```
/*
The LAG() function allows access to a value stored in a different row above the
current row.
The row above may be adjacent or some number of rows above, as sorted by a specified
column or set of columns.

Syntax:
LAG(expression,[offset],default_value]) OVER(ORDER BY columns)

LAG() takes three arguments: the name of the column or an expression from which the
value is obtained,
the number of rows to skip (offset) above, and the default value to be returned if the
stored value obtained
from the row above is empty. Only the first argument is required.
The third argument (default value) is allowed only if you specify the second argument,
the offset.

The LEAD() is similar to LAG(). Whereas LAG() accesses a value stored in a row above,
LEAD() accesses a value stored in a row below.

Syntax:
LEAD(expression [,offset[,default_value]]) OVER(ORDER BY columns)

Just like LAG(), the LEAD() function takes three arguments: the name of a column or an
expression,
the offset to be skipped below, and the default value to be returned if the stored
value obtained
from the row below is empty. Only the first argument is required.
The third argument, the default value, can be specified only if you specify the second
argument, the offset.
*/

CREATE TABLE dbo.SalesData
(
ID                  INT,
Seller_Name VARCHAR(20),
Sale_Value    MONEY
)
INSERT INTO dbo.SalesData VALUES
(3,'Stef',7000),
(1,'Alic',12000),
(2,'Mili',25000)

SELECT * FROM dbo.SalesData

-- LAG
SELECT seller_name, sale_value,
  LAG(sale_value) OVER(ORDER BY sale_value) as previous_sale_value
FROM dbo.SalesData;

SELECT seller_name, sale_value,
  LAG(sale_value,1,0) OVER(ORDER BY sale_value) as previous_sale_value
FROM dbo.SalesData;

-- LEAD
SELECT seller_name, sale_value,
  LEAD(sale_value) OVER(ORDER BY sale_value) as next_sale_value
FROM dbo.SalesData;

SELECT seller_name, sale_value,
  LEAD(sale_value,1,0) OVER(ORDER BY sale_value) as next_sale_value
FROM dbo.SalesData;
```

```sql
/*
Using LAG() and LEAD() to Compare Values
An important use for LAG() and LEAD() in reports is comparing the values in the
current row with the values in the same column but in a row above or below.

Consider the following table, annual_sale, shown below:
*/
CREATE TABLE dbo.SaleByYear
(
	SaleYear	INT,
	TotalSales	MONEY
)

INSERT INTO dbo.SaleByYear VALUES
(2015,23000),
(2016,25000),
(2017,34000),
(2018,32000),
(2019,33000),
(2020,37000),
(2021,37000),
(2022,42000),
(2023,45000)

SELECT * FROM dbo.SaleByYear

SELECT
   SaleYear,
   TotalSales AS current_total_sale,
   LAG(TotalSales) OVER(ORDER BY SaleYear) AS previous_total_sale,
   TotalSales - LAG(TotalSales) OVER(ORDER BY SaleYear) AS difference
FROM dbo.SaleByYear;

-- LEAD AND LAG WITH OFFSET
CREATE TABLE dbo.EmpBonus
(
	EmpId INT,
	BonusYear	INT,
	BonusQuar	SMALLINT,
	BonusAmt	DECIMAL
)

INSERT INTO dbo.EmpBonus VALUES
(1,2017,1,100),
(1,2017,2,250),
(1,2017,3,60),
(1,2017,4,20),
(1,2018,1,80),
(1,2018,2,80),
(1,2018,3,0),
(1,2018,4,0),
(1,2019,1,0),
(1,2019,2,100),
(1,2019,3,0),
(1,2019,4,150)

--The query below selects the bonus for the employee with ID=1 for each quarter of
each year.
--It then identifies the bonuses for the corresponding quarter in the year before and
the year after.
```

```
SELECT * FROM dbo.EmpBonus

SELECT
       BonusYear,
       BonusQuar,
       LAG(BonusAmt,4) OVER(ORDER BY BonusYear,BonusQuar) AS previous_bonus,
       BonusAmt AS current_bonus,
       LEAD(BonusAmt,4) OVER(ORDER BY BonusYear,BonusQuar) AS next_bonus
FROM dbo.EmpBonus
WHERE EmpId=1;
```

**UPDATABLE CTE:**

**Is it possible to UPDATE a CTE?**
**Yes & No**, depending on the number of base tables, the CTE is created upon, and the number of base tables affected by the UPDATE statement. If this is not clear at the moment, don't worry. We will try to understand this with an example.

