CS108: Advanced Database

- Database Programming

Lecture 06:

Subqueries

Overview

- Subqueries
- Table Expression
  - Derived Tables
  - Common Table Expression
  - View
  - Inline Table-Valued Function

Subquery

- SQL supports writing queries within queries, or nesting queries.
- The outermost query is a query whose result set is returned to the caller and is known as the outer query.
- The inner query is a query whose result is used by the outer query and is known as a subquery.
- A subquery can be either *self-contained* or *correlated*.
- A self-contained subquery has no dependency on the outer query that it belongs to, whereas a correlated subquery does.
- A subquery can be single-valued, multivalued, or table-valued.

Subquery Search Conditions

- A subquery usually appears as part of a search condition in the WHERE or HAVING clause.
- The type of subquery search conditions are:
  - *Subquery comparison test* - Compares the value of an expression with a single value produced by a subquery.
  - *Subquery set membership test* - Checks whether the value of an expression matches one of the set of values produced by a subquery.
  - *Existence test* - Tests whether a subquery produces any rows of query results.
  - *Quantified comparison test* - Compares the value of an expression with each of the sets of values produced by a subquery.

Self-Contained Subqueries

- Every subquery has an outer query that it belongs to.
- Self-contained subqueries are subqueries that are independent of the outer query that they belong to.

```
SELECT <SELECT list>
FROM   <SomeTable>
WHERE  <SomeColumn> {=, <>, <, <=, >, >=} (
            SELECT <single column>
            FROM   <SomeTable>
            WHERE  <condition that results in only one row returned>)
```

```
SELECT <SELECT list>
FROM   <SomeTable>
WHERE  <SomeColumn> IN (
            SELECT <single column>
            FROM   <SomeTable>
            [WHERE <condition>)]
```

Self-Contained Subqueries (cont.)

- Every subquery has an outer query that it belongs to.
- Self-contained subqueries are subqueries that are independent of the outer query that they belong to.

```
SELECT <SELECT list>
FROM   <SomeTable>
WHERE  <SomeColumn> EXIST (
           SELECT <single column>
           FROM   <SomeTable>
           WHERE  <condition>)
```

```
SELECT <SELECT list>
FROM   <SomeTable>
WHERE  <SomeColumn> {ALL, ANY, SOME} {=, <>, <, <=, >, >=} (
           SELECT <single column>
           FROM   <SomeTable>
           [WHERE <condition>)]
```

The Subquery Comparison Test

- A ***scalar*** subquery is a subquery that returns a single value - regardless of whether it is self-contained.
- 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 that we need to query the Orders table in the AdventureWorks database and return information about the order that has the maximum order ID in the table.

- We could accomplish the task by using a variable

```sql
DECLARE @maxid AS INT = (SELECT MAX(SalesOrderID)
                         FROM   Sales.SalesOrderHeader);

SELECT   SalesOrderID, OrderDate, SalesPersonID, CustomerID
FROM     Sales.SalesOrderHeader
WHERE    SalesOrderID = @maxid;
```

```
SalesOrderID OrderDate               SalesPersonID CustomerID
------------ ----------------------- ------------- -----------
75123        2014-06-30 00:00:00.000 NULL          18759
```

- We can substitute the technique that uses a variable with an embedded subquery.

```sql
SELECT   SalesOrderID, OrderDate, SalesPersonID, CustomerID
FROM     Sales.SalesOrderHeader
WHERE    SalesOrderID = (SELECT MAX(SalesOrderID)
                         FROM   Sales.SalesOrderHeader);
```

Self-Contained Scalar Subquery

- For a scalar subquery to be valid, it **MUST** return no more than one value.
- If a scalar subquery can return more than one value, it might fail at run time.

```
SELECT SalesOrderID
FROM   Sales.SalesOrderHeader
WHERE  SalesPersonID = (SELECT P.BusinessEntityID
                        FROM   Sales.SalesPerson AS P
                        WHERE  P.TerritoryID = 5);
```

```
...
71950
71951

(429 row(s) affected)
```

```
SELECT SalesOrderID
FROM   Sales.SalesOrderHeader
WHERE  SalesPersonID = (SELECT P.BusinessEntityID
                        FROM   Sales.SalesPerson AS P
                        WHERE  P.TerritoryID = 1);
```

```
Subquery returned more than 1 value. This is not permitted when the subquery
follows =, !=, <, <= , >, >= or when the subquery is used as an expression.
```

The Subquery Comparison Test

- If a scalar subquery returns no value, it returns a NULL.
- Recall that a comparison with a NULL yields UNKNOWN and that query filters do not return a row for which the filter expression evaluates to UNKNOWN.
- For example, the Sales.SalesPerson table currently has no person whose TerritoryID = 50; therefore, the following query returns an empty set.

```
(0 row(s) affected)
```

```
SELECT SalesOrderID
FROM   Sales.SalesOrderHeader
WHERE  SalesPersonID = (SELECT P.BusinessEntityID
                        FROM   Sales.SalesPerson AS P
                        WHERE  P.TerritoryID = 50);
```

The Set Membership Test (IN)

- A multivalued subquery is a subquery that returns multiple values as a single column.
- The subquery set membership test (IN) compares a single data value with a column of data values produced by a subquery and returns a TRUE result if the data value matches one of the values in the column.
- Note that the use of DISTINCT in the subquery isn’t strictly necessary. If the same record appears multiple times in the subquery results instead of only once, the outer query yields the same results.

