CS108: Advanced Database

- Database Programming

Lecture 07:

Advanced Query

Windowing Functions

Window Functions

- A window function is a function that, for each row, computes a scalar result value based on a calculation against a subset of the rows from the underlying query.
- The subset of rows is known as a window and is based on a window descriptor that relates to the current row.
- The syntax for window functions uses a clause called OVER, in which we provide the window specification.

Window Functions

- A window function, simply think of the need to perform a calculation against a set and return a single value.
- A classic example would be aggregate calculations such as SUM, COUNT, and AVG, but there are others as well, such as ranking functions.
- Different from grouped queries which will lose the detail, a window function has an OVER clause that defines the set of rows for the function to work with.

Window Functions

- Different from subqueries which start from a fresh view of the data, a window function is applied to a subset of rows from the underlying query’s result set - not a fresh view of the data.
- Another benefit of window functions is the ability to define order, when applicable, as part of the specification of the calculation, without conflicting with relational aspects of the result set.

Window Functions

- The syntax of a window function is

```
<window function> OVER (
   [ PARTITION BY <value_expression> [,... n] ]
   [ ORDER BY <order_by_expression> [ ASC | DESC ][,...n] ]
   [ { ROWS | RANGE }
     { BETWEEN { UNBOUNDED PRECEDING
                 | <unsigned_value> { PRECEDING | FOLLOWING }
                 | CURRENT ROW }
       AND     { UNBOUNDED PRECEDING
                 | <unsigned_value> { PRECEDING | FOLLOWING }
                 | CURRENT ROW }
     }
   ]
)
```

Example: RunningTotal

- For example, the following query uses a window aggregate function to compute the running total values for each employee

```
SELECT SalesPersonID, OrderDate, TotalDue,
       SUM(TotalDue) OVER (PARTITION BY SalesPersonID
                           ORDER BY OrderDate
                           ROWS BETWEEN UNBOUNDED PRECEDING
                                    AND CURRENT ROW) AS RunningTotal
FROM   Sales.SalesOrderHeader
WHERE  SalesPersonID IS NOT NULL
```

```
SalesPersonID OrderDate      TotalDue             RunningTotal
------------- -------------- -------------------- --------------------
274           2011-07-01     23130.2957           23130.2957
274           2011-08-01     2297.0332            25427.3289
...
274           2014-05-01     2613.7411            1235934.4451
275           2011-05-31     6893.2549            6893.2549
…
```

Window Functions

- The window specification in the OVER clause has three main parts: Partitioning, Ordering, and Framing.
  - An empty OVER() clause exposes to the function a window made of all rows from the underlying query's result set.
- The window partition clause (PARTITION BY) restricts the window to the subset of rows from the underlying query's result set that share the *same values* in the partitioning *columns* as in the current row.

Window Functions

- The window order clause (ORDER BY) defines ordering in the window, but don't confuse this with presentation ordering; the window ordering is what gives meaning to *window framing*.
- After order has been defined in the window, a window frame clause (ROWS BETWEEN <top delimiter> AND <bottom delimiter>) filters a frame, or a subset, of rows from the window partition between the two specified delimiters.

Window Framing

- Three key phrases: UNBOUNDED PRECEDING, UNBOUNDED FOLLOWING and CURRENT ROW to define a window framing.
- For example, imagine that we have 100 rows in the results and we are viewing the rows from the perspective of row 10

Frame Definition	Rows in Frame for Row 10
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	Rows 1 - 10
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING	Rows 10 - 100
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	Rows 1 - 100

- UNBOUNDED PRECEDING means every row up to row 10
- UNBOUNDED FOLLOWING means every row greater than row 10

Window Framing

- We can also specify an offset, or the actual number of rows removed from the current row
- For example, imagine that we have 100 rows in the results and we are viewing the rows from the perspective of row 10

Frame Definition	Rows in Frame for Row 10
ROWS BETWEEN 3 PRECEDING AND CURRENT ROW	Rows 7 - 10
ROWS BETWEEN CURRENT ROW AND 5 FOLLOWING	Rows 10 - 15
ROWS BETWEEN UNBOUNDED PRECEDING AND 5 FOLLOWING	Rows 1 - 15
ROWS BETWEEN 3 PRECEDING AND 5 FOLLOWING	Rows 5 - 15

Window Functions

- Note that because the starting point of a window function is the underlying query's result set, and the underlying query's result set is generated only when we reach the SELECT phase, window functions are *allowed* only in the SELECT and ORDER BY clauses of a query.
- If we need to refer to a window function in an earlier logical query processing phase (such as WHERE), we need to use a table expression.

Ranking Window Functions

- Ranking window functions allow us to rank each row in respect to others in several different ways.
  - SQL Server supports four ranking functions: ROW_NUMBER, RANK, DENSE_RANK, and NTILE.
- For example:

```
SELECT    SalesOrderID, CustomerID, SubTotal,
          ROW_NUMBER() OVER(ORDER BY SubTotal) AS RowNum,
          RANK()       OVER(ORDER BY SubTotal) AS Rank,
          DENSE_RANK() OVER(ORDER BY SubTotal) AS DenseRank,
          NTILE(100)   OVER(ORDER BY SubTotal) AS Bucket
FROM      Sales.SalesOrderHeader
WHERE     CustomerID BETWEEN 11000 AND 11100
ORDER BY SubTotal
```

SalesOrderID	CustomerID	SubTotal	RowNum	Rank	DenseRank	Bucket
60531	11091	2.29	1	1	1	1
64737	11019	2.29	2	1	1	1
68475	11098	4.99	3	3	2	1
56816	11091	4.99	4	3	2	1
57494	11091	6.28	5	5	3	2
69583	11049	6.28	6	5	3	2
64472	11067	6.28	7	5	3	2
68413	11012	6.28	8	5	3	2
74538	11065	6.28	9	5	3	3
74658	11023	6.28	10	5	3	3
73431	11091	7.28	11	11	4	3
57639	11019	7.28	12	11	4	3
62413	11078	7.28	13	11	4	4
58717	11074	8.99	14	14	5	4
56910	11019	8.99	15	14	5	4
54436	11079	9.99	16	16	6	4
53993	11078	11.94	17	17	7	5
...						
44356	11099	3399.99	303	282	146	99
44220	11060	3399.99	304	282	146	99
44323	11072	3399.99	305	282	146	99
43765	11010	3399.99	306	282	146	100
43701	11003	3399.99	307	282	146	100
45389	11090	3578.27	308	308	147	100

(308 row(s) affected)

Ranking Window Functions

- ROW_NUMBER function assigns incrementing sequential integers to the rows in the result set of a query.
- RANK or DENSE_RANK function produce the same ranking value in all rows that have the same logical ordering value.
  - RANK indicates how many rows have a lower ordering value.
  - DENSE_RANK indicates how many distinct ordering values are lower.
- NTILE function allows us to associate the rows in the result with equally sized groups of rows by assigning a tile number to each row.

