APPLY JOIN

For INNER and OUTER joins of two tables, all rows in the two tables included in the Join operation are evaluated for whether the Join condition is true. If the condition is true, a row is created and included in the result table. We will evaluate for all combinations of all rows that are in the two tables when the operation starts.

With APPLY JOIN it is different. For each row in the left table - called the outer table - a Table-Valued Function or a Sub-Select is performed. The rows returned from the Table Valued Function or the Sub-Select are the right table in the join operation and is Cross-Joined with the left row currently being evaluated. The rows used from the right table are not semantically known at the start of execution, but first known when a Table-Valued Function or a Sub-Select is executed for each row in the left table.

The Function or Sub-Select is performed semantically as many times as there are rows in the left/outer table. When performing the Function or Sub-Select, reference is normally but not necessarily made to one or more columns in the left/outer table. This is the Join condition.

A subset of the possible rows is returned from the right table. In the sketch, it is the TOP 2 rows selected based on the value of the numeric column Number in DESC order for rows with the same ID value as the ID value in the row from the left table which is currently being processed. With CROSS APPLY, the Cartesian product is formed between this row and the result table returned from a Function or a Sub-Select. We first look at the row from the left table with the value ID = 1. There are 5 rows in the right table that match this ID. But we return only the two of the five rows from the right table. The other three rows are ignored and will not be included in the result. Then a CROSS JOIN is performed with ID = 1 row from the left table and the 2 rows are returned as an intermediate result from the right table. Then we continue with the row ID = 2 from the left table. Since the value 2/red in the left table does not match any row in the right table, 0/zero rows are returned in the intermediate result. The CROSS JOIN operation between one row and zero rows will result in zero rows, and ID = 2/red row from the left table is not included in the result when CROSS APPLY is used. Then it continues with the row ID = 3 and then with the row ID = 4. Therefore, the semantic understanding is that you loop through all the rows from the left table.

When using OUTER APPLY, ID =2/red row is included in the result according to the same principle as with OUTER JOIN. ID = 2/red row is included in the result with NULL for the columns from the right table.

Tables

ID	Text
1	aaaaa
2	bbbbbbbbb
3	cccccccc
4	ddddd

ID	Number
1	20
1	17
1	16
1	12
1	10
3	99
3	93
3	87
4	65

CROSS APPLY

1	aaaaa	1	20
1	aaaaa	1	17
3	cccccccc	3	99
3	cccccccc	3	93
4	ddddd	4	65

OUTER APPLY

1	aaaaa	1	20
1	aaaaa	1	17
2	bbbbbbbbb	NULL	NULL
3	cccccccc	3	99
3	cccccccc	3	93
4	ddddd	4	65

If we execute an INNER JOIN or a LEFT OUTER JOIN with the same data the result is shown in the following sketch. A quite different result than using APPLY JOIN above.

Tables

ID	Text
1	aaaaa
2	bbbbbbbbb
3	cccccccc
4	ddddd

ID	Number
1	20
1	17
1	16
1	12
1	10
3	99
3	93
3	87
4	65

INNER JOIN

1	aaaaa	1	16
1	aaaaa	1	12
1	aaaaa	1	10
1	aaaaa	1	20
1	aaaaa	1	17
3	cccccccc	3	87
3	cccccccc	3	99
3	cccccccc	3	93
4	ddddd	4	65

LEFT OUTER JOIN

1	aaaaa	1	16
1	aaaaa	1	12
1	aaaaa	1	10
1	aaaaa	1	20
1	aaaaa	1	17
2	bbbbbbbbb	NULL	NULL
3	cccccccc	3	87
3	cccccccc	3	99
3	cccccccc	3	93
4	ddddd	4	65

The example shows, that a CROSS APPLY returns a table with 5 rows and OUTER APPLY the result contains 6 rows. If we execute an INNER JOIN on the two tables, the result table contains 9 rows and the LEFT OUTER JOIN

contains 10 rows. With the INNER and LEFT JOIN operations, the rows from the two tables are known before execution of the operation. Unlike APPLY JOIN, where the relevant/interesting rows from the right table are semantically continuously found as part of the operation.