- For example:

```
SELECT SalesOrderID
FROM   Sales.SalesOrderHeader
WHERE  SalesPersonID IN (SELECT P.BusinessEntityID
                         FROM   Sales.SalesPerson AS P
                         WHERE  P.TerritoryID = 1);
```

```
...
71943

(424 row(s) affected)
```

- Similarly, we can solve this problem using joins.

```
SELECT SalesOrderID
FROM   Sales.SalesOrderHeader AS S
JOIN   Sales.SalesPerson      AS P
  ON   S.SalesPersonID = P.BusinessEntityID
WHERE  P.TerritoryID   = 1
```

```
...
71943

(424 row(s) affected))
```

- Many querying problems that we can solve with either subqueries or joins.

Subqueries and Joins

- Many queries write using subqueries can also write as multitable queries, or joins.
- Conversely, many queries with subqueries cannot be translated into an equivalent join.
- For example, list the ID which SubTotal have above average SubTotal.

```
SELECT SalesOrderID
FROM   Sales.SalesOrderHeader
WHERE  SubTotal > (SELECT AVG(SubTotal) FROM Sales.SalesOrderHeader)
```

- In this case, the inner query is a summary query and the outer query is not, so there is no way the two queries can be combined into a single join.

The Set Membership Test (IN)

- As with any other predicate, we can negate the IN predicate with the NOT logical operator.
- For example, the following query returns customers who did not place any orders.

```
...
700
701

(701 row(s) affected)
```

```
SELECT CustomerID
FROM   Sales.Customer
WHERE  CustomerID NOT IN
                  (SELECT CustomerID FROM Sales.SalesOrderHeader)
```

- Note that best practice is to qualify the subquery to exclude NULL marks.
- If the subquery includes NULL marks then the finally result will be an empty set.

- Note that best practice is to qualify the subquery to exclude NULL marks.

```
SELECT CustomerID
FROM   Sales.Customer
WHERE  CustomerID NOT IN (
           SELECT CustomerID FROM Sales.SalesOrderHeader
           UNION ALL
           SELECT NULL )
```

```
CustomerID
-----------

(0 row(s) affected)
```

- Remember that T-SQL uses three-valued logic. The WHERE only accepts true.
- X NOT IN (n_1, n_2, …, *NULL*, …, n_N)
- = NOT (X IN (n_1, n_2, …, *NULL*, …, n_N))
- = NOT (X = n_1 OR X = n_2 OR *X = NULL* OR … OR X = n_N)
- = NOT (X = n_1 OR X = n_2 OR *UNKNOWN* OR … OR X = n_N)
- = (X ≠ n_1 AND X ≠ n_2 … AND X ≠ n_N) AND *NOT UNKNOWN*
- ∈ { *FALSE*, *UNKNOWN* } *(row alway be filtered out)*

Example: Set Membership Test (IN)

- Example: Using a nested SELECT to find missing records

```
SELECT    O.Description
FROM      Sales.SpecialOffer O
LEFT JOIN Sales.SpecialOfferProduct P
       ON P.SpecialOfferID = O.SpecialOfferID
WHERE     P.SpecialOfferID IS NULL;
```

```
Description
--------------------
Volume Discount over 60
(1 row(s) affected)
```

```
SELECT   Description
FROM     Sales.SpecialOffer O
WHERE    O.SpecialOfferID NOT IN
         (SELECT SpecialOfferID FROM Sales.SpecialOfferProduct);
```

```
Description
--------------------
Volume Discount over 60
(1 row(s) affected)
```

Example: Set Membership Test (IN)

- Example: Return all individual CustomerIDs that are missing between the minimum and maximum in the table.
- The Nums table is useful in this example.
  - The Nums table contains a sequence of integers, starting with 1, with no gaps.
  - To return all missing IDs from the Customer table, query the Nums table filter only
    - The numbers that are between the minimum and maximum in the Customer table and
    - The numbers do not appear in the set of CustomerIDs in the Customer table.

