CHAPTER 12

Working with Hierarchical Data and HierarchyID

Modeling and working with data hierarchies have long been requirements for SQL Server developers. Traditionally, hierarchical data has been modeled using a self-join on a table, between two columns. One column contains the ID of the hierarchical member, and the other, the ID of its parent hierarchical member. Newer versions of SQL Server (2008 and later versions) offer `HierarchyID`, however. `HierarchyID` is a data type written in .NET and exposed in SQL Server. Using `HierarchyID` can offer performance benefits and simplified code, compared to using a table with a self-join. The data type exposes many methods that can be called against the data, to allow developers to easily determine the ancestors and descendants of a hierarchical member, as well as determine other useful information, such as the level of a specific hierarchical member within the hierarchy.

In this chapter, we will examine first the use cases for hierarchical data. I will discuss how to model a traditional hierarchy, before explaining how we can remodel it using `HierarchyID`. Finally, we will look at the methods that are exposed against the `HierarchyID` data type.

Hierarchical Data Use Cases

There are many data requirements that need a hierarchy to be maintained. For example, consider an employee hierarchy, modeled from an organizational chart (see Figure 12-1). A human resources department may have reporting requirements, to determine how many staff directly or indirectly report to a manager. It may also have to report on more complex requirements, such as how much revenue has been generated by staff reporting to each group head.

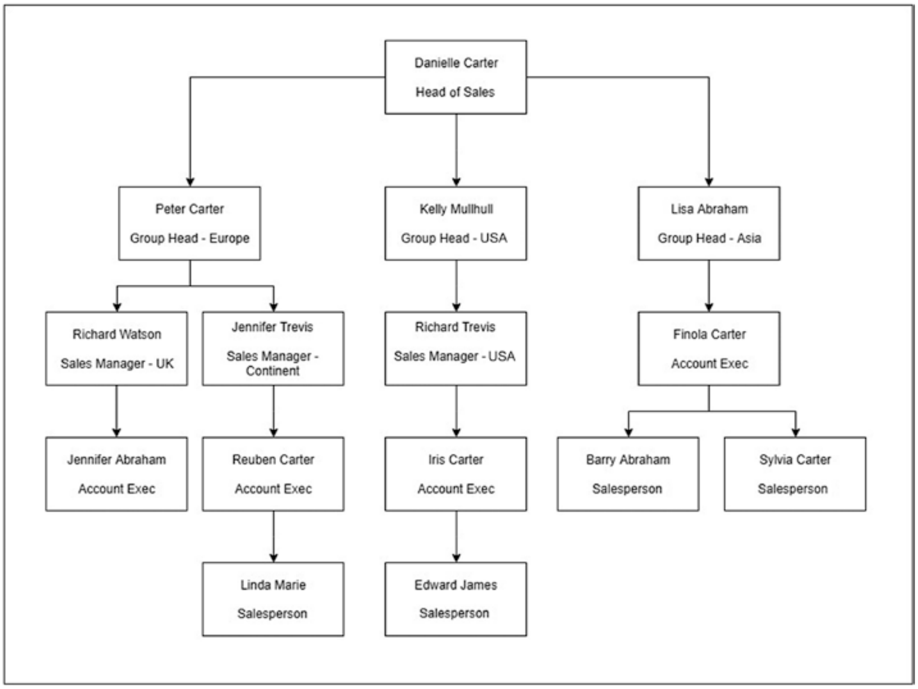


Figure 12-1. *Sales department organization chart*

Another classic use case for hierarchical data is a bill of materials (BoM). A BoM defines a hierarchy of parts that are required to produce a product. For example, Figure 12-2 illustrates a simple BoM for a home computer. A computer manufacturer would have to maintain a hierarchy of these parts for stock reporting.