Ranking Window Functions

- Ranking functions support window partition clauses.
  - Window partitioning restricts the window to only those rows that share the same values in the partitioning attributes.
- For example,

```
SELECT SalesOrderID, CustomerID, SubTotal,
       ROW_NUMBER() OVER(PARTITION BY CustomerID
                         ORDER BY SubTotal) AS RowNum
FROM   Sales.SalesOrderHeader
```

```
SalesOrderID CustomerID  SubTotal              RowNum
------------ ----------- --------------------- --------------------
51522        11000       2341.97               1
57418        11000       2507.03               2
43793        11000       3399.99               3
72773        11001       588.96                1
51493        11001       2419.93               2
43767        11001       3374.99               3
...
```

Offset Window Functions

- Offset window functions allow us to return an element from a row that is at a certain offset from the current row or from the beginning or end of a window frame.
- The LAG and LEAD functions support *window partition* and *window order* clauses.
  - There's no relevance to window framing here.
  - These functions allow us to obtain an element from a row that is at a certain offset from the current row within the partition, based on the indicated ordering.

The LAG and LEAD Functions

- The LAG and LEAD functions.
  - The LAG function looks before the current row, and the LEAD function looks ahead.
  - The first argument to the functions (which is mandatory) is the element we want to return;
  - The second argument (optional) is the offset (1 if not specified);
  - The third argument (optional) is the default value to return in case there is no row at the requested offset (NULL if not specified).

- For example, the following query uses the LAG function to return the value of the previous customer's order and the LEAD function to return the value of the next customer's order.

```
SELECT SalesOrderID, CustomerID, SubTotal,
       LAG(SubTotal)  OVER (PARTITION BY CustomerID
                    ORDER BY OrderDate, SalesOrderID) AS PreSubTotal,
       LEAD(SubTotal) OVER (PARTITION BY CustomerID
                   ORDER BY OrderDate, SalesOrderID) AS NextSubTotal
FROM   Sales.SalesOrderHeader
```

```
SalesOrderID    CustomerID    SubTotal       PreSubTotal      NextSubTotal
--------------- ------------- -------------- ---------------- --------------
43793           11000         3399.99        NULL             2341.97
51522           11000         2341.97        3399.99          2507.03
57418           11000         2507.03        2341.97          NULL
43767           11001         3374.99        NULL             2419.93
51493           11001         2419.93        3374.99          588.96
72773           11001         588.96         2419.93          NULL
43736           11002         3399.99        NULL             2294.99
51238           11002         2294.99        3399.99          2419.06
53237           11002         2419.06        2294.99          NULL
...
```

The {FIRST, LAST}_VALUE Functions

- The FIRST_VALUE and LAST_VALUE functions allow us to return an element from the first and last rows in the *window frame*, respectively.
  - If we want the element from the *first row* in the window partition, use FIRST_VALUE with the window frame extent ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.
  - If we want the element from the *last row* in the window partition, use LAST_VALUE with the window frame extent ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING.

- For example

```
SELECT CustomerID, SalesOrderID, SubTotal,
       FIRST_VALUE(SubTotal) OVER (PARTITION BY CustomerID
            ORDER BY OrderDate, SalesOrderID
             ROWS BETWEEN UNBOUNDED PRECEDING
                      AND CURRENT ROW) AS FirstVal,
       LAST_VALUE(SubTotal) OVER (PARTITION BY CustomerID
            ORDER BY OrderDate, SalesOrderID
             ROWS BETWEEN CURRENT ROW
                      AND UNBOUNDED FOLLOWING) AS LastVal
FROM   Sales.SalesOrderHeader
ORDER BY CustomerID, OrderDate, SalesOrderID
```

CustomerID	SalesOrderID	SubTotal	FirstVal	LastVal
11000	43793	3399.99	3399.99	2507.03
11000	51522	2341.97	3399.99	2507.03
11000	57418	2507.03	3399.99	2507.03
11001	43767	3375.99	3375.99	588.96
11001	51493	2419.93	3375.99	588.96
11001	72773	588.96	3375.99	588.96
11002	43736	3399.99	3399.99	2419.06
11002	51238	2294.99	3399.99	2419.06
11002	53237	2419.06	3399.99	2419.06
...				

Aggregate Window Functions

- Window aggregate functions allow us to return summary values like SUM, MAX, COUNT and AVG along with details.
- An OVER clause with empty parentheses exposes a window of all rows
  - SUM(val) OVER() returns the grand total of all values.
- A window partition clause, we expose a restricted window to the function, with only those rows from the underlying query's result set that share the same values
  - SUM(val) OVER(PARTITION BY CustomerID) returns the total values for the current customer.

- For example, the following query return grand total of all order values, as well as the customer total.

```
SELECT CustomerID, SalesOrderID, SubTotal,
       SUM(SubTotal) OVER() AS TotalValue,
       SUM(SubTotal) OVER(PARTITION BY CustomerID) AS CustTotalValue
FROM   Sales.SalesOrderHeader
ORDER BY CustomerID, OrderDate
```

```
CustomerID  SalesOrderID  SubTotal     TotalValue          CustTotalValue
----------- ------------- ------------ ------------------- ---------------
11000       43793         3399.99      109847038.6439      8248.99
11000       51522         2341.97      109847038.6439      8248.99
11000       57418         2507.03      109847038.6439      8248.99
11001       43767         3375.99      109847038.6439      6384.88
11001       51493         2419.93      109847038.6439      6384.88
11001       72773         588.96       109847038.6439      6384.88
11002       43736         3399.99      109847038.6439      8114.04
11002       51238         2294.99      109847038.6439      8114.04
11002       53237         2419.06      109847038.6439      8114.04
11003       43701         3399.99      109847038.6439      8139.29
...
```