For me it is incorrect use of APPLY JOIN if we for each row in the left table select all rows from the right table where the "join condition" is true. There are several examples on the web where OUTER APPLY is used as an alternative to LEFT OUTER JOIN. To me it shows that the semantics of the operation are not known. We also see examples of using GROUP BY without any aggregation functions as a substitute for using DISTINCT. Both examples show that the knowledge of the semantics is not present, and perhaps more problematically, that the statement is less understandable. Sometimes the comments on the statement show a lack of semantic understanding. If GROUP BY is used instead of DISTINCT and OUTER APPLY is used instead of OUTER JOIN, the result will be correct, but we have the different operations to use for different solutions that we need for our solutions.

And as soon as it is pointed out, we can see examples of better performance using the "wrong" method. But this is not the general picture of how it works, and such examples can always be found. Maybe you've seen an example/article where adding a new index will slow down the execution of a statement - but is that the normal behavior when defining new indexes? We can see examples where FULL JOIN is faster than INNER JOIN, but that should not be the reason to use FULL JOIN when we want to return the INNER JOIN rows – it can cause serious problems with the result set later.

To show the APPLY JOIN, we create two tables. The table dbo.SalesPerson contains all the salespeople from the company. In the second table dbo.Sale, we have an overview of how much a salesperson has sold in total to a given customer. This table is not necessarily a persistent table, but may be the result of a GROUP BY of millions of rows. This grouping can be specified in a CTE, View, or Sub-Select. We use a persistent table for simplicity. The definition and content of the tables look like this.

```
CREATE TABLE dbo.SalesPerson
(
        SalesPersonID       INT             NOT NULL
                            CONSTRAIN PK_SalesPerson PRIMARY KEY,
        Name                VARCHAR (30)    NOT NULL
);

CREATE TABLE dbo.Sale
(
        SaleID              INT             NOT NULL        IDENTITY
                            CONSTRAINT PK_Sale PRIMARY KEY,
        CustomerID          INT             NOT NULL,
        Amount              INT             NOT NULL,
        SalesPersonID       INT             NOT NULL
                            CONSTRAINT FK_Sale_SalesPerson FOREIGN KEY REFERENCES dbo.SalesPerson (SalesPersonID)
);
GO
INSERT INTO dbo.SalesPerson (SalesPersonID, Name) VALUES
        (1, 'Ole'),         (2, 'Ida'),         (3, 'Ane'),         (4, 'Per');

INSERT INTO dbo.Sale (CustomerID, Amount, SalesPersonID) VALUES
        (23, 100, 1),       (63, 200, 1),       (13, 400, 1),       (56, 700, 1),       (78, 200, 1),
        (18, 300, 2),       (53, 500, 2),       (17, 500, 2),                           (49, 200, 4);
```

For each salesperson, we want a list of her/his 3 "best" customers. In this task, it is the customers with the largest value in the Amount column. SalesPersonID = 1 have traded with 5 customers, which we can see in the table dbo.Sale. An INNER JOIN cannot be performed between the two tables as this operation will return all 5 customers for SalesPersonID = 1 and not just the top 3. Of course we must define what is “best”. Is it the biggest amount, the newest, if we have a DATE column, …, fewest, …

With CROSS APPLY we use a Sub-Select. The Sub-Select returns a maximum of three rows, the rows with the largest value in the Amount column for a given Salesperson. Note that no join condition is specified, as the join condition is indirectly specified in the Sub-Select, but may also be omitted.

```
SELECT      *
        FROM dbo.SalesPerson CROSS APPLY
                                (SELECT     TOP (3)
                                            CustomerID,
                                            Amount
                                            FROM dbo.Sale
                                            WHERE Sale.SalesPersonID = SalesPerson.SalesPersoniD
                                            ORDER BY Amount DESC) AS TopSale;
```

CROSS APPLY forms the Cartesian Product between rows from the outer table and the result returned from the Sub-Select. Since the Sub-Select returns 0 rows for the salesperson with the value 3, this person is not included in the result. For SalesPesonID = 4 there is only 1 customer.