Example: Set Membership Test (IN)

- Example: Return all individual CustomerIDs that are missing between the minimum and maximum in the table.

```
SELECT N
FROM  (SELECT O1.N + O2.N * 10 + O3.N * 100 + O4.N*1000 + O5.N*10000 AS N
       FROM       (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O1(N)
       CROSS JOIN (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O2(N)
       CROSS JOIN (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O3(N)
       CROSS JOIN (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O4(N)
       CROSS JOIN (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O5(N)
       ) AS Num
WHERE  N BETWEEN (SELECT MIN(CustomerID) FROM Sales.Customer AS O)
             AND (SELECT MAX(CustomerID) FROM Sales.Customer AS O)
   AND N  NOT IN (SELECT     CustomerID  FROM Sales.Customer AS O);
```

```
N
-----------
...
10989
10999

(10298 row(s) affected)
```

Correlated Subqueries

- Correlated subqueries are subqueries that refer to attributes from the table that appears in the outer query.
- This means that the subquery is dependent on the outer query and cannot be invoked independently. (difficult to debug)

- For example, we want to know the first order of each customer

```sql
-- Get a list of customers and the date of their first order
SELECT      CustomerID, MIN(OrderDate) AS OrderDate
INTO        #MinOrderDates
FROM        Sales.SalesOrderHeader
GROUP BY    CustomerID;

-- Do something additional with that information
SELECT      A.CustomerID, A.SalesOrderID, A.OrderDate
FROM        Sales.SalesOrderHeader AS A
JOIN        #MinOrderDates AS T
  ON        A.CustomerID = T.CustomerID
 AND        A.OrderDate  = T.OrderDate
ORDER BY    A.CustomerID;

DROP TABLE #MinOrderDates;
```

```
...
30118       47378       2012-07-31 00:00:00.000
(19134 row(s) affected)
```

- If we want this to run in a single query, we must find a way to look up each individual customer.
- We can do this by making use of an *inner* query that performs a lookup based on the current CustomerID in the *outer* query.
- We then must return a value back out to the outer query so it can match things up based on the earliest order date.

```
SELECT    OUT.CustomerID, OUT.SalesOrderID, OUT.OrderDate
FROM      Sales.SalesOrderHeader AS OUT
WHERE     OUT.OrderDate = (
              SELECT MIN(IN.OrderDate)
              FROM   Sales.SalesOrderHeader AS IN
              WHERE  IN.CustomerID = OUT.CustomerID )
ORDER BY OUT.CustomerID;
```

Example: Correlated Subqueries

- Example: suppose that we need to query the Sales.SalesOrderHeader view and return for each order the percentage that the current order value is of the total values of all of the customer’s orders.

```
SELECT   O1.SalesOrderID, O1.CustomerID,
   CAST ( 100. * O1.SubTotal / (SELECT SUM(O2.SubTotal)
                                 FROM   Sales.SalesOrderHeader AS O2
                                 WHERE  O2.CustomerID = O1.CustomerID )
           AS NUMERIC(5, 2) ) AS [%]
FROM     Sales.SalesOrderHeader AS O1
ORDER BY CustomerID, SalesOrderID
```

```
...
58928         30118         12.84
65221         30118         8.71
71803         30118         11.74

(31465 row(s) affected)
```

The Existence Test (EXISTS)

- SQL supports a predicate called EXISTS that accepts a subquery as input and returns TRUE if the subquery returns any rows and FALSE otherwise.
- Notice that the EXISTS search condition doesn't really use the results of the subquery at all.
- It merely tests to see whether the subquery produces any results.
  - For this reason, SQL relaxes the rule that "subqueries must return a single column of data" and allows us to use the SELECT * form in the subquery of an EXISTS test.

- Example: find customers who did place any orders.
- Using EXISTS

```
SELECT C.CustomerID, C.AccountNumber
FROM   Sales.Customer AS C
WHERE  EXISTS (
           SELECT  *
           FROM    Sales.SalesOrderHeader AS O
           WHERE   O.CustomerID = C.CustomerID )
```

- Using IN

```
SELECT C.CustomerID, C.AccountNumber
FROM   Sales.Customer AS C
WHERE  C.CustomerID IN (SELECT O.CustomerID
                        FROM   Sales.SalesOrderHeader AS O
                        WHERE  O.CustomerID = C.CustomerID )
```

- Using JOIN

```
SELECT DISTINCT C.CustomerID, C.AccountNumber
FROM            Sales.Customer AS C
LEFT OUTER JOIN Sales.SalesOrderHeader AS O
             ON O.CustomerID = C.CustomerID
WHERE           O.SalesOrderID IS NOT NULL
```

The Existence Test (EXISTS)

- Another aspect of the EXISTS predicate that is interesting to note is that unlike most predicates in SQL, EXISTS uses two-valued logic and not three-valued logic.
  - TRUE - If the subquery returns any rows
  - FALSE - otherwise.

```
SELECT 1
WHERE  EXISTS (SELECT NULL)           -- (1 row(s) affected)

SELECT 1
WHERE  EXISTS (SELECT 1)              -- (1 row(s) affected)

SELECT 1
WHERE  EXISTS (SELECT TOP(0) 1)       -- (0 row(s) affected)

SELECT 1
WHERE  EXISTS (SELECT 1 WHERE -1 = 1) -- (0 row(s) affected)
```

Quantified Tests (ANY and ALL)

- SQL provides two quantified tests, ANY and ALL, that extend comparison operators, such as greater than.
- Both of these tests compare a data value with the column of data values produced by a subquery.
- For example: List the salespeople who have taken an order that represents more than 10 percent of their quota.

```
SELECT *
FROM   Sales.SalesPerson AS S
WHERE  (.1 * S.SalesQuota) < ANY (
                      SELECT TotalDue
                      FROM   Sales.SalesOrderHeader AS O
                      WHERE  O.SalesPersonID = S.BusinessEntityID )
```

- Conceptually, the main query tests each row one by one.