**Let's create a simple common table expression**, based on tblEmployee table. **Employees_Name_Gender** CTE is getting all the required columns from one base table tblEmployee.

```
With Employees_Name_Gender
as
(
 Select Id, Name, Gender from tblEmployee
)
Select * from Employees_Name_Gender
```

**Let's now, UPDATE JOHN's gender from Male to Female**, using the **Employees_Name_Gender CTE**

```
With Employees_Name_Gender
as
(
 Select Id, Name, Gender from tblEmployee
)
Update Employees_Name_Gender Set Gender = 'Female' where Id = 1
```

**Now, query the tblEmployee table**. JOHN's gender is actually UPDATED. So, if a CTE is created on one base table, then it is possible to UPDATE the CTE, which in turn will update the underlying base table. In this case, UPDATING **Employees_Name_Gender** CTE, updates **tblEmployee** table.

**Now, let's create a CTE, on both the tables - tblEmployee and tblDepartment.** The CTE should return, Employee Id, Name, Gender and Department. In short the output should be as shown below.

**CTE, that returns Employees by Department**

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
```

```
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Select * from EmployeesByDepartment
```

**Let's update this CTE.** Let's change JOHN's Gender from **Female to Male**. Here, the CTE is based on 2 tables, but the UPDATE statement affects only one base table **tblEmployee**. So the UPDATE succeeds. So, if a CTE is based on more than one table, and if the UPDATE affects only one base table, then the UPDATE is allowed.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set Gender = 'Male' where Id = 1
```

Now, let's try to **UPDATE the CTE**, in such a way, that the update affects both the tables - **tblEmployee and tblDepartment**. This UPDATE statement changes **Gender** from **tblEmployee** table and **DeptName** from **tblDepartment** table. When you execute this UPDATE, you get an error stating - 'View or function EmployeesByDepartment is not updatable because the modification affects multiple base tables'. So, if a CTE is based on multiple tables, and if the UPDATE statement affects more than 1 base table, then the UPDATE is not allowed.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set
Gender = 'Female', DeptName = 'IT'
where Id = 1
```

**Finally, let's try to UPDATE just the DeptName**. Let's change JOHN's DeptName from HR to IT. Before, you execute the UPDATE statement, notice that BEN is also currently in HR department.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set
DeptName = 'IT' where Id = 1
```

After you execute the UPDATE. Select data from the CTE, and you will see that BEN's DeptName is also changed to IT.

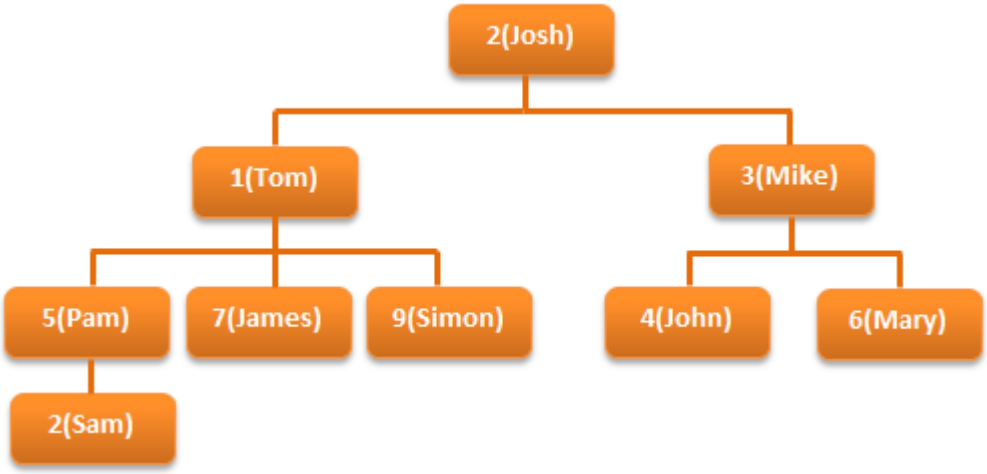
**This is because**, when we updated the **CTE**, the UPDATE has actually changed the **DeptName** from **HR** to **IT**, in **tblDepartment** table, instead of changing the **DepartmentId** column (from 3 to 1) in **tblEmployee** table. So, if a CTE is based on multiple tables, and if the UPDATE statement affects only one base table, the update succeeds. But the update may not work as you expect.