SalesPersonID	Name	CustomerID	Amount
1	Ole	56	700
1	Ole	13	400
1	Ole	63	200
2	Ida	17	500
2	Ida	53	500
2	Ida	18	300
4	Per	49	200

Instead of CROSS APPLY, the OUTER APPLY can be specified. As with OUTER JOIN, rows are included in the result even if there are no related rows in the right table. The columns from this table are NULL in the result.

```
SELECT    *
          FROM dbo.SalesPerson OUTER APPLY
                                        (SELECT    TOP (3)
                                                   CustomerID,
                                                   Amount
                                                   FROM dbo.Sale
                                                   WHERE Sale.SalesPersonID = SalesPerson.SalesPersonID
                                                   ORDER BY Amount DESC) AS TopSale;
```

SalesPersonID with the value 3 is included in the result table with NULL for the columns from the Sub-Select. In this case the columns CustomerID and Amount.

SalesPersonID	Name	CustomerID	Amount
1	Ole	56	700
1	Ole	13	400
1	Ole	63	200
2	Ida	17	500
2	Ida	53	500
2	Ida	18	300
3	Ane	NULL	NULL
4	Per	49	200

Instead of a Sub-Select, a Table-Valued Function can be used. A function has the advantage that more complex code can be executed. Data can be taken from different tables depending on a parameter value, which is difficult to specify with a Sub-Select.

The following shows the use of a Function with the same data as used in the above example. The number of top rows could be a parameter.

```
CREATE FUNCTION dbo.ufn_Top3Customer (@SalesPersonID INT)
RETURNS TABLE
AS
RETURN  SELECT      TOP (3)
                    CustomerID,
                    Amount
                    FROM dbo.Sale
                    WHERE SalesPersonID = @SalesPersonID
                    ORDER BY Amount DESC;
GO
SELECT    *
          FROM dbo.SalesPerson CROSS APPLY dbo.ufn_Top3Customer (SalesPerson.SalesPersonid)          AS TopSale;
GO
SELECT    *
          FROM dbo.SalesPerson OUTER APPLY dbo.ufn_Top3Customer (SalesPerson.SalesPersonid)          AS TopSale;
```

Another use of APPLY JOIN is to remove duplicates in a table, where duplicates are removed by comparing only some of the columns from the result. Therefore, DISTINCT cannot be used. In the example, there are 5 columns in the table. There should only be one row in the results table for each value of the combination of the columns Firstname, Lastname, and Zipcode. In the table we have several rows where the columns have the values Hanne/Carlsen/2000. Only one row must be included in the result.

```
CREATE TABLE dbo.Person
(
          PersonID            INT                 NOT NULL
                              CONSTRAINT PK_Person PRIMARY KEY,
```

```
        Firstname       VARCHAR (20)        NOT NULL,
        Lastname        VARCHAR (20)        NOT NULL,
        Address         VARCHAR (30)        NOT NULL,
        Zipcode         SMALLINT            NOT NULL
);
GO
INSERT INTO dbo.Person VALUES
        (1, 'Jens', 'Olsen', 'Nygade 3', 8000),
        (2, 'Jens', 'Olsen', 'Nygade 3 2. tv.', 8000),
        (3, 'Ida', 'Larsen', 'Vestergade 6', 2000),
        (4, 'Lars', 'Knudsen', 'Torvet 13', 9000),
        (5, 'Hanne', 'Carlsen', 'Borgergade 7', 2000),
        (6, 'Hanne', 'Carlsen', 'Borgergade 7 1. th', 2000),
        (7, 'Hanne', 'Carlsen', 'Borgergade 7', 2000),
        (8, 'Tina', 'Knudsen', 'Torvet 13', 9000);
```

The solution is to define a Sub-Select where only one row is returned for each combination of the same values in the three columns. With this intermediate result called DistinctData, an APPLY JOIN is performed. DistinctData is the left table and with a Sub-Select that returns TOP 1 as the right table.

```
SELECT  Data.*
        FROM (SELECT    DISTINCT
                        Zipcode,
                        Firstname,
                        Lastname
                        FROM dbo.Person)    AS DistinctData
                                CROSS APPLY
                                        (SELECT    TOP (1)
                                                   *
                                                   FROM dbo.Person
                                                   WHERE    Person.Zipcode = DistinctData.Zipcode                AND
                                                            Person.Firstname = DistinctData.Firstname            AND
                                                            Person.Lastname = DistinctData.Lastname
                                                   ORDER BY Person.PersonID)                AS Data
        ORDER BY Zipcode, Firstname, Lastname;
```