The ANY/SOME Test

- The ANY/SOME test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single test value with a column of data values produced by a subquery.
- To perform the test, SQL uses the specified comparison operator to compare the test value with each data value in the column, one at a time.
- If any of the individual comparisons yields a TRUE result, the ANY/SOME test returns a TRUE result.

The ANY/SOME Test (cont.)

- The ANSI/ISO SQL standard specifies these detailed rules describing the results of the ANY/SOME test when the test value is compared with the column of subquery results:
  - If the subquery produces an empty column of query results, the ANY/SOME test returns FALSE - no value is produced by the subquery for which the comparison test holds.
  - If the comparison test is TRUE for at least one of the data values in the column, then the ANY/SOME search condition returns TRUE - indeed some value is produced by the subquery for which the comparison test holds.

The ANY/SOME Test (cont.)

- The ANSI/ISO SQL standard specifies these detailed rules describing the results of the ANY/SOME test when the test value is compared with the column of subquery results:
  - If the comparison test is FALSE for every data value in the column, then the ANY/SOME search condition returns FALSE - no value which the comparison test holds.
  - If the comparison test is not TRUE for any data value in the column, but it is NULL (unknown) for one or more of the data values, then the ANY/SOME search condition returns NULL.

The ANY/SOME Test (cont.)

```
SELECT 'NULL'      --NULL
WHERE NOT EXISTS (
            SELECT 1
            WHERE  NOT (1 = ANY ( SELECT 2 UNION SELECT NULL))
                   OR  (1 = ANY ( SELECT 2 UNION SELECT NULL)))

SELECT 'NOT NULL' --NOT NULL
WHERE  EXISTS   (
            SELECT 1
            WHERE  NOT (1 = ANY ( SELECT 2 UNION SELECT 100))
                   OR  (1 = ANY ( SELECT 2 UNION SELECT 100)))
```

- If the comparison test is not TRUE for any data value in the column, but it is NULL (unknown) for one or more of the data values, then the ANY/SOME search condition returns NULL.

The ANY/SOME Test (cont.)

- The ANY comparison operator can be very tricky to use in practice, especially in conjunction with the inequality (<>) comparison operator.
- Here is an example that shows the problem: List the ID of all the SalesPerson who do not have an order.
- It’s tempting to express this query as shown in this example:

```
SELECT S.BusinessEntityID
FROM   Sales.SalesPerson AS S
WHERE  S.BusinessEntityID <> ANY (SELECT O.SalesPersonID
                                  FROM   Sales.SalesOrderHeader AS O)
```

```sql
SELECT S.BusinessEntityID
FROM   Sales.SalesPerson AS S
WHERE  S.BusinessEntityID <> ANY (SELECT O.SalesPersonID
                                  FROM   Sales.SalesOrderHeader AS O)
```

- The subquery produces the SalesPersonID of the orders, and therefore the query seems to be saying: Find each salesperson who is not the owner of **any (some)** order.
- But the query is: Find each salesperson who, for any order, is not the owner of that order.

```sql
SELECT S.BusinessEntityID
FROM   Sales.SalesPerson AS S
WHERE  NOT ( S.BusinessEntityID = ANY (
                          SELECT O.SalesPersonID
                          FROM   Sales.SalesOrderHeader AS O ) )
```

```sql
SELECT S.BusinessEntityID
FROM   Sales.SalesPerson AS S
WHERE  S.BusinessEntityID NOT IN (SELECT O.SalesPersonID
                                  FROM   Sales.SalesOrderHeader AS O )
```

The ALL Test

- Like the ANY test, the ALL test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single test value with a column of data values produced by a subquery.
- To perform the test, SQL uses the specified comparison operator to compare the test value with each data value in the column, one at a time.
- If all of the individual comparisons yield a TRUE result, the ALL test returns a TRUE result.

The ALL Test (cont.)

- The ANSI/ISO SQL standard specifies rules describing the results of the ALL test:
  - If the subquery produces an empty column of query results, the ALL test returns TRUE - the comparison test does hold for every value produced by the subquery.
  - If the comparison test is TRUE for every data value in the column, then the ALL search condition returns TRUE.

The ALL Test (cont.)

- The ANSI/ISO SQL standard specifies rules describing the results of the ALL test:
  - If the comparison test is FALSE for any data value in the column, then the ALL search condition returns FALSE.
  - If the comparison test is not FALSE for any data value in the column, but it is NULL for one or more of the data values, then the ALL search condition returns NULL.

