY* and *SELECT* clauses, because as far as the outer query is concerned, it queries a table called *D* with columns called *orderyear* and *custid*.

Using arguments

In the query that defines a derived table, you can refer to arguments. The arguments can be local variables and input parameters to a routine, such as a stored procedure or function

For example, the following code declares and initializes a variable called *@empid*, and the query in the derived table *D* refers to that variable in the *WHERE* clause:

```
DECLARE @empid AS INT = 3;
SELECT orderyear,COUNT(DISTINCT custid) AS numcusts
FROM(SELECTYEAR(orderdate) AS orderyear, custid
     FROM Sales.Orders
     WHERE empid = @empid) AS D
GROUP BY orderyear;
```



orderyear	custid
2014	84
2014	34
2014	88
2014	87
2014	63
2014	46
2014	37
2014	38

.

.

orderyear	custid
2016	32
2016	63
2016	11
2016	30
2016	35
2016	74
2016	75
2016	65
2016	78
2016	50
2016	32

```
orderyear   numcusts
----------- ---------
2014        16
2015        46
2016        30
```

NESTING

Nested Queries:

```
SELECT orderyear, numcusts
FROM (SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
      FROM (SELECT YEAR(orderdate) AS orderyear, custid
            FROM Sales.Orders) AS D1
      GROUP BY orderyear) AS D2
WHERE numcusts > 70;
```

The purpose of the innermost derived table, *D1*, is to assign the column alias *orderyear* to the expression *YEAR(orderdate)*. The query against *D1* refers to *orderyear* in both the *GROUP BY* and *SELECT* clauses and assigns the column alias *numcusts* to the expression *COUNT(DISTINCT custid)*. The query against *D1* is used to define the derived table *D2*. The query against *D2* refers to *numcusts* in the *WHERE* clause to filter order years in which more than 70 customers were handled.

Common Table Expression

Common table expressions (CTEs) are another standard form of table expression similar to derived tables, yet with a couple of important advantages.

CTEs are defined by using a *WITH* statement and have the following general form:

WITH_<CTE_Name>[(<target_column_list>)]

AS

(

<inner_query_defining_CTE>

) <outer_query_against_CTE>;

The inner query defining the CTE must follow all requirements mentioned earlier to be valid to define a table expression. As a simple example, the following code defines a CTE called *USACusts* based on a query that returns all customers from the United States, and the outer query selects all rows from the CTE:

```
WITH  USACusts  AS
(
  SELECT  custid,  companyname
  FROM  Sales.Customers
  WHERE  country = N'USA'
)
SELECT *
FROM  USACusts;
```

As with derived tables, as soon as the outer query finishes, the CTE goes out of scope

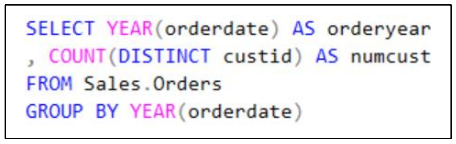
Assigning column aliases in CTEs

CTEs also support two forms of column aliasing:

1)**Inline**: For the inline form, specify *<expression> AS <column_alias>*;

2)**External**: For the external form, specify the target column list in parentheses immediately after the CTE name.

Here's an example of the inline form



```
SELECT YEAR(orderdate) AS orderyear
, COUNT(DISTINCT custid) AS numcust
FROM Sales.Orders
GROUP BY YEAR(orderdate)
```

```
WITH  C  AS
(
  SELECT  YEAR(orderdate)  AS  orderyear,  custid
  FROM  Sales.Orders
)
SELECT  orderyear,  COUNT(DISTINCT  custid)  AS  numcusts
FROM  C
GROUP  BY  orderyear;
```

And here's an example of the external form:

```
WITH  C (orderyear,  custid)  AS
(
  SELECT  YEAR(orderdate),  custid
  FROM  Sales.Orders
)
SELECT  orderyear, COUNT(DISTINCT  custid)  AS  numcusts
FROM  C
GROUP  BY  orderyear;
```

Multiple CTE's

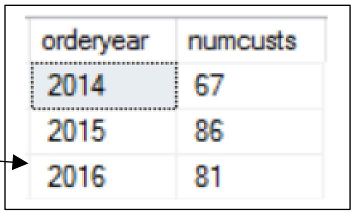
On the surface, the difference between derived tables and CTEs might seem to be merely semantic. However, the fact that you first name and define a CTE and then use it gives it several important advantages over derived tables. **One advantage** is that if you need to refer to one CTE from another, you don't nest them; rather, you separate them by commas. Each CTE can refer to all previously defined CTEs, and the outer query can refer to all CTEs.