The result can be seen in the following table, where only 5 of the original 8 rows from the table dbo.Person are included. Since there is not immediately a column in the table that indicates the newest or oldest row based on a column with the DATE data type, the PersonID column is used. It is assumed that the highest value of the PersonID column is the most recent information. Remember that we only are using DISTINCT on some of the columns and the other columns - in the example only the column Zipcode - can have different values.

PersonID	Firstname	Lastname	Address	Zipcode
5	Hanne	Carlsen	Borgergade 7	2000
3	Ida	Larsen	Vestergade 6	2000
1	Jens	Olsen	Nygade 3	8000
4	Lars	Knudsen	Torvet 13	9000
8	Tina	Knudsen	Torvet 13	9000

In the following, we look at an example where data is used in a report. Since a salesperson can have many customers, the report becomes too large/has too many lines. For the individual salesperson, the customers with the largest amount are displayed, but only for the number of customers specified in the table dbo. SalesPerson in the NumberTop column. This value is different for the individual salespersons, because some have many customers and others have few customers.

However, to show the entire revenue, we summarize the revenue for the customers who are not the top customers for each SalesPerson. The tables and data are shown below.

```
CREATE TABLE dbo.SalesPerson
(
        SalesPersonID       INT                 NOT NULL
                            CONSTRAINT PK_SalesPerson PRIMARY KEY,
        Name                VARCHAR (30)        NOT NULL,
        NumberTop           INT                 NOT NULL
);

CREATE TABLE dbo.Sale
(
        SalesID             INT                 NOT NULL            IDENTITY
                            CONSTRAINT PK_Sale PRIMARY KEY,
        CustomerID          INT                 NOT NULL,
        Amount              INT                 NOT NULL,
```

```
        SalesPersonID       INT                 NULL
                            CONSTRAINT FK_Sale_SalesPerson FOREIGN KEY REFERENCES dbo.SalesPerson (SalesPersonID)
);
GO
INSERT INTO dbo.SalesPerson (SalesPersonID, Name, NumberTop) VALUES
        (1, 'Ole Jensen', 4),   (2, 'Ida Hansen', 7),   (3, 'Ane Sørensen', 2),         (4, 'Per Olsen', 2),    (5, 'Lars Hansen', 4);

INSERT INTO dbo.Sale (CustomerID, Amount, SalesPersonID) VALUES
        (23, 100, 1), (63, 200, 1), (13, 400, 1), (56, 700, 1), (78, 200, 1), (113, 10, 1), (256, 70, 1), (378, 20, 1),
        (18, 300, 2), (53, 500, 2), (17, 500, 2), (14, 700, 2), (34, 100, 2), (76, 100, 2), (36, 100, 2), (59, 100, 2), (46, 100, 2), (99, 100, 2), (98, 100, 2),
        (49, 200, 4),
        (46, 200, 5), (76, 300, 5), (23, 800, 5),
        (144, 200, NULL), (122, 400, NULL), (117, 210, NULL);
```

The following statement shows the solution. For each Salesperson, TopData contains all the customers whose detailed data must be included in the result. With NotTopData, we have all the rows grouped into one row for each Salesperson. With a UNION ALL of TopData and the aggregated NotTopData, the result to be returned is formed. This is done in the CTE named Result.