Aggregate Window Functions

- An advantage of window functions is that by enabling us to return *detail* elements and *aggregate* them in the same row
  - They also enable us to write expressions that mix detail and aggregates.
- For example,

```
SELECT CustomerID, SalesOrderID, SubTotal,
   100. * SubTotal / SUM(SubTotal) OVER() AS [% All],
   100. * SubTotal / SUM(SubTotal) OVER(PARTITION BY CustomerID)
                                                AS [% Cust]
FROM   Sales.SalesOrderHeader
ORDER BY CustomerID, OrderDate
```

```
CustomerID  SalesOrderID SubTotal    % All                    % Cust
----------- ------------ ----------- ---------------------- ------------------
11000       43793        3399.99     0.003095204060094      41.217045965627307
11000       51522        2341.97     0.002132028344971      28.390990897067398
...
```

Aggregate Window Functions

- Window ordering and framing for aggregate functions allows for more sophisticated calculations such as running and moving aggregates, and others.
- For example,

```
SELECT CustomerID, OrderDate, SubTotal,
       SUM(SubTotal) OVER(PARTITION BY CustomerID
                          ORDER BY OrderDate
                          ROWS BETWEEN UNBOUNDED PRECEDING
                                   AND CURRENT ROW) AS [RunTotal]
FROM   Sales.SalesOrderHeader
```

```
CustomerID  OrderDate               SubTotal            RunTotal
----------- ----------------------- ------------------- --------------------
11000       2011-06-21 00:00:00.000 3399.99             3399.99
11000       2013-06-20 00:00:00.000 2341.97             5741.96
11000       2013-10-03 00:00:00.000 2507.03             8248.99
11001       2011-06-17 00:00:00.000 3375.99             3375.99
11001       2013-06-18 00:00:00.000 2419.93             5795.92
...
```

Pivoting, Grouping Data

Pivoting Data

- Pivoting data involves rotating data from a state of rows to a state of *columns*, possibly aggregating values along the way.
- A pivoted query displays the values of one column as *column headers* instead.
- There are two techniques to write pivoted queries : CASE and PIVOT.

Store	Product	Sales
A	TV	2
A	TV	4
B	TV	6
B	DVD	8

Store	Avg(Sales) for TV	Avg(Sales) for DVD
A	3	(Empty)
B	6	8

```
CREATE TABLE Orders (
    OrderID    INT NOT NULL,
    OrderDate DATE NOT NULL,
    EmpID      INT NOT NULL,
    CustID     VARCHAR(5) NOT NULL,
    Qty        INT NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(OrderID)
);

INSERT INTO Orders(OrderID, OrderDate, EmpID, CustID, Qty)
VALUES
    (30001, '20070802', 3, 'A', 10),
    (10001, '20071224', 2, 'A', 12),
    (10005, '20071224', 1, 'B', 20),
    (40001, '20080109', 2, 'A', 40),
    (10006, '20080118', 1, 'C', 14),
    (20001, '20080212', 2, 'B', 12),
    (40005, '20090212', 3, 'A', 10),
    (20002, '20090216', 1, 'C', 20),
    (30003, '20090418', 2, 'B', 15),
    (30004, '20070418', 3, 'C', 22),
    (30007, '20090907', 3, 'D', 30);
```

OrderID	OrderDate	EmpID	CustID	Qty
10001	2007-12-24	2	A	12
10005	2007-12-24	1	B	20
10006	2008-01-18	1	C	14
20001	2008-02-12	2	B	12
20002	2009-02-16	1	C	20
30001	2007-08-02	3	A	10
30003	2009-04-18	2	B	15
30004	2007-04-18	3	C	22
30007	2009-09-07	3	D	30
40001	2008-01-09	2	A	40
40005	2009-02-12	3	A	10

```
SELECT EmpID, CustID, SUM(Qty) AS SumQty
FROM   Orders
GROUP BY EmpID, CustID;
```

EmpID	CustID	SumQty
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30

Pivoting Data

```
SELECT EmpID, CustID, SUM(Qty) AS SumQty
FROM   Orders
GROUP BY EmpID, CustID;
```

```
EmpID    CustID   SumQty
-------- -------- -----------
2        A        52
3        A        20
1        B        20
2        B        27
1        C        34
3        C        22
3        D        30
```

- A pivoted query displays the values of one column as column headers instead.
- For example, pivoted view of total quantity per employee (on Rows) and customer (on Columns)

empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30

Pivoting Data

- The PIVOT involves the following three logical phases. P1: Grouping, P2: Spreading and P3: Aggregating
  - *Grouping*: The first phase groups the rows. (implicitly based on all columns that weren't mentioned in PIVOT's inputs)
  - *Spreading*: PIVOT's second phase (P2) spreads values to their corresponding target columns.
  - *Aggregating*: PIVOT's third phase (P3) applies the specified aggregate function on top of each spreaded columns, generating the result columns.

Pivoting with Standard SQL

- The standard solution for pivoting handles all three phases involved in a very straightforward manner.
  - *Grouping*: The grouping phase is achieved with a GROUP BY clause; in this case, GROUP BY EmpID.
  - *Spreading*: The spreading phase is achieved in the SELECT clause with a CASE expression for each target column.

```
CASE WHEN CustID = 'A' THEN Qty END
CASE WHEN CustID = 'B' THEN Qty END
...
```

  - *Aggregating*: The aggregation phase is achieved by applying the relevant aggregate function (SUM, in this case) to the result of each CASE expression.

Pivoting with Standard SQL

- The standard solution for pivoting handles all three phases involved in a very straightforward manner.

```
SELECT EmpID,
       SUM(CASE WHEN CustID = 'A' THEN Qty END) AS A,
       SUM(CASE WHEN CustID = 'B' THEN Qty END) AS B,
       SUM(CASE WHEN CustID = 'C' THEN Qty END) AS C,
       SUM(CASE WHEN CustID = 'D' THEN Qty END) AS D
FROM   Orders
GROUP BY EmpID;
```

```
EmpID       A           B           C           D
----------- ----------- ----------- ----------- -----------
1           NULL        20          34          NULL
2           52          27          NULL        NULL
3           20          NULL        22          30
```

Pivoting with the PIVOT Operator

- The PIVOT operator operates in the context of the FROM clause of a query like other table operators
- It operates on a source table or table expression, pivots the data, and returns a result table.
- The general form of a query with the PIVOT operator is shown as following.

```
SELECT <grouping col>, <pivoted value1>, <pivoted value2>
FROM
     ( SELECT <grouping col>, <value column>, <pivoted column> )
       AS <query alias>
       PIVOT (<aggregate function>(<value column>)
         FOR <pivoted column> IN (<pivoted value1>, <value2>,...) )
       AS <pivot alias>
[ORDER BY <grouping col>]
```

aggregation element

spreading element

- For example: The aggregate function (SUM), aggregation element (Qty), spreading element (CustID), and the list of target column names (A, B, C, D).

```
SELECT EmpID, A, B, C, D
FROM (  SELECT EmpID, CustID, Qty
        FROM   Orders ) AS D
        PIVOT( SUM(Qty) FOR CustID IN (A, B, C, D) ) AS P;
```