Subqueries in the HAVING Clause

- Subqueries can also be used in the HAVING clause of a query.
- When a subquery appears in the HAVING clause, it works as part of the row group selection performed by the HAVING clause.
- Because the subquery is evaluated once for each row group, all outer references in the correlated subquery must be single-valued for each row group.
  - Either be a reference to a grouping column of the outer query
  - Or be contained within a column (aggregate) function.

Subqueries in the HAVING Clause

- Example: List the salespeople whose average TotalDue for selling product No. 910 is at least as big as that salesperson's overall average TotalDue.

```
SELECT   H1.SalesPersonID
FROM     Sales.SalesOrderHeader AS H1
JOIN     Sales.SalesOrderDetail AS D
  ON     H1.SalesOrderID = D.SalesOrderID
WHERE    D.ProductID = 910
GROUP BY H1.SalesPersonID
HAVING   AVG(H1.TotalDue) >= (
                    SELECT AVG(H2.TotalDue)
                    FROM   Sales.SalesOrderHeader AS H2
                    WHERE  H2.SalesPersonID = H1.SalesPersonID )
```

Subqueries Example

- Example: Returning Previous or Next Values

```
SELECT SalesOrderID, OrderDate, SalesPersonID, CustomerID,
       (SELECT MAX(SalesOrderID)
        FROM   Sales.SalesOrderHeader AS O2
        WHERE  O2.SalesOrderID < O1.SalesOrderID) AS PrevSalesOrderID
FROM   Sales.SalesOrderHeader AS O1
```

```
SalesOrderID OrderDate               SalesPersonID CustomerID  PrevSalesOrderID
------------ ----------------------- ------------- ----------- ----------------
43659        2011-05-31 00:00:00.000 279           29825       NULL
43660        2011-05-31 00:00:00.000 279           29672       43659
...
```

```
SELECT SalesOrderID, OrderDate, SalesPersonID, CustomerID,
       (SELECT MIN(SalesOrderID)
        FROM   Sales.SalesOrderHeader AS O2
        WHERE  O2.SalesOrderID > O1.SalesOrderID) AS NextSalesOrderID
FROM   Sales.SalesOrderHeader AS O1
```

```
...
75122        2014-06-30 00:00:00.000 NULL          15868       75123
75123        2014-06-30 00:00:00.000 NULL          18759       NULL
```

- Example: Running Aggregates

```
SELECT    YEAR(OrderDate) AS OrderYear, SUM(TotalDue) AS Qty
INTO      #OrderYear
FROM      Sales.SalesOrderHeader
WHERE     SalesPersonID = 282
GROUP BY YEAR(OrderDate)

SELECT    OrderYear, Qty,
          (SELECT SUM(O2.Qty)
           FROM   #OrderYear AS O2
           WHERE  O2.OrderYear <= O1.OrderYear) AS RunQty
FROM      #OrderYear AS O1
ORDER BY OrderYear

DROP TABLE #OrderYear
```

```
OrderYear   Qty                   RunQty
----------- --------------------- ---------------------
2011        1323328.6346          1323328.6346
2012        2070323.5014          3393652.136
2013        2112546.1213          5506198.2573
2014        1177338.401           6683536.6583
```

Table Expression

Table Expressions

- A table expression is a named query expression that represents a valid relational table.
- We can use table expressions in data manipulation statements much like we use other tables.
- Microsoft SQL Server supports four types of table expressions:
  - Derived tables
  - Common table expressions (CTEs)
  - Views
  - Inline table-valued functions (inline TVFs)

Derived Tables

- Derived tables (also known as 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.
- We specify the query that defines the derived table within parentheses, followed by the AS clause and the derived table name.

Derived Tables

- We specify the query that defines the derived table within parentheses, followed by the AS clause and the derived table name.
- Example:

```
SELECT *
FROM    (SELECT ProductID, Name, ListPrice
         FROM   Production.Product
         WHERE  Color = 'Black') AS BlackColorProduct
```

	ProductID	Name	ListPrice
1	317	LL Crankarm	0.00
2	318	ML Crankarm	0.00
3	319	HL Crankarm	0.00
4	322	Chainring	0.00
5	680	HL Road Frame - Black, 58	1431.50
6	708	Sport-100 Helmet, Black	34.99
7	722	LL Road Frame - Black, 58	337.22
8	723	LL Road Frame - Black, 60	337.22
9	724	LL Road Frame - Black, 62	337.22

Derived Tables

- A query must meet three requirements to be valid to define a table expression of any kind:
  - *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.
    - For this reason, standard SQL disallows an ORDER BY clause in queries that are used to define table expressions, unless the ORDER BY serves another purpose besides presentation (OFFSET-FETCH and TOP filter).

Derived Tables

- A query must meet three requirements to be valid to define a table expression of any kind:
  - *All columns must have names.*
    - All columns in a table must have names; therefore, we 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.