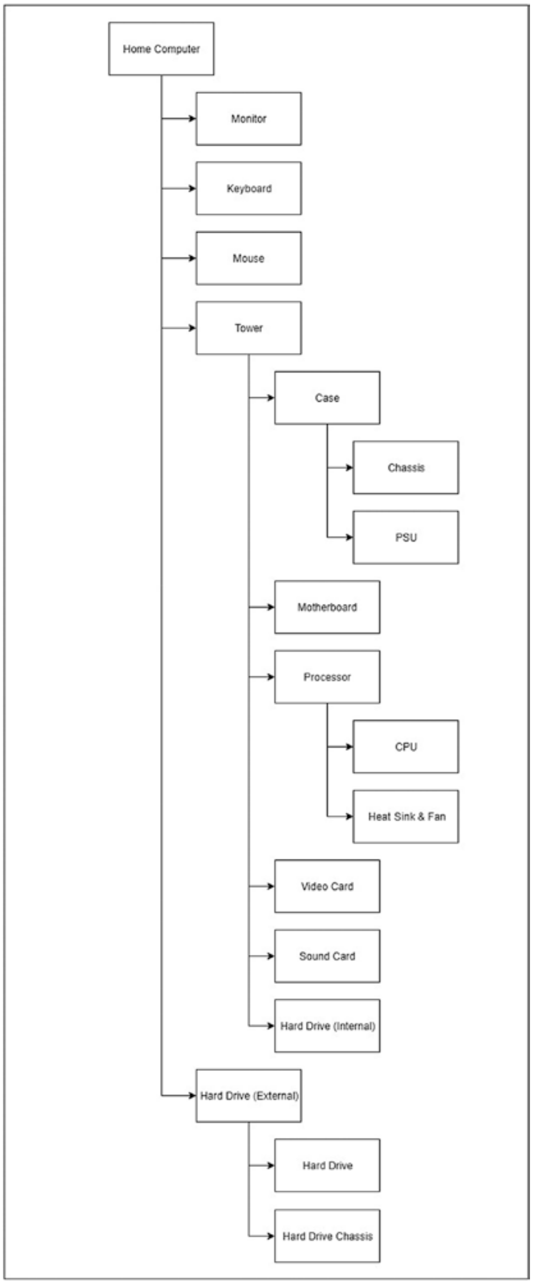


Figure 12-2. *Bill of materials*

In this chapter, we will be working with the example of a sales area hierarchy. As illustrated in Figure 12-3, we will be modeling global sales regions. The hierarchy is ragged, meaning that there can be a varying number of levels in each branch of the hierarchy.

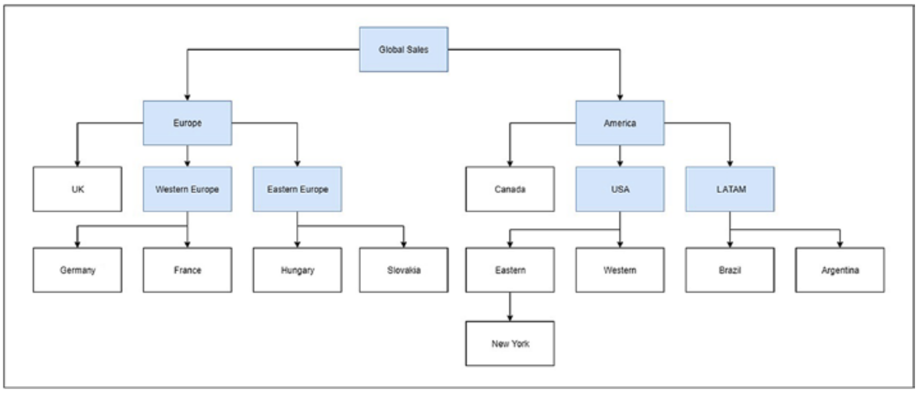


Figure 12-3. *Sales area hierarchy*

Maintaining a sales area hierarchy is important to many companies, as it allows them to report on many factors, from revenue taken in each region to the number of salespeople in each region to average revenue per salesperson and region, etc. In this example, as well as in standard sales regions, there are areas of the hierarchy that are aggregation areas. This means that no salespeople exist, and sales are not directly taken for these regions. Instead, they are reporting levels, to allow lower levels of the hierarchy to be rolled up. These aggregation areas are highlighted in blue.

Modeling Traditional Hierarchies

Let's look at how we might model a sales area hierarchy, using a traditional approach. To do this, consider the data in Table 12-1.

Table 12-1. *Sales Area Hierarchy*

SalesAreaID	ParentSalesAreaID	SalesArea Name	CountOfSalesPeople	SalesYTD
1	NULL	Global Sales	NULL	NULL
2	1	Europe	NULL	NULL
3	1	America	NULL	NULL
4	2	UK	3	300,000
5	2	Western Europe	NULL	NULL
6	2	Eastern Europe	NULL	NULL
7	3	Canada	4	350,000
8	3	USA	NULL	NULL
9	3	LATAM	NULL	NULL
10	5	Germany	3	150,000
11	5	France	2	100,000
12	6	Hungary	1	50,000
13	6	Slovakia	2	80,000
14	8	Eastern	4	140,000
15	8	Western	3	280,000
16	9	Brazil	1	100,000
17	9	Argentina	2	70,000
18	14	New York	2	120,000

In the preceding table, the SalesAreaID column would be the primary key of the table, and the ParentSalesAreaID column would be a foreign key, which references the SalesAreaID column, creating what is known as a self-join.

Note The ParentSalesAreaID column is `NULL` for the GlobalSales area, because it does not have a parent. This is known as the root of the hierarchy.

The SalesAreaName column describes the sales area, while the CountOfSalesPeople and SalesYTD columns detail how each sales area is performing. We will use these columns to explore how to work with hierarchical data. You will notice that the CountOfSalesPeople and SalesYTD columns are populated with `NULL` values for aggregation areas. This is because they are "virtual" areas, where no salespeople are based.