- It is important to note that with the PIVOT operator, we do not explicitly specify the grouping elements.
  - The PIVOT operator figures out the grouping elements implicitly as all attributes from the source table (or table expression) that were not specified as either the spreading element or the aggregation element.

```
SELECT EmpID, A, B, C, D
FROM (  SELECT EmpID, CustID, Qty
        FROM   Orders ) AS D
        PIVOT( SUM(Qty) FOR CustID IN (A, B, C, D) ) AS P;
```

- Instead of operating directly on the Orders table, the PIVOT operator operates on a derived table called D that includes only the pivoting elements EmpID, CustID, and Qty.

```
SELECT EmpID, A, B, C, D
FROM   Orders PIVOT( SUM(Qty) FOR CustID IN(A, B, C, D) ) AS P;
```

```
EmpID       A           B           C           D
----------- ----------- ----------- ----------- -----------
2           12          NULL        NULL        NULL
1           NULL        20          NULL        NULL
1           NULL        NULL        14          NULL
...
(11 row(s) affected)
```

- The attributes (OrderID, OrderDate, and EmpID) are all considered the grouping elements.

Unpivoting Data

- Unpivoting is a technique to rotate data from a state of columns to a state of rows.
- Usually it involves querying a pivoted state of the data, producing from each source row multiple result rows, each with a different source column value.

from:

RowID	Col A	Col B	Col C	Col D
Row0	A1	B1	C1	D1
Row1	A2	B2	C2	D2
Row2	A3	B3	C3	D3

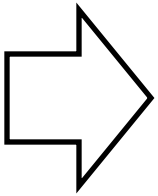
to:

RowIDs	ColumnNames	ColumnValues
Row0	ColA	A1
Row0	ColB	B1
Row0	ColC	C1
Row1	ColA	A2
Row1	ColB	B2
Row1	ColC	C2
Row2	ColA	A3
Row2	ColB	B3
Row2	ColC	C3

```sql
CREATE TABLE EmpCustOrders
(
    EmpID INT NOT NULL CONSTRAINT PK_EmpCustOrders PRIMARY KEY,
    A VARCHAR(5) NULL,
    B VARCHAR(5) NULL,
    C VARCHAR(5) NULL,
    D VARCHAR(5) NULL
);

INSERT INTO EmpCustOrders(EmpID, A, B, C, D)
SELECT EmpID, A, B, C, D
FROM (  SELECT EmpID, CustID, Qty
        FROM   Orders) AS D
        PIVOT( SUM(Qty) FOR CustID IN(A, B, C, D) ) AS P;
```

EmpID	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30



EmpID	CustID	SumQty
1	B	20
1	C	34
2	A	52
2	B	27
3	C	22
3	D	30
3	A	20

Unpivoting Data

- UNPIVOT is the inverse of PIVOT, rotating data from columns to rows, involving, U1: Generating Copies, U2: Extracting Elements and U3: Removing Rows with NULLs.
  - *Generating Copies*: The first step (U1) generates copies of the rows from the source table expression as an input
  - *Extracting Elements*: The second step (U2) extracts the value from the source column corresponding to the unpivoted element that the current copy of the row represents.
  - *Removing Rows with NULLs*: UNPIVOT's third and final step (U3) is to remove rows with NULLs in the result value column

Unpivoting with Standard SQL

- Generating Copy: The first step in the solution involves producing multiple copies of each source row - one for each column that we need to unpivot. (Using CROSS JOIN)

```
SELECT *
FROM   EmpCustOrders
CROSS JOIN (VALUES('A'),('B'),('C'),('D')) AS Custs(CustID);
```

```
EmpID       A     B     C     D     CustID
----------- ----- ----- ----- ----- ------
1           NULL  20    34    NULL  A
1           NULL  20    34    NULL  B
1           NULL  20    34    NULL  C
1           NULL  20    34    NULL  D
2           52    27    NULL  NULL  A
2           52    27    NULL  NULL  B
2           52    27    NULL  NULL  C
2           52    27    NULL  NULL  D
3           20    NULL  22    30    A
3           20    NULL  22    30    B
3           20    NULL  22    30    C
3           20    NULL  22    30    D
```

Unpivoting with Standard SQL

- Extracting Elements: The second step in the solution is to produce a column that returns the value from the column that corresponds to the customer represented by the current copy.

```
SELECT EmpID, CustID,
       CASE CustID
         WHEN 'A' THEN A
         WHEN 'B' THEN B
         WHEN 'C' THEN C
         WHEN 'D' THEN D
       END AS Qty
FROM  EmpCustOrders
CROSS JOIN (VALUES('A'),
                  ('B'),
                  ('C'),
                  ('D'))
        AS Custs(CustID);
```

EmpID	A	B	C	D	CustID
1	NULL	20	34	NULL	A
1	NULL	20	34	NULL	B
1	NULL	20	34	NULL	C
1	NULL	20	34	NULL	D
2	52	27	NULL	NULL	A
2	52	27	NULL	NULL	B
2	52	27	NULL	NULL	C
2	52	27	NULL	NULL	D
3	20	NULL	22	30	A
3	20	NULL	22	30	B
3	20	NULL	22	30	C
3	20	NULL	22	30	D

Unpivoting with Standard SQL

- Extracting Elements: The second step in the solution is to produce a column that returns the value from the column that corresponds to the customer represented by the current copy.

```
SELECT EmpID, CustID,
       CASE CustID
         WHEN 'A' THEN A
         WHEN 'B' THEN B
         WHEN 'C' THEN C
         WHEN 'D' THEN D
       END AS Qty
FROM  EmpCustOrders
CROSS JOIN (VALUES('A'),
                  ('B'),
                  ('C'),
                  ('D'))
        AS Custs(CustID);
```

EmpID	CustID	Qty
1	A	NULL
1	B	20
1	C	34
1	D	NULL
2	A	52
2	B	27
2	C	NULL
2	D	NULL
3	A	20
3	B	NULL
3	C	22
3	D	30

Unpivoting with Standard SQL

- Removing Rows with NULLs: To eliminate irrelevant intersections, define a table expression and filter out NULL marks in the outer query.

```
SELECT *
FROM (SELECT EmpID, CustID,
        CASE CustID
          WHEN 'A' THEN A
          WHEN 'B' THEN B
          WHEN 'C' THEN C
          WHEN 'D' THEN D
        END AS Qty
      FROM  EmpCustOrders
      CROSS JOIN (VALUES('A'),
                        ('B'),
                        ('C'),
                        ('D'))
                AS Custs(CustID)) AS D
WHERE Qty IS NOT NULL;
```