Using Arguments

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

```
DECLARE @Color AS NVARCHAR(100) = 'Black';

SELECT *
FROM   (SELECT ProductID, Name, ListPrice
        FROM   Production.Product
        WHERE  Color = @Color) AS BlackColorProduct;
```

Nesting

- If we need to define a derived table by using a query that itself refers to a derived table, we end up nesting derived tables.
- Nesting of derived tables is a result of the fact that a derived table is defined in the FROM clause of the outer query and not separately.
- Example:

```
OrderYear   NumCusts
----------- -----------
2013        11095
2014        10354
2012        3162
```

```
SELECT OrderYear, NumCusts
FROM   (SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
        FROM   (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
                FROM   Sales.SalesOrderHeader) AS D1
        GROUP BY OrderYear) AS D2
WHERE  NumCusts > 2000
```

Multiple References

- Another problematic aspect of derived tables stems from the fact that derived tables are defined in the FROM clause of the outer query and not prior to the outer query.
- As far as the FROM clause of the outer query is concerned, the derived table doesn’t exist yet; therefore, if we need to refer to multiple instances of the derived table, we can’t.
- Instead, we have to define multiple derived tables based on the same query.

Multiple References

```sql
SELECT Cur.OrderYear,
       Cur.NumCusts AS CurNumCusts, Prv.NumCusts AS PrvNumCusts,
       Cur.NumCusts - Prv.NumCusts AS Growth
FROM      (SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
           FROM   (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
                   FROM   Sales.SalesOrderHeader) AS D1
           GROUP BY OrderYear) AS Cur
LEFT JOIN (SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
           FROM   (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
                   FROM   Sales.SalesOrderHeader) AS D1
           GROUP BY OrderYear) AS Prv
       ON Cur.OrderYear = Prv.OrderYear + 1
ORDER BY  OrderYear
```

```
OrderYear   CurNumCusts PrvNumCusts Growth
----------- ----------- ----------- -----------
2011        1406        NULL        NULL
2012        3162        1406        1756
2013        11095       3162        7933
2014        10354       11095       -741
```

Common Table Expressions

- Common table expressions (CTEs) are another standard form of table expression very 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.

Assigning Column Aliases in CTEs

- CTEs also support two forms of column aliasing - inline and external.
- For the inline form, specify <expression> AS <column_alias>.
- For the external form, specify the target column list in parentheses immediately after the CTE name.

```
WITH <CTE_Name>
AS
(
    SELECT [<expression> AS <column_alias>] …
    …
)
<outer_query_against_CTE>;
```

Example: Find the TotalCost (TotalDue + Freight) for customers

```
WITH Orders AS (
   SELECT SalesOrderID, CustomerID, TotalDue + Freight AS TotalCost
   FROM   Sales.SalesOrderHeader
)
SELECT C.CustomerID, Orders.SalesOrderID, Orders.Total
FROM   Sales.Customer AS C
JOIN   Orders ON C.CustomerID = Orders.CustomerID;
```

```
CustomerID  SalesOrderID TotalCost
----------- ------------ ------------
29825       43659        23769.3323
29672       43660        1496.1564
29734       43661        37851.3542
...
```

- Or

```
WITH Orders ([Order ID], [Customer ID], Total)
AS (
   SELECT SalesOrderID, CustomerID, TotalDue + Freight AS TotalCost
   FROM   Sales.SalesOrderHeader
)
SELECT C.CustomerID, Orders.[Order ID], Orders.Total
FROM   Sales.Customer AS C
JOIN   Orders ON C.CustomerID = Orders.[Customer ID];
```

Views and Inline TVFs

- Derived tables and CTEs have a very limited scope, which is the single-statement scope. Derived tables and CTEs are *not reusable*.
- Views and inline table-valued functions (inline TVFs) are two *reusable* types of table expressions; their definitions are stored as database objects.
- In most other respects, views and inline TVFs are treated like derived tables and CTEs.

Views

- A view is a reusable table expression whose definition is stored in the database.
- Note that the general recommendation to avoid using SELECT * has specific relevance in the context of views.
    - The columns are enumerated in the compiled form of the view, and new table columns will **NOT** be automatically added to the view.

```
CREATE VIEW <view name> [(<column name list>)]
[WITH [ENCRYPTION] [[,] SCHEMABINDING] [[,] VIEW_METADATA]]
AS
  <SELECT statement>
[WITH CHECK OPTION][;]
```

Views

- A view is a reusable table expression whose definition is stored in the database.
- Example:

```
IF OBJECT_ID('Production.BLACKColorProduct') IS NOT NULL
   DROP VIEW Production.BLACKColorProduct;
GO

CREATE VIEW Production.BLACKColorProduct
AS
   SELECT ProductID, Name, ListPrice
   FROM   Production.Product
   WHERE  Color = 'Black';
GO

SELECT *
FROM   Production.BLACKColorProduct
```

Views and the ORDER BY Clause

- The query that we use to define a view *must meet* all requirements mentioned earlier with respect to table expressions in the context of derived tables.
- The view should *not guarantee* any order to the rows, all view columns must have names, and all column names must be unique.
- Remember that a presentation ORDER BY clause is *not* allowed in the query defining a table expression because there’s no order among the rows of a relational table.