This table can be created by using the script in Listing 12-1.

Listing 12-1. Creating a Traditional Hierarchical Table

```
USE WideWorldImporters
GO

CREATE TABLE Sales.SalesAreaTraditionalHierarchy
(
        SalesAreaID        INT    NOT NULL    PRIMARY KEY,
        ParentSalesAreaID  INT    NULL
                                       REFERENCES Sales.Sales
                                       AreaTraditionalHierarch
                                       y(SalesAreaID),
        SalesAreaName            NVARCHAR(20)    NOT NULL,
        CountOfSalesPeople       INT             NULL,
        SalesYTD                 MONEY           NULL
) ;

INSERT INTO Sales.SalesAreaTraditionalHierarchy (
          SalesAreaID
        , ParentSalesAreaID
        , SalesAreaName
        , CountOfSalesPeople
        , SalesYTD
)
VALUES
(1, NULL, 'GlobalSales', NULL, NULL),
(2, 1, 'Europe', NULL, NULL),
(3, 1, 'America', NULL, NULL),
(4, 2, 'UK', 3, 300000),
(5, 2, 'Western Europe', NULL, NULL),
(6, 2, 'Eastern Europe', NULL, NULL),
(7, 3, 'Canada', 4, 350000),
(8, 3, 'USA', NULL, NULL),
(9, 3, 'LATAM', NULL, NULL),
(10, 5, 'Germany', 3, 150000),
(11, 5, 'France', 2, 100000),
(12, 6, 'Hungary', 1, 50000),
(13, 6, 'Slovakia', 2, 80000),
(14, 8, 'Eastern', 4, 140000),
(15, 8, 'Western', 3, 280000),
(16, 9, 'Brazil', 1, 100000),
(17, 9, 'Agentina', 2, 70000),
(18, 14, 'New York', 2, 120000) ;
```

Let's imagine that we must answer a business question, using this traditional hierarchy. For example, we may have to answer the question, "What is the total of SalesYTD for the America region?" To determine this, we would have to write a query that joins the table to itself and rolls up the SalesYTD column for all subregions under the America region.

The best way of achieving this is to use a recursive CTE (common table expression). A CTE is a temporary result set that is defined within the context of a `SELECT`, `UPDATE`, `INSERT`, `DELETE`, or `CREATE VIEW` statement and can only be referenced within that query. It is similar to using a subquery as a derived table but has the benefit that it can be referenced multiple times and also reference itself. When a CTE references itself, it becomes recursive.

A CTE is declared using a `WITH` statement, which specifies the name of the CTE, followed by the column list that will be returned by the CTE, in parentheses. The `AS` keyword is then used, followed by the body of the CTE, again within parentheses. This is shown, for our example, in Listing 12-2.

Tip The queries in Listings 12-2 to 12-14 are not meant to be run separately. Listing 12-15 brings them all together, into a useable script.

Listing 12-2. CTE Structure

```
; WITH AreaHierarchy (SalesAreaID, SalesYTD, ParentSalesAreaID)
AS
(
        [Body of CTE]
)
```

The definition of a recursive CTE has two queries, joined with a `UNION ALL` clause. The first query is known as the anchor query and defines the initial result set. The second query is known as the recursive query and references the CTE. The first level of recursion will join to the anchor query, and subsequent levels of recursion will join to the level of recursion immediately above them.

Therefore, in our example, the anchor query will have to return the SalesAreaID of the America sales area. Because all queries joined with `UNION ALL` must contain the same number of columns, our anchor query must also return the ParentSalesAreaID and SalesYTD columns, even though these values will be `NULL`.

The query in Listing 12-3 shows the required anchor query.

Listing 12-3. Anchor Query

```
SELECT
          SalesAreaID
        , SalesYTD
        , ParentSalesAreaID
FROM Sales.SalesAreaTraditionalHierarchy RootLevel
WHERE SalesAreaName = 'America'
```

The recursive query will return the same column list as the anchor query but will include a `JOIN` clause, which joins the ParentSalesAreaID column in the recursive query to the SalesAreaID column of the CTE, as demonstrated in Listing 12-4.

Listing 12-4. Recursive Query

```
SELECT
          Area.SalesAreaID
        , Area.SalesYTD
        , Area.ParentSalesAreaID
FROM Sales.SalesAreaTraditionalHierarchy Area
INNER JOIN AreaHierarchy
        ON Area.ParentSalesAreaID = AreaHierarchy.SalesAreaID
```

Following the declaration of this CTE, we will be able to run a `SELECT` statement, which rolls up the SalesYTD, to return the total of SalesYTD for the whole of America. The script in Listing 12-5 brings all these components together.