EmpID	CustID	Qty
1	B	20
1	C	34
2	A	52
2	B	27
3	A	20
3	C	22
3	D	30

Unpivoting with the UNPIVOT Operator

- Unpivoting data involves producing *two* result columns from any number of source columns that we unpivot.
- Like the PIVOT operator, UNPIVOT was also implemented as a table operator in the context of the FROM clause.
- The general form of a query with the UNPIVOT operator:

```
SELECT <regular columns>, <summary column>, <unpivoted column>
FROM (
        SELECT <regular columns>,<col header 1>, <col header 2>
                                                  [, ... <col header N>]
        FROM   <table to unpivot>
    ) AS <alias>
    UNPIVOT (
        <summary column> FOR <unpivoted column> IN
                (<col header 1>, <col header 2>, [, ...<col header N>])
    ) AS <alias>;
```

Unpivoting with the UNPIVOT Operator

- Unpivoting data involves producing two result columns from any number of source columns that we unpivot.
- For example

```
SELECT EmpID, CustID, Qty
FROM   EmpCustOrders
       UNPIVOT( Qty FOR CustID IN(A, B, C, D) ) AS U;
```

- Note that the UNPIVOT operator implements the same logical processing phases described earlier.
- Also note that unpivoting a pivoted table cannot bring back the original table. Rather, unpivoting is just a rotation of the pivoted values into a new format.

Grouping Sets

- A grouping set is simply a set of attributes by which we group.

  Traditionally in SQL, a single aggregate query defines a single grouping set.

- When added grouping sets to an aggregate query, which allows us to combine different *grouping levels* within one statement.

- Grouping set is equivalent to combining multiple aggregate queries with UNION.

Grouping Sets

- For example, each of the following four queries defines a single grouping set.

```
SELECT EmpID, CustID, SUM(Qty) AS SumQty
FROM   Orders GROUP BY EmpID, CustID;

SELECT EmpID, SUM(Qty) AS SumQty
FROM   Orders GROUP BY EmpID;

SELECT CustID, SUM(Qty) AS SumQty
FROM   Orders GROUP BY CustID;

SELECT SUM(Qty) AS SumQty
FROM   Orders;
```

- The first query defines the grouping set (EmpID, CustID); the second (EmpID), the third (CustID), and the last query define what's known as the empty grouping set, ().

- Suppose that instead of four separate result sets, we wanted a single unified result set with the aggregated data for all four grouping sets.
- We could achieve this by using the UNION ALL set operation to unify the result sets of all four queries.

```
SELECT EmpID, CustID, SUM(Qty) AS SumQty
FROM   Orders GROUP BY EmpID, CustID;

UNION ALL

SELECT EmpID, NULL, SUM(Qty) AS SumQty
FROM   Orders GROUP BY EmpID;

UNION ALL

SELECT NULL, CustID, SUM(Qty) AS SumQty
FROM   Orders GROUP BY CustID;

UNION ALL

SELECT NULL, NULL, SUM(Qty) AS SumQty
FROM   Orders;
```

EmpID	CustID	SumQty
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30
1	NULL	54
2	NULL	79
3	NULL	72
NULL	A	72
NULL	B	47
NULL	C	56
NULL	D	30
NULL	NULL	205

The GROUPING SETS Subclause

- The GROUPING SETS subclause is a powerful enhancement to the GROUP BY clause.
- By using this subclause, we can define multiple grouping sets in the same query.
- For example:

```
SELECT EmpID, CustID, SUM(Qty) AS SumQty
FROM  Orders
GROUP BY GROUPING SETS
        (
            (EmpID, CustID),
            (EmpID),
            (CustID),
            ()
        );
```

The CUBE Subclause

- The CUBE subclause of the GROUP BY clause provides an abbreviated way to define multiple grouping sets.
- In the parentheses of the CUBE subclause, we provide a list of members separated by commas, and we get all possible grouping sets that can be defined based on the input members.
- For example, CUBE(a, b, c) is equivalent to GROUPING SETS((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ()).

Example: The CUBE Subclause

```
SELECT EmpID, CustID, SUM(Qty) AS SumQty
FROM  Orders
GROUP BY CUBE(EmpID, CustID)
```

EmpID	CustID	SumQty
2	A	52
3	A	20
NULL	A	72
1	B	20
2	B	27
NULL	B	47
1	C	34
3	C	22
NULL	C	56
3	D	30
NULL	D	30
NULL	NULL	205
1	NULL	54
2	NULL	79
3	NULL	72

The ROLLUP Subclause

- The ROLLUP subclause of the GROUP BY clause also provides an abbreviated way to define multiple grouping sets.
- ROLLUP assumes a *hierarchy* among the input members and produces all grouping sets that make sense considering the hierarchy.
- ROLLUP(a, b, c) produces only four grouping sets, assuming the hierarchy a > b > c, and is the equivalent of specifying GROUPING SETS((a, b, c), (a, b), (a), ()).

The ROLLUP Subclause

- For example, suppose that we want to return total quantities for all grouping sets that can be defined based on the time hierarchy order year > order month > order day.
- We could use the GROUPING SETS subclause and explicitly list all four possible grouping sets.

```
GROUPING SETS(
  (YEAR(OrderDate), MONTH(OrderDate), DAY(OrderDate)),
  (YEAR(OrderDate), MONTH(OrderDate)),
  (YEAR(OrderDate)),
  () )
```

- Or

```
ROLLUP(YEAR(OrderDate), MONTH(OrderDate), DAY(OrderDate))
```

```
SELECT
    YEAR(OrderDate)  AS OrderYear,
    MONTH(OrderDate) AS OrderMonth,
    DAY(OrderDate)   AS OrderDay,
    SUM(Qty) AS SumQty
FROM  Orders
GROUP BY ROLLUP(YEAR(OrderDate), MONTH(OrderDate), DAY(OrderDate));
```

OrderYear	OrderMonth	OrderDay	SumQty
2007	4	18	22
2007	4	NULL	22
2007	8	2	10
2007	8	NULL	10
2007	12	24	32
2007	12	NULL	32
2007	NULL	NULL	64
2008	1	9	40
2008	1	18	14
2008	1	NULL	54
2008	2	12	12
2008	2	NULL	12
2008	NULL	NULL	66
2009	2	12	10
2009	2	16	20
2009	2	NULL	30
2009	4	18	15
2009	4	NULL	15
2009	9	7	30
2009	9	NULL	30
2009	NULL	NULL	75
NULL	NULL	NULL	205

The GROUPING{,_ID} Functions

- When we have a single query that defines multiple grouping sets, we might need to be able to associate result rows and grouping sets.
  - If a grouping column is defined as NOT NULL then the NULL in those columns can only represent a placeholder, indicating that the column did not participate in the current grouping set.
  - If a grouping column is defined as allowing NULL marks in the table, we cannot tell for sure whether a NULL in the result set originated from the data or is a placeholder for a nonparticipating member in a grouping set.