- An attempt to create an ordered view is absurd because it violates fundamental properties of a relation as defined by the relational model.

```
ALTER VIEW Production.BLACKColorProduct
AS
    SELECT   ProductID, Name, ListPrice
    FROM     Production.Product
    WHERE    Color = 'Black'
    ORDER BY Name
GO
```

```
Msg 1033, Level 15, State 1, Procedure BLACKColorProduct, Line 73
The ORDER BY clause is invalid in views, inline functions, derived
tables, subqueries, and common table expressions, unless TOP, OFFSET
or FOR XML is also specified.
```

- The error message indicates that SQL Server allows the ORDER BY clause in three exceptional cases - when the TOP, OFFSET-FETCH option is used.

Views and the ORDER BY Clause

- Because T-SQL allows an ORDER BY clause in a view when TOP or OFFSET-FETCH is also specified, but we can not create “ordered views” in SQL Server.
- One of the ways to try to achieve this is by using TOP (100) PERCENT, like the following.

```
ALTER VIEW Production.BLACKColorProduct
AS
    SELECT TOP(100) PERCENT ProductID, Name, ListPrice
    FROM   Production.Product
    WHERE  Color = 'Black'
    ORDER BY Name
GO
```

- Even though the code is technically valid and the view is created, we should be aware that because the query is used to define a table expression, the ORDER BY clause here is only guaranteed to serve the logical filtering purpose for the TOP option.
- If we query the view and don't specify an ORDER BY clause in the outer query, presentation order is **not** guaranteed.

```
SELECT Name, ListPrice, ProductID
FROM   Production.BLACKColorProduct
```

```
Name                          ListPrice             ProductID
----------------------------- --------------------- -----------
LL Crankarm                   0.00                  317
ML Crankarm                   0.00                  318
HL Crankarm                   0.00                  319
Chainring                     0.00                  322
HL Road Frame - Black, 58     1431.50               680
...
```

View Option

- The ENCRYPTION option indicates that SQL Server will internally store the text with the definition of the object in an obfuscated format.
- 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 (base table or tables) cannot be dropped and that referenced columns cannot be dropped or altered.
- The purpose of CHECK OPTION is to prevent modifications through the view that conflict with the view’s filter - assuming that one exists in the query defining the view.

Inline Table-Valued Functions

- Inline TVFs are reusable table expressions that support input parameters. In all respects except for the support for input parameters, inline TVFs are similar to views.
- Inline TVF is the user defined function that return a table.

```
CREATE FUNCTION <function name>
    ( [ <@parameter name> [AS] <data type>
        [ = <default value> [READONLY]]
      [ ,...n ] ] )
RETURNS {<scalar type> | TABLE [(<table definition>)]}
    [ WITH [ENCRYPTION] | [SCHEMABINDING] |
      [RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT] |
      [EXECUTE AS { CALLER|SELF|OWNER|<'user name'> } ]
] [AS] { EXTERNAL NAME <external method> |
BEGIN
    [<function statements>]
    {RETURN <type as defined in RETURNS clause>|RETURN (<SELECT statement>)}
END }[;]
```

- For example, the following code creates an inline TVF called GetCustOrders in the database.

```
IF OBJECT_ID('GetCustOrders') IS NOT NULL
DROP FUNCTION GetCustOrders;
GO

CREATE FUNCTION GetCustOrders(@cid AS INT)
RETURNS TABLE
AS
   RETURN
      SELECT SalesOrderID, CustomerID,
             SalesPersonID, OrderDate, ShipMethodID,
             ShipDate, TerritoryID, Freight, SubTotal
      FROM   Sales.SalesOrderHeader
      WHERE  CustomerID = @cid;
GO

SELECT O.SalesOrderID, O.CustomerID
FROM   GetCustOrders(30085) AS O;
```

```
SalesOrderID CustomerID
------------ ----------
46937        30085
47968        30085
49044        30085
50196        30085
51796        30085
...
```

- As with other tables, we can refer to an inline TVF as part of a join.
- For example, the following query joins the inline TVF returning customer 30085's orders with the SalesOrderDetails table, matching customer 30085's orders with the related order lines.

```
SELECT O.SalesOrderID, O.CustomerID, OD.ProductID, OD.OrderQty
FROM   GetCustOrders(30085)     AS O
JOIN   Sales.SalesOrderDetail   AS OD
  ON   O.SalesOrderID = OD.SalesOrderID;
```

```
SalesOrderID CustomerID  ProductID   OrderQty
------------ ----------- ----------- --------
46937        30085       809         1
47968        30085       817         1
47968        30085       815         1
47968        30085       859         4
49044        30085       814         1
...
(22 row(s) affected)
```

The APPLY Operator

- The APPLY operator is a very powerful table operator.
- Like all table operators, this operator is used in the FROM clause of a query.
- The two supported types of APPLY operator are CROSS APPLY and OUTER APPLY - similar to cross join, outer join.
- CROSS APPLY implements only one logical query processing phase, whereas OUTER APPLY implements two (add a NULL marks to a resulting empty set).