For example, the following code is the CTE alternative to the nested derived tables approach in Listing 5-2:

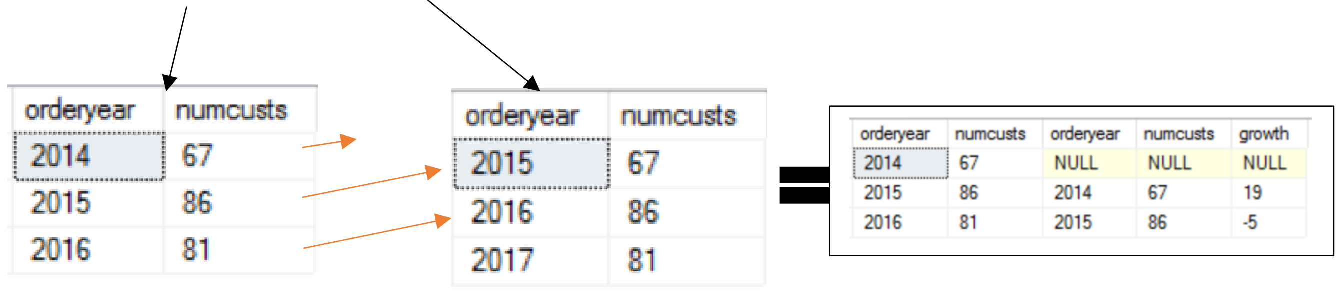
```
WITH C1 AS
(
  SELECT YEAR(orderdate) AS orderyear, custid
  FROM Sales.Orders
),
C2 AS
(
  SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C1
  GROUP BY orderyear
)
SELECT orderyear, numcusts
FROM C2
WHERE numcusts > 70;
```

The fact that a CTE is named and defined first and then queried **has another advantage**: as far as the *FROM* clause of the outer query is concerned, the CTE already exists; therefore, you can refer to multiple instances of the same CTE in table operators like joins. For example, the following code is the CTE alternative to the solution shown earlier in Listing 5-3 with derived tables:

```
WITH  YearlyCount  AS
(
  SELECT  YEAR(orderdate)  AS  orderyear,
          COUNT(DISTINCT  custid)  AS  numcusts
  FROM  Sales.Orders
  GROUP  BY  YEAR(orderdate)
)
SELECT  Cur.orderyear,
  Cur.numcusts  AS  curnumcusts,  Prv.numcusts  AS  prvnumcusts,
  Cur.numcusts - Prv.numcusts  AS  growth
FROM  YearlyCount  AS  Cur
  LEFT  OUTER  JOIN  YearlyCount  AS  Prv
  ON  Cur.orderyear  =  Prv.orderyear + 1;
```



orderyear	numcusts
2014	67
2015	86
2016	81



orderyear	numcusts
2014	67
2015	86
2016	81

orderyear	numcusts
2015	67
2016	86
2017	81

orderyear	numcusts	orderyear	numcusts	growth
2014	67	NULL	NULL	NULL
2015	86	2014	67	19
2016	81	2015	86	-5

VIEWS

Derived tables and CTEs have a single-statement scope, which means they are not reusable. Views and inline table-valued functions (inline TVFs) are two types of table expressions whose definitions are **stored as permanent objects in the database**, making them reusable. In most other respects, views and inline TVFs are treated like derived tables and CTEs.

For example, when querying a view or an inline TVF, SQL Server expands the definition of the table expression and queries the underlying objects directly, as with derived tables and CTEs.

As an example, the following code creates a view called *USACusts* in the *Sales* schema in the *TSQLV4* database, representing all customers from the United States:

```
DROP VIEW IF EXISTS Sales.USACusts;
GO
CREATE VIEW Sales.USACusts
AS
SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO ;
```

Note that the general recommendation to avoid using *SELECT ** has specific relevance in the context of views.

to avoid confusion, the best practice is to explicitly list the column names you need in the definition

After you create this view, you can query it much like you query other tables in the database:

```
SELECT custid, companyname
FROM Sales.USACust
```

Remember that a presentation *ORDER BY* clause is not allowed in the query defining a table expression because a relation isn't ordered. The only way to guarantee presentation order is to have an *ORDER BY* clause in the outer query. If you need to return rows from a view sorted for presentation purposes, you should specify a presentation *ORDER BY* clause in the outer query against the view, like this:

```
SELECT custid, companyname, region
FROM Sales.USACusts
ORDER BY region;
```

```
ALTER VIEW Sales.USACusts
AS
SELECT
  custid, companyname, contactname, contacttitle,
address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
ORDER BY region;
GO
```

VIEW OPTIONS

When you create or alter a view, you can specify view attributes and options as part of the view definition. In the header of the view, under the *WITH* clause, you can specify attributes such as *ENCRYPTION* and *SCHEMABINDING*, and at the end of the query you can specify *WITH CHECK OPTION*.