The GROUPING{,_ID} Functions

- The GROUPING function accepts a name of a column and returns 0 if it is a member of the current grouping set and 1 otherwise.
- The GROUPING_ID function can further simplify the process of associating result rows and grouping sets.
  - For example, GROUPING_ID(a, b, c, d) - the function returns an integer bitmap in which each bit represents a different input element the rightmost element represented by the rightmost bit.
    - The grouping set (a, b, c, d) is represented by the integer 0 (0×8 + 0×4 + 0×2 + 0×1).
    - The grouping set (a, c) is represented by the integer 5 (0×8 + 1×4 + 0×2 + 1×1), and so on.

```
SELECT
    GROUPING_ID(EmpID, CustID) AS GroupingSet,
    GROUPING(EmpID)  AS GrpEmp,
    GROUPING(CustID) AS GrpCust,
    EmpID, CustID, SUM(Qty) AS SumQty
FROM  Orders
GROUP BY CUBE(EmpID, CustID);
```

GroupingSet	GrpEmp	GrpCust	EmpID	CustID	SumQty
0	0	0	2	A	52
0	0	0	3	A	20
2	1	0	NULL	A	72
0	0	0	1	B	20
0	0	0	2	B	27
2	1	0	NULL	B	47
0	0	0	1	C	34
0	0	0	3	C	22
2	1	0	NULL	C	56
0	0	0	3	D	30
2	1	0	NULL	D	30
3	1	1	NULL	NULL	205
1	0	1	1	NULL	54
1	0	1	2	NULL	79
1	0	1	3	NULL	72

- The integer 0 (binary 00) represents the grouping set (EmpID, CustID); the integer 1 (binary 01) represents (EmpID); the integer 2 (binary 10) represents (CustID); and the integer 3 (binary (11) represents ().

Advanced Query

Advanced CTE Queries

- Common table expressions (CTEs) are defined by using a WITH statement and have the following general form.

```
WITH <CTEName> [(<Col1>, <Col2>)]
AS (SELECT <Col3>, <Col4> FROM <table>)
SELECT <Col1>, <Col2>
FROM <CTEName>;
```

- CTEs allow us to isolate part of the query logic or do things we could not ordinarily do, such as use an aggregate expression in an update.
- We can do several things with CTEs that are not possible with derived tables, such as write a recursive query.

Defining Multiple CTEs

- Defining a CTE and then use it gives it several important advantages over derived tables.
- One of those advantages is that if we need to refer to one CTE from another, we don’t end up nesting them as we do with derived tables.
- Instead, we simply define multiple CTEs separated by *commas* under the same WITH statement.

Defining Multiple CTEs

- Multiple CTEs can be simply defined by separated by commas under the same WITH statement.

```
WITH <CTEName1> AS (SELECT <col1> FROM <table1>),
     <CTEName2> AS (SELECT <col2> FROM <table2> [<JOIN> <CTEName1>]),
     <CTEName3> AS (SELECT <col3> FROM <table3> [<JOIN> <CTEName1>]
                                                [<JOIN> <CTEName2>])
     …
SELECT <col1>, <col2>, <col3>, …
FROM <CTEName1> <JOIN> <CTEName2> ON <join condition1>
     <JOIN> <CTEName3> ON <join condition2> …
```