The APPLY Operator

- The APPLY operator operates on two input tables, the second of which can be a table expression; we'll refer to them as the "left" and "right" tables.
- The right table is usually a derived table or an inline TVF.
- The CROSS APPLY operator implements one logical query processing phase - it applies the right table expression to each row from the left table and produces a result table with the unified result sets.

The CROSS APPLY Operator

- The CROSS APPLY operator is very similar to a cross join.

```
SELECT N
FROM   (SELECT O1.N + O2.N * 10 AS N
        FROM       (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O1(N)
       CROSS JOIN (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O2(N)
       ) AS Num;
```

```
N
-------
0
…
99
```

```
SELECT N
FROM   (SELECT O1.N + O2.N * 10 AS N
        FROM       (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O1(N)
       CROSS APPLY (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O2(N)
       ) AS Num;
```

- However, with the CROSS APPLY operator, the right table expression can represent a different set of rows per each row from the left table, unlike in a join.

- The CROSS APPLY is intended to enable joining to TVFs.

```
SELECT N
FROM   (SELECT O1.N + O2.N * 10 AS N
        FROM    (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O1(N)
        CROSS JOIN (VALUES(O1.N + 0), (O1.N + 1), (O1.N + 2))   AS O2(N)
       ) AS Num
```

```
SELECT N
FROM   (SELECT O1.N + O2.N * 10 AS N
        FROM    (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS O1(N)
        CROSS APPLY (VALUES(O1.N + 0), (O1.N + 1), (O1.N + 2))  AS O2(N)
       ) AS Num
```

- We can use a derived table in the right side, and in the derived table query refer to attributes from the left side.
- Alternatively, when we use an inline TVF, we can pass attributes from the left side as input arguments.

- The CROSS APPLY is intended to enable joining to TVFs.

```
CREATE FUNCTION CreateNUM(@N AS INT)
RETURNS TABLE
AS
   RETURN
      SELECT N
      FROM  (VALUES(@N+0),(@N+1),(@N+2),(@N+3),...,(@N+9)) AS O(N)
GO
```

```
SELECT O1.N + O2.N * 10
FROM        CreateNUM(0) AS O1
CROSS JOIN  CreateNUM(0) AS O2
```

```
SELECT O1.N + O2.N * 10
FROM        CreateNUM(0) AS O1
CROSS JOIN  CreateNUM(O1.N) AS O2
```

```
SELECT O1.N + O2.N * 10
FROM         CreateNUM(0) AS O1
CROSS APPLY  CreateNUM(O1.N) AS O2
```

- The CROSS APPLY is intended to enable joining to TVFs.

```
SELECT SOH.CustomerID, SOH.OrderDate, SOH.TotalDue, CRT.RunningTotal
FROM   Sales.SalesOrderHeader AS SOH
CROSS APPLY (
       SELECT SUM(TotalDue) AS RunningTotal
       FROM   Sales.SalesOrderHeader AS RT
       WHERE  RT.CustomerID    = SOH.CustomerID
         AND  RT.SalesOrderID <= SOH.SalesOrderID
      ) AS CRT
ORDER BY SOH.CustomerID, SOH.SalesOrderID
```

```
CustomerID  OrderDate               TotalDue              RunningTotal
----------- ----------------------- --------------------- -------------
11000       2011-06-21 00:00:00.000 3756.989              3756.989
11000       2013-06-20 00:00:00.000 2587.8769             6344.8659
11000       2013-10-03 00:00:00.000 2770.2682             9115.1341
11001       2011-06-17 00:00:00.000 3729.364              3729.364
11001       2013-06-18 00:00:00.000 2674.0227             6403.3867
11001       2014-05-12 00:00:00.000 650.8008              7054.1875
11002       2011-06-09 00:00:00.000 3756.989              3756.989
11002       2013-06-02 00:00:00.000 2535.964              6292.953
...
```

The OUTER APPLY Operator

- The OUTER APPLY operator adds a second logical phase that identifies rows from the left side for which the right table expression returns an empty set, and it adds those rows to the result table as outer rows with NULL marks in the right side’s attributes as placeholders.
- In a sense, this phase is similar to the phase that adds outer rows in a left outer join.

The OUTER APPLY Operator

- Use OUTER APPLY like a LEFT OUTER JOIN to return a row from the left even if there is nothing returned from the right.

```
SELECT   PRD.ProductID, S.SalesOrderID
FROM     Production.Product AS PRD
OUTER APPLY (
        SELECT TOP(2) SalesOrderID
        FROM     Sales.SalesOrderDetail AS SOD
        WHERE    SOD.ProductID = PRD.ProductID
        ORDER BY SalesOrderID) AS S
ORDER BY PRD.ProductID
```

```
ProductID   SalesOrderID
----------- ------------
1           NULL
2           NULL
3           NULL
4           NULL
316         NULL
317         NULL
...
```