**So in short if,**
**1.** A CTE is based on a single base table, then the UPDATE suceeds and works as expected.
**2.** A CTE is based on more than one base table, and if the UPDATE affects multiple base tables, the update is not allowed and the statement terminates with an error.

**3.** A CTE is based on more than one base table, and if the UPDATE affects only one base table, the UPDATE succeeds(but not as expected always)

**RECURSIVE CTE:**

**A CTE that references itself is called as recursive CTE**. Recursive CTE's can be of great help when displaying hierarchical data. Example, displaying employees in an organization hierarchy. A simple organization chart is shown below.




**Let's create EmpMngr table, which holds the data, that's in the organization chart.**

```
Create Table EmpMngr
(
  EmployeeId int Primary key,
  Name nvarchar(20),
  ManagerId int
)
Insert into EmpMngr values (1, 'Tom', 2)
Insert into EmpMngr values (2, 'Josh', null)
Insert into EmpMngr values (3, 'Mike', 2)
Insert into EmpMngr values (4, 'John', 3)
Insert into EmpMngr values (5, 'Pam', 1)
Insert into EmpMngr values (6, 'Mary', 3)
Insert into EmpMngr values (7, 'James', 1)
```

```
Insert into EmpMngr values (8, 'Sam', 5)
Insert into EmpMngr values (9, 'Simon', 1)
```

**Since, a MANAGER is also an EMPLOYEE**, both manager and employee details are stored in tblEmployee table. Data from tblEmployee is shown below.

| EmployeeId | Name | ManagerId |
|---|---|---|
| 1 | Tom | 2 |
| 2 | Josh | NULL |
| 3 | Mike | 2 |
| 4 | John | 3 |
| 5 | Pam | 1 |
| 6 | Mary | 3 |
| 7 | James | 1 |
| 8 | Sam | 5 |
| 9 | Simon | 1 |

**Let's say, we want to display, EmployeeName along with their ManagerName**. The ouptut should be as shown below.

| Employee Name | Manager Name |
|---|---|
| Tom | Josh |
| Josh | Super Boss |
| Mike | Josh |
| John | Mike |
| Pam | Tom |
| Mary | Mike |
| James | Tom |
| Sam | Pam |
| Simon | Tom |

**To achieve this, we can simply join tblEmployee with itself.** Joining a table with itself is called as self join. In the output, notice that since **JOSH** does not have a Manager, we are displaying **'Super Boss'**, instead of **NULL**. We used **IsNull**(), function to replace NULL with 'Super Boss'.

**SELF JOIN QUERY:**

```
Select Employee.Name as [Employee Name],
IsNull(Manager.Name, 'Super Boss') as [Manager Name]
from EmpMngr Employee
left join EmpMngr Manager
on Employee.ManagerId = Manager.EmployeeId
```

- **Along with Employee and their Manager name**, we also want to display their level in the organization. The output should be as shown below

| Employee | Manager | Level |
|---|---|---|
| Josh | Super Boss | 1 |
| Tom | Josh | 2 |
| Mike | Josh | 2 |
| John | Mike | 3 |
| Mary | Mike | 3 |
| Pam | Tom | 3 |
| James | Tom | 3 |
| Simon | Tom | 3 |
| Sam | Pam | 4 |

**We can easily achieve this using a self referencing CTE.**

```
With
  EmployeesCTE (EmployeeId, Name, ManagerId, [Level])
  as
  (
    Select EmployeeId, Name, ManagerId, 1
    from tblEmployee
    where ManagerId is null

    union all

    Select tblEmployee.EmployeeId, tblEmployee.Name,
    tblEmployee.ManagerId, EmployeesCTE.[Level] + 1
    from tblEmployee
    join EmployeesCTE
    on tblEmployee.ManagerID = EmployeesCTE.EmployeeId
  )
Select EmpCTE.Name as Employee, Isnull(MgrCTE.Name, 'Super Boss') as Manager,
EmpCTE.[Level]
from EmployeesCTE EmpCTE
left join EmployeesCTE MgrCTE
on EmpCTE.ManagerId = MgrCTE.EmployeeId
```

The **EmployeesCTE** contains 2 queries with **UNION ALL** operator. The first query selects the EmployeeId, Name, ManagerId, and 1 as the level from **tblEmployee** where ManagerId is NULL. So, here we are giving a LEVEL = 1 for **super boss** (Whose Manager Id is NULL). In the second query, we are joining **tblEmployee** with **EmployeesCTE** itself, which allows us to loop thru the hierarchy. Finally to get the reuired output, we are joining **EmployeesCTE** with itself.