To make the final result directly usable in a report, the result can be joined with the dbo.SalesPerson table to display the name of the SalesPerson. In addition, a reasonable sort order is formed, where the sum for NotTopData always becomes the last row for the SalesPerson by using a Sub-Select in ORDER BY. In addition, there are sales where no SalesPersons is specified. These sales are aggregated and listed as one row with the customer name 'Unknown SalesPerson' and shown as the last row in the result.

```
WITH
TopData
AS
(
SELECT  SalesPerson.SalesPersonID,
        Sale.CustomerID,
        Sale.Amount
        FROM dbo.SalesPerson CROSS APPLY
                                (SELECT  TOP (SalesPerson.NumberTop)
                                         Sale.CustomerID,
                                         Sale.Amount
                                         FROM dbo.Sale
                                         WHERE Sale.SalesPersonID = SalesPerson.SalesPersonID
                                         ORDER BY Amount DESC)      AS Sale
),
NotTopData
AS
(
SELECT  Sale.SalesPersonID,
        Sale.CustomerID,
        Sale.Amount
        FROM dbo.Sale
EXCEPT
SELECT  TopData.SalesPersonID,
        TopData.CustomerID,
        TopData.Amount
        FROM TopData
),
Result
AS
(
SELECT  TopData.SalesPersonID,
        TopData.CustomerID,
        TopData.Amount,
        'TopSale'           AS Type
        FROM TopData
UNION ALL
SELECT  SalesPersonID,
        NULL AS CustomerID,
        SUM (Amount) AS Amount,
        CONCAT ('Sum of ', COUNT (*), ' Customers, which is not in Top Sale')
        FROM NotTopData
        GROUP BY SalesPersonID
)
SELECT  SalesPerson.SalesPersonID,
        COALESCE (SalesPerson.Name, 'Unknown SalesPerson ')     AS SalesPersonName,
        Result.CustomerID,
        Result.Amount,
        Type
```

```
        FROM Result LEFT JOIN dbo.SalesPerson ON Result.SalesPersonID = SalesPerson.SalesPersonID


        ORDER BY  COALESCE (SalesPerson.SalesPersonID, (SELECT        MAX (SalesPersonID) + 1
                                                      FROM dbo.Sale)),
                  COALESCE (Result.CustomerID, (SELECT      MAX (CustomerID) + 1
                                                FROM dbo.Sale));
```

The result from executing the query is the following table.

SalesPersonID	SalesPersonName	CustomerID	Amount	Type
1	Ole Jensen	13	400	TopSale
1	Ole Jensen	56	700	TopSale
1	Ole Jensen	63	200	TopSale
1	Ole Jensen	78	200	TopSale
1	Ole Jensen	NULL	200	Sum of 4 Customers, which is not in Top Sale
2	Ida Hansen	14	700	TopSale
2	Ida Hansen	17	500	TopSale
2	Ida Hansen	18	300	TopSale
2	Ida Hansen	34	100	TopSale
2	Ida Hansen	36	100	TopSale
2	Ida Hansen	53	500	TopSale
2	Ida Hansen	76	100	TopSale
2	Ida Hansen	NULL	400	Sum of 4 Customers, which is not in Top Sale
4	Per Olsen	49	200	TopSale
5	Lars Hansen	23	800	TopSale
5	Lars Hansen	46	200	TopSale
5	Lars Hansen	76	300	TopSale
NULL	Unknown SalesPerson	NULL	810	Sum of 3 Customers, which is not in Top Sale

The result should probably not look like this in the real world, but the result shows the information that can be included. For the rows with the sum for NotTopData, the CustomerID column is displayed as NULL. In the ORDER BY clause, a value is calculated that is greater than the largest value of all values in the CustomerID column. This ensures that the sum is always the last row, regardless of the values for the CustomerID column inserted in the table. The same principle has been applied to information about SalesPerson.

In the Sub-Select statement we can of course use PERCENT.

```
SELECT    *
          FROM dbo.SalesPerson CROSS APPLY
                                    (SELECT    TOP (50) PERCENT
                                               CustomerID,
                                               Amount
                                               FROM dbo.Sale
                                               WHERE Sale.SalesPersonID = SalesPerson.SalesPersonID
                                               ORDER BY Amount DESC)          AS t;
```

As previously mentioned, a TableValued Function can be used instead of a Sub-Select. This is especially useful if some more complex operations are performed. In the next example, all the salespersons are listed. For each person, the TOP 3 customers for a given year are displayed. When we look at the TOP 3 customers for the year 2021, we must also include the same customers' purchases from year 2020 in the result. It is not necessary the TOP 3 customers from 2020, but the same customers as from 2021. Maybe the customers did not buy anything in the year 2020, so the column can be NULL.