- Because we define a CTE before we use it, we don't end up nesting CTEs. Each CTE appears separately in the code in a modular manner.

```
CREATE TABLE Employee (
    EmployeeID INT NOT NULL,
    ContactID  INT NOT NULL,
    ManagerID  INT NULL,
    Title      NVARCHAR(50) NOT NULL);

CREATE TABLE Contact (
    ContactID  INT NOT NULL,
    FirstName  NVARCHAR(50) NOT NULL,
    MiddleName NVARCHAR(50) NULL,
    LastName   NVARCHAR(50) NOT NULL);

CREATE TABLE JobHistory (
    EmployeeID       INT NOT NULL,
    EffDate          DATE NOT NULL,
    EffSeq           INT NOT NULL,
    EmploymentStatus CHAR(1) NOT NULL,
    JobTitle         VARCHAR(50) NOT NULL,
    Salary           MONEY NOT NULL,
    ActionDesc       VARCHAR(20)
    CONSTRAINT PK_JobHistory PRIMARY KEY CLUSTERED (
        EmployeeID, EffDate, EffSeq) );
```

```
INSERT INTO Contact (ContactID, FirstName, MiddleName, LastName) VALUES
    (1030,'Kevin','F', 'Brown'),
    (1009,'Thierry','B', 'DHers'),
    (1028,'David','M','Bradley'),
    (1070,'JoLynn','M','Dobney'),
    (1071,'Ruth','Ann','Ellerbrock'),
    (1005,'Gail','A','Erickson'),
    (1076,'Barry','K','Johnson'),
    (1006,'Jossef','H','Goldberg'),
    (1001,'Terri','Lee','Duffy'),
    (1072,'Sidney','M','Higa'),
    (1067,'Taylor','R','Maxwell'),
    (1073,'Jeffrey','L','Ford'),
    (1068,'Jo','A','Brown'),
    (1074,'Doris','M','Hartwig'),
    (1069,'John','T','Campbell'),
    (1075,'Diane','R','Glimp'),
    (1129,'Steven','T','Selikoff'),
    (1231,'Peter','J','Krebs'),
    (1172,'Stuart','V','Munson'),
    (1173,'Greg','F','Alderson'),
    (1113,'David','N','Johnson'),
    (1054,'Zheng','W','Mu'),
    (1007,'Ovidiu','V','Cracium'),
    (1052,'James','R','Hamilton'),
    (1053,'Andrew','R','Hill'),
    (1056,'Jack','S','Richins'),
    (1058,'Michael','Sean','Ray'),
    (1064,'Lori','A','Kane'),
    (1287,'Ken','J','Sanchez');
```

```
INSERT INTO Employee(EmployeeID, ContactID, ManagerID, Title) VALUES
    (1, 1209, 16,'Production Technician - WC60'),
    (2, 1030, 6,'Marketing Assistant'),
    (3, 1002, 12,'Engineering Manager'),
    (4, 1290, 3,'Senior Tool Designer'),
    (5, 1009, 263,'Tool Designer'),
    (6, 1028, 109,'Marketing Manager'),
    (7, 1070, 21,'Production Supervisor - WC60'),
    (8, 1071, 185,'Production Technician - WC10'),
    (9, 1005, 3,'Design Engineer'),
    (10, 1076, 185,'Production Technician - WC10'),
    (11, 1006, 3,'Design Engineer'),
    (12, 1001, 109,'Vice President of Engineering'),
    (13, 1072, 185,'Production Technician - WC10'),
    (14, 1067, 21,'Production Supervisor - WC50'),
    (15, 1073, 185,'Production Technician - WC10'),
    (16, 1068, 21,'Production Supervisor - WC60'),
    (17, 1074, 185,'Production Technician - WC10'),
    (18, 1069, 21,'Production Supervisor - WC60'),
    (19, 1075, 185,'Production Technician - WC10'),
    (20, 1129, 173,'Production Technician - WC30'),
    (21, 1231, 148,'Production Control Manager'),
    (22, 1172, 197,'Production Technician - WC45'),
    (23, 1173, 197,'Production Technician - WC45'),
    (24, 1113, 184,'Production Technician - WC30'),
    (25, 1054, 21,'Production Supervisor - WC10'),
    (109, 1287, NULL, 'Chief Executive Officer'),
    (148, 1052, 109, 'Vice President of Production'),
    (173, 1058, 21, 'Production Supervisor - WC30'),
    (184, 1056, 21, 'Production Supervisor - WC30'),
    (185, 1053, 21, 'Production Supervisor - WC10'),
    (197, 1064, 21, 'Production Supervisor - WC45'),
    (263, 1007, 3, 'Senior Tool Designer');
```

```
INSERT INTO JobHistory(EmployeeID, EffDate, EffSeq,
        EmploymentStatus, JobTitle, Salary, ActionDesc) VALUES
(1000,'07-31-2008',1,'A','Intern',2000,'New Hire'),
(1000,'05-31-2009',1,'A','Production Technician',2000,'Title Change'),
(1000,'05-31-2009',2,'A','Production Technician',2500,'Salary Change'),
(1000,'11-01-2009',1,'A','Production Technician',3000,'Salary Change'),
(1200,'01-10-2009',1,'A','Design Engineer',5000,'New Hire'),
(1200,'05-01-2009',1,'T','Design Engineer',5000,'Termination'),
(1100,'08-01-2008',1,'A','Accounts Payable Specialist I',2500,'New Hire'),
(1100,'05-01-2009',1,'A','Accounts Payable Specialist II',2500,'Title Change'),
(1100,'05-01-2009',2,'A','Accounts Payable Specialist II',3000,'Salary Change');
```

- The system contains history information with an *effective date* and an *effective sequence*.
- The system adds one row to these tables for each change to the employee's data.
- For a particular *effective date*, the system can add *more than one* row along with an incrementing *effective sequence*.
- To display information valid on a particular date, we first have to figure out the *latest effective date* before the date in mind and then figure out the *effective sequence* for that date.

- Multiple CTE Example: Finding employee's name and their manager's name using multiple CTEs

```
WITH
Emp AS(
   SELECT E.EmployeeID, E.ManagerID, E.Title AS EmpTitle,
          C.FirstName + ' ' + C.LastName AS EmpName
   FROM Employee AS E
   JOIN Contact  AS E ON E.ContactID = C.ContactID
),
Mgr AS(
   SELECT E.EmployeeID AS ManagerID, E.Title AS MgrTitle,
          C.FirstName + ' ' + C.LastName AS MgrName
   FROM Employee AS E
   JOIN Contact  AS C ON E.ContactID = C.ContactID
)
SELECT EmployeeID, Emp.ManagerID, EmpName,
       EmpTitle, MgrName, MgrTitle
FROM Emp
JOIN Mgr ON Emp.ManagerID = Mgr.ManagerID
ORDER BY EmployeeID;
```

- Multiple CTE Example: Finding employee's name and their manager's name using multiple CTEs

	EmployeeID	ManagerID	EmpName	EmpTitle	MgrName	MgrTitle
1	2	6	Kevin Brown	Marketing Assistant	David Bradley	Marketing Manager
2	5	263	Thierry DHers	Tool Designer	Ovidiu Cracium	Senior Tool Designer
3	6	109	David Bradley	Marketing Manager	Ken Sanchez	Chief Executive Officer
4	7	21	JoLynn Dobney	Production Supervisor - WC60	Peter Krebs	Production Control Manager
5	8	185	Ruth Ellerbrock	Production Technician - WC10	Andrew Hill	Production Supervisor - WC10
6	10	185	Barry Johnson	Production Technician - WC10	Andrew Hill	Production Supervisor - WC10
7	12	109	Terri Duffy	Vice President of Engineering	Ken Sanchez	Chief Executive Officer
8	13	185	Sidney Higa	Production Technician - WC10	Andrew Hill	Production Supervisor - WC10
9	14	21	Taylor Maxw...	Production Supervisor - WC50	Peter Krebs	Production Control Manager
10	15	185	Jeffrey Ford	Production Technician - WC10	Andrew Hill	Production Supervisor - WC10
11	16	21	Jo Brown	Production Supervisor - WC60	Peter Krebs	Production Control Manager
12	17	185	Doris Hartwig	Production Technician - WC10	Andrew Hill	Production Supervisor - WC10
13	18	21	John Campbell	Production Supervisor - WC60	Peter Krebs	Production Control Manager
14	19	185	Diane Glimp	Production Technician - WC10	Andrew Hill	Production Supervisor - WC10
15	20	173	Steven Selikoff	Production Technician - WC30	Michael Ray	Production Supervisor - WC30
16	21	148	Peter Krebs	Production Control Manager	James Hamilt...	Vice President of Production
17	22	197	Stuart Munson	Production Technician - WC45	Lori Kane	Production Supervisor - WC45