THE *ENCRYPTION* option

The *ENCRYPTION* option is available when you create or alter views, stored procedures, triggers, and user-defined functions (UDFs). The *ENCRYPTION* option indicates that SQL Server will internally store the text with the definition of the object in an obfuscated format. The obfuscated text is not directly visible to users through any of the catalog objects—only to privileged users through special means.

```
ALTER VIEW Sales.USACusts WITH ENCRYPTION
AS
SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
```

THE *SCHEMABINDING* option

The *SCHEMABINDING* option is available to views and UDFs; it binds the schema of referenced objects and columns to the schema of the referencing object. It indicates that referenced objects cannot be dropped and that referenced columns cannot be dropped or altered.

For example, alter the *USACusts* view with the *SCHEMABINDING* option:

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Now try to drop the *address* column from the *Customers* table:

```
ALTER TABLE Sales.Customers DROP COLUMN address;
```

You get the following error:

Without the *SCHEMABINDING* option, you would have been allowed to make such a schema change, as well as drop the *Customers* table altogether. This can lead to errors at run time when you try to query the view and referenced objects or columns do not exist. If you create the view with the *SCHEMABINDING* option, you can avoid these errors.

To support the *SCHEMABINDING* option, the object definition must meet a couple of requirements. The query is not allowed to use * in the *SELECT* clause; instead, you have to explicitly list column names. Also, you must use schema-qualified two-part names when referring to objects. Both requirements are actually good practices in general.

THE ***CHECK OPTION*** option

The purpose of *CHECK OPTION* is to prevent modifications through the view that conflict with the view's filter.

The query defining the view *USACusts* filters customers from the United States. The view is currently defined without *CHECK OPTION*. This means you can currently insert through the view customers from other countries, and you can update the country of existing customers through the view to one other than the United States.

For example, the following code successfully inserts a customer from the United Kingdom through the view:

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
 VALUES(
  N'Customer ABCDE', N'Contact ABCDE', N'Title ABCDE', N'Address ABCDE',
  N'London', NULL, N'12345', N'UK', N'012-3456789', N'012-3456789');
```

The row was inserted through the view into the *Customers* table. However, because the view filters only customers from the United States, if you query the view looking for the new customer, you get an empty set back:

```
SELECT custid, companyname, country
FROM Sales.USACusts
WHERE companyname = N'Customer ABCDE';
```

Similarly, if you update a customer row through the view, changing the country attribute to a country other than the United States, the update succeeds. But that customer information doesn't show up anymore in the view because it doesn't satisfy the view's query filter.

If you want to prevent modifications that conflict with the view's filter, add *WITH CHECK OPTION* at the end of the query defining the view:

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS
SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
WITH CHECK OPTION;
```

```
Now if you try to insert in to table it will not go through
```

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
 VALUES(
  N'Customer FGHIJ', N'Contact FGHIJ', N'Title FGHIJ', N'Address FGHIJ',
  N'London', NULL, N'12345', N'UK', N'012-3456789', N'012-3456789');
```

```
You get the following error
```

```
Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either specifies WITH CHECK
OPTION…The statement has been terminated.
```

Inline Table-Valued Functions

Inline TVFs are reusable table expressions that support input parameters. In most respects, except for the support for input parameters, inline TVFs are similar to views. For this reason, I like to think of inline TVFs as parameterized views, even though they are not formally referred to this way.

For example, the following code creates an inline TVF called *GetCustOrders* in the *TSQLV4*

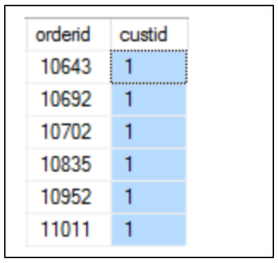
```
USE TSQLV4;
DROP FUNCTION IF EXISTS dbo.GetCustOrders;
GO
CREATE FUNCTION dbo.GetCustOrders
(@cid AS INT) RETURNS TABLE
AS
RETURN

SELECT orderid, custid, empid, orderdate, requireddate,
shippeddate, shipperid, freight, shipname, shipaddress, shipcity,
shipregion, shippostalcode, shipcountry
FROM Sales.Orders
WHERE custid = @cid;
```

This inline TVF accepts an input parameter called *@cid*, representing a customer ID, and returns all orders placed by the input customer. You query inline TVFs by using DML statements, which is the same way you query other tables. If the function accepts input parameters, you specify those in parentheses

Accessed Using

```
SELECT orderid, custid
FROM dbo.GetCustOrders(1) AS O;
```



orderid	custid
10643	1
10692	1
10702	1
10835	1
10952	1
11011	1