The following data is used to demonstrate the use of a FUNCTION in conjunction with APPLY JOIN.

```
CREATE TABLE dbo.SalesPerson
(
          SalesPersonID       INT                 NOT NULL
                              CONSTRAINT PK_SalesPerson PRIMARY KEY,
          Name                VARCHAR (30)        NOT NULL
);

CREATE TABLE dbo.Sale
(
          SalesID             INT                 NOT NULL            IDENTITY,
          CustomerID          INT                 NOT NULL,
          Amount              INT                 NOT NULL,
          Year                SMALLINT            NOT NULL,
          SalesPersonID       INT                 NOT NULL
                              CONSTRAINT FK_Sale_SalesPerson FOREIGN KEY REFERENCES dbo.SalesPerson (SalesPersonID),
```

```
        CONSTRAINT PK_Sale PRIMARY KEY (SalesPersonID, CustomerID, Year)
);
GO

INSERT INTO dbo.SalesPerson (SalesPersonID, Name) VALUES
        (1, 'Ole Jensen'),      (2, 'Ida Hansen'),      (3, 'Ane Sørensen'),    (4, 'Lars Carlsen');

INSERT INTO dbo.Sale (CustomerID, Amount, Year, SalesPersonID) VALUES
        (23, 100, 2020, 1),     (23, 125, 2021, 1),
        (44, 200, 2019, 1),     (44, 175, 2020, 1),     (44, 325, 2021, 1),
        (36, 400, 2019, 1),     (36, 475, 2020, 1),     (36, 500, 2021, 1),
        (22, 125, 2019, 1),     (22, 350, 2021, 1),
        (75, 475, 2021, 1),

        (112, 100, 2020, 2),
        (102, 200, 2019, 2),    (102, 175, 2020, 2),
        (176, 300, 2019, 2),    (176, 333, 2020, 2),    (176, 366, 2021, 2),
        (135, 700, 2019, 2),    (135, 600, 2021, 2),
        (199, 30, 2018, 2),     (199, 525, 2021, 2),

        (223, 500, 2020, 3),    (223, 450, 2021, 3),
        (255, 300, 2019, 3),    (255, 250, 2020, 3),    (255, 450, 2021, 3),
        (288, 200, 2019, 3),    (288, 100, 2020, 3),    (288, 400, 2021, 3),
        (229, 125, 2020, 3),    (229, 350, 2021, 3),
        (237, 250, 2020, 3),    (237, 325, 2021, 3),

        (415, 175, 2021, 4),    (478, 125, 2021, 4),
        (415, 575, 2020, 4);
```

In the function we create the result table to be returned. SalesPersonID is not included in the table since the function is called once for each salesperson with SalesPersonID as the input parameter. In addition, year is a parameter, where we in the example use 2021.

First, the TOP 3 customers for the year 2021 are selected. Afterwards, the result table is updated with data from the year 2020, but only for the same customers, if this data exists.

```
CREATE FUNCTION dbo.TopSale
(
        @SalesPersonID      INT,
        @Year               SMALLINT
)
RETURNS @TopSaleData TABLE
        (
        CustomerID          INT             NOT NULL,
        AmountLastYear      NT              NULL,
        YearLast            SMALLINT        NULL,
        AmountActualYear    INT             NOT NULL,
        YearActual          SMALLINT        NOT NULL
        )
AS
BEGIN
    INSERT INTO @TopSaleData (CustomerID, AmountActualYear, YearActual)
        SELECT  TOP (3)
                CustomerID,
                Amount,
                Year
                FROM dbo.Sale
                WHERE   SalesPersonID = @SalesPersonID          AND
                        Year = @Year
                ORDER BY Amount DESC;

    UPDATE @TopSaleData
        SET AmountLastYear = Sale.Amount, YearLast = Sale.Year
        FROM dbo.Sale INNER JOIN @TopSaleData AS TsD    ON Sale.CustomerID = Tsd.CustomerID
        WHERE   Sale.SalesPersonID = @SalesPersonID     AND
                Sale.Year = @Year - 1;

    RETURN;
END;
```