Joining a CTE to Another CTE

- Finding employee who effective date and JobTitle

```
DECLARE @Date DATE = '20090502';

WITH EffectiveDate AS (
    SELECT   MAX(EffDate) AS MaxDate, EmployeeID
    FROM     JobHistory
    WHERE    EffDate <= @Date
    GROUP BY EmployeeID
),
EffectiveSeq AS (
    SELECT   MAX(EffSeq) AS MaxSeq, J.EmployeeID, MaxDate
    FROM     JobHistory     AS J
    JOIN     EffectiveDate  AS D
      ON     J.EffDate = D.MaxDate AND J.EmployeeID = D.EmployeeID
    GROUP BY J.EmployeeID, MaxDate )
SELECT J.EmployeeID, EmploymentStatus, JobTitle, Salary
FROM   JobHistory    AS J
JOIN   EffectiveSeq  AS E
  ON   J.EmployeeID = E.EmployeeID
 AND   J.EffDate = E.MaxDate AND J.EffSeq = E.MaxSeq;
```

- Example: Finding employee who effective date and JobTitle

EmployeeID	EmploymentStatus	JobTitle	Salary
1000	A	Intern	2000.00
1100	A	Accounts Payable Specialist II	3000.00
1200	T	Design Engineer	5000.00

- The first CTE, *EffectiveDate*, just determines the maximum EffDate from the JobHistory table for each employee.
- The second CTE, *EffectiveSeq*, joins the JobHistory table to the EffectiveDate CTE to find the maximum EffSeq for each employee.
- Finally, the outer query joins the JobHistory table on the EffectiveSeq CTE to display the valid data for each employee.

Recursive CTEs

- A recursive CTE is defined by at least two queries (more are possible) - at least one query known as the anchor member and at least one query known as the recursive member.
- The general form of a basic recursive CTE looks like the following.

```
WITH <CTE_Name>[(<target_column_list>)]
AS
(
    <anchor_member>
    UNION ALL
    <recursive_member>
)
<outer_query_against_CTE>;
```

Recursive CTEs

- The anchor member is a query that returns a valid relational result table - like a query that is used to define a nonrecursive table expression. The anchor member query is invoked *only once*.
- The recursive member is a query that has a reference to the *CTE name*. The reference to the CTE name represents what is logically the *previous result set* in a sequence of executions.
- The recursive member is invoked repeatedly until it returns an *empty set* or *exceeds some limit*.

- Example: a recursive CTE to return information about an employee (EmployeeID = 185)

```
WITH EmpsCTE AS
(
    SELECT E.EmployeeID, E.ManagerID, T.FirstName, T.LastName
    FROM   Employee AS E
    JOIN   Contact  AS T ON E.ContactID = T.ContactID
    WHERE  EmployeeID = 185

    UNION ALL

    SELECT E.EmployeeID, E.ManagerID, T.FirstName, T.LastName
    FROM   EmpsCTE  AS P
    JOIN   Employee AS E ON P.EmployeeID = E.ManagerID
    JOIN   Contact  AS T ON E.ContactID  = T.ContactID
)
SELECT EmployeeID, ManagerID, FirstName, LastName
FROM   EmpsCTE;
```

Boss

P.ManagrID = E.EmployeeID

Subordinate

EmployeeID	ManagerID	FirstName	LastName
185	21	Andrew	Hill
8	185	Ruth	Ellerbrock
10	185	Barry	Johnson
13	185	Sidney	Higa
15	185	Jeffrey	Ford
17	185	Doris	Hartwig
19	185	Diane	Glimp

- Example: a recursive CTE to return information about an employee (EmployeeID = 21)

```
WITH EmpsCTE AS
(
    SELECT E.EmployeeID, E.ManagerID, T.FirstName, T.LastName
    FROM   Employee AS E
    JOIN   Contact AS T ON E.ContactID = T.ContactID
```

EmployeeID	ManagerID	FirstName	LastName
21	148	Peter	Krebs
7	21	JoLynn	Dobney
14	21	Taylor	Maxwell
16	21	Jo	Brown
18	21	John	Campbell
25	21	Zheng	Mu
185	21	Andrew	Hill
184	21	Jack	Richins
173	21	Michael	Ray
197	21	Lori	Kane
22	197	Stuart	Munson
23	197	Greg	Alderson
20	173	Steven	Selikoff
24	184	David	Johnson
8	185	Ruth	Ellerbrock
10	185	Barry	Johnson
13	185	Sidney	Higa
15	185	Jeffrey	Ford
17	185	Doris	Hartwig
19	185	Diane	Glimp

Example: Generating Numbers

- We can use CTEs to generate numbers
- For example, generating numbers from 0 to 99

```
WITH Numbers AS
(
    SELECT 0 AS N
    UNION ALL
    SELECT N + 1
    FROM   Numbers
    WHERE  N < 99
)
SELECT N
FROM   Numbers
ORDER BY N
```

- For example,

  generating numbers

  from 0 to 999

Data Manipulation with CTEs

- We can use CTEs when modifying data.
- Example, inserts rows from an AdventureWorks query within the CTE supplying the CustomerID and names.

```
CREATE TABLE CTEExample(CustomerID INT, FirstName NVARCHAR(50),
                        LastName NVARCHAR(50), Sales Money);

WITH Cust AS(
    SELECT CustomerID, FirstName, LastName
    FROM   Sales.Customer AS C
    JOIN   Person.Person  AS P
      ON   C.CustomerID = P.BusinessEntityID
)
INSERT INTO CTEExample(CustomerID, FirstName, LastName)
SELECT CustomerID, FirstName, LastName
FROM   Cust;
```

- Usually when inserting data from a query, the word INSERT is first. When using a CTE, the statement begins with the CTE and then it is used in the outer query as any other table.

- Example, update with CTE

```
WITH Totals AS
(
    SELECT   CustomerID, SUM(TotalDue) AS CustTotal
    FROM     Sales.SalesOrderHeader
    GROUP BY CustomerID
)
UPDATE C SET Sales = CustTotal
FROM CTEExample AS C
JOIN Totals
  ON C.CustomerID = Totals.CustomerID;
```

Updates the table with an aggregate expression, SUM(TotalDue). It is important to note that it is not possible to update directly with an aggregate, so we must come up with some way to isolate the aggregate query.

- Example, delete with CTE

```
WITH Cust AS
(
    SELECT CustomerID, Sales
    FROM   CTEExample
)
DELETE Cust
WHERE  Sales < 10000;
```

Not only is it using a CTE to delete some rows, it is actually deleting rows from the CTEExample itself.