The function can be used in a SELECT statement.

```
SELECT  *
        FROM dbo.SalesPerson OUTER APPLY dbo.TopSale (SalesPerson.SalesPersonID, 2021)  AS t3Sale
        ORDER BY SalesPerson.SalesPersonID, t3Sale.CustomerID;
```

SalesPersonID	Name	CustomerID	AmountLastYear	YearLast	AmountActualYear	YearActual
1	Ole Jensen	22	NULL	NULL	350	2021
1	Ole Jensen	36	475	2020	500	2021
1	Ole Jensen	75	NULL	NULL	475	2021
2	Ida Hansen	135	NULL	NULL	600	2021
2	Ida Hansen	176	333	2020	366	2021
2	Ida Hansen	199	NULL	NULL	525	2021
3	Ane Sørensen	223	500	2020	450	2021
3	Ane Sørensen	255	250	2020	450	2021
3	Ane Sørensen	288	100	2020	400	2021
4	Lars Carlsen	415	575	2020	175	2021
4	Lars Carlsen	478	NULL	NULL	125	2021

The Function can be modified so that data is not taken from the last year, but from the most recent year relative to the year specified in the @Year parameter. An APPLY JOIN is used in the Function because the largest recent year is not known. In the example, it could be 2018, 2019 or 2020.

```
CREATE OR ALTER FUNCTION dbo.TopSale
(
        @SalesPersonID      INT,
        @Year               SMALLINT
)
RETURNS @TopSaleData TABLE
        (
        CustomerID          INT             NOT NULL,
        AmountLatestYear    INT             NULL,
        YearLatest          SMALLINT        NULL,
        AmountActualYear    INT             NOT NULL,
        YearActual          SMALLINT        NOT NULL
        )
AS
BEGIN
    INSERT INTO @TopSaleData (CustomerID, AmountActualYear, YearActual)
        SELECT  TOP (3)
                CustomerID,
                Amount,
                Year
                FROM dbo.Sale
                WHERE   SalesPersonID = @SalesPersonID          AND
                        Year = @Year
                ORDER BY Amount DESC;

    UPDATE @TopSaleData
        SET AmountLatestYear = Top1Sale.Amount, YearLatest = Top1Sale.Year
        FROM @TopSaleData AS TS CROSS APPLY
                                (SELECT  TOP (1)
                                Amount,
                                Year
                                FROM dbo.Sale
                                WHERE   SalesPersonID = @SalesPersonID          AND
                                        Year < @Year                            AND
                                        TS.CustomerID = Sale.CustomerID
                                ORDER BY Sale.Year DESC ) AS Top1Sale
        RETURN;
END;
GO
SELECT  *
        FROM dbo.SalesPerson OUTER APPLY dbo.TopSale (SalesPerson.SalesPersonID, 2021)  AS t3Sale
        ORDER BY SalesPerson.SalesPersonID, t3Sale.CustomerID;
```

In the result, the most recent year will not necessarily be 2020. The customer with CustomerID 199 did not buy anything in 2019 and 2020. The most recent year/latest year is 2018.

SalesPersonID	Name	CustomerID	AmountLatestYear	YearLatest	AmountActualYear	YearActual
1	Ole Jensen	22	125	2019	350	2021
1	Ole Jensen	36	475	2020	500	2021
1	Ole Jensen	75	NULL	NULL	475	2021
2	Ida Hansen	135	700	2019	600	2021
2	Ida Hansen	176	333	2020	366	2021
2	Ida Hansen	199	30	2018	525	2021
3	Ane Sørensen	223	500	2020	450	2021
3	Ane Sørensen	255	250	2020	450	2021
3	Ane Sørensen	288	100	2020	400	2021
4	Lars Carlsen	415	575	2020	175	2021

4	Lars Carlsen	478	NULL	NULL	125	2021

Hopefully these examples demonstrate the use of APPLY JOIN, which cannot be simply defined without the APPLY JOIN operation.