Listing 12-5. Bringing It All Together

```
WITH AreaHierarchy (SalesAreaID, SalesYTD, ParentSalesAreaID)
AS
(
        SELECT
                  SalesAreaID
                , SalesYTD
                , ParentSalesAreaID
      FROM Sales.SalesAreaTraditionalHierarchy RootLevel
      WHERE SalesAreaName = 'America'
      UNION ALL
      SELECT
                Area.SalesAreaID
        , Area.SalesYTD
        , Area.ParentSalesAreaID
      FROM Sales.SalesAreaTraditionalHierarchy Area
      INNER JOIN AreaHierarchy
        ON Area.ParentSalesAreaID = AreaHierarchy.SalesAreaID
)
SELECT SUM(SalesYTD)
FROM AreaHierarchy ;
```

Modeling Hierarchies with HierarchyID

When modeling hierarchical data using `HierarchyID`, there is no need to perform a self-join against a table. Therefore, instead of having a column that references its parent area's primary key, we will instead have a `HierarchyID` column, which defines each area's position within the hierarchy. To explain this concept further, let's consider the data in Table 12-2.

Table 12-2. *Sales Area Hierarchy with* `HierarchyID`

SalesAreaID	SalesAreaHierarchy	SalesAreaName	CountOfSalesPeople	SalesYTD
1	/	GlobalSales	NULL	NULL
2	/1/	Europe	NULL	NULL
3	/2/	America	NULL	NULL
4	/1/1/	UK	3	300,000
5	/1/2/	Western Europe	NULL	NULL
6	/1/3/	Eastern Europe	NULL	NULL
7	/2/1/	Canada	4	350,000
8	/2/2/	USA	NULL	NULL
9	/2/3/	LATAM	NULL	NULL
10	/1/2/1/	Germany	3	150,000
11	/1/2/2/	France	2	100,000
12	/1/3/1/	Hungary	1	50,000
13	/1/3/2/	Slovakia	2	80,000
14	/2/2/1/	Eastern	4	140,000
15	/2/2/2/	Western	3	280,000
16	/2/3/1/	Brazil	1	100,000
17	/2/3/2/	Argentina	2	70,000
18	/2/2/1/1/	New York	2	120,000

In Table 12-2, you will notice that the hierarchy is represented by the format `/[Node]/[Child Node]/[GrandchildNode]/`, in which a row containing just `/` is the root of the hierarchy. For example, we can see that the value `/1/2/1/` for Germany tells us that Germany is a child of `/1/2/`

(Western Europe), which in turn is a child of /1/ (Europe). /1/ is the child of / (Global Sales), which is the root of the hierarchy.

We can create and populate this table by using the script in Listing 12-6.

Listing 12-6. Create Table with HierarchyID

```
USE WideWorldImporters
GO

CREATE TABLE Sales.SalesAreaHierarchyID
(
SalesAreaID               INT                     NOT
NULL    PRIMARY KEY,
SalesAreaHierarchy   HIERARCHYID     NOT NULL,
SalesAreaName          NVARCHAR(20)    NOT NULL,
CountOfSalesPeople    INT                     NULL,
SalesYTD                    MONEY                  NULL
) ;

INSERT INTO Sales.SalesAreaHierarchyID (
          SalesAreaID
        , SalesAreaHierarchy
        , SalesAreaName
        , CountOfSalesPeople
        , SalesYTD
)
VALUES
(1, '/', 'GlobalSales', NULL, NULL),
(2, '/1/', 'Europe', NULL, NULL),
(3, '/2/', 'America', NULL, NULL),
(4, '/1/1/', 'UK', 3, 300000),
(5, '/1/2/', 'Western Europe', NULL, NULL),
(6, '/1/3/', 'Eastern Europe', NULL, NULL),
(7, '/2/1/', 'Canada', 4, 350000),
(8, '/2/2/', 'USA', NULL, NULL),
(9, '/2/3/', 'LATAM', NULL, NULL),
(10, '/1/2/1/', 'Germany', 3, 150000),
(11, '/1/2/2/', 'France', 2, 100000),
(12, '/1/3/1/', 'Hungary', 1, 50000),
(13, '/1/3/2/', 'Slovakia', 2, 80000),
(14, '/2/2/1/', 'Eastern', 4, 140000),
(15, '/2/2/2/', 'Western', 3, 280000),
(16, '/2/3/1/', 'Brazil', 1, 100000),
(17, '/2/3/2/', 'Agentina', 2, 70000),
(18, '/2/2/1/1/', 'New York', 2, 120000) ;
```

Even though we have inserted human-readable strings into the SalesAreaHierarchyID column, SQL Server converts these strings and stores them as hexadecimal values. This makes the column extremely compact and efficient. The size of the `HierarchyID` column and that of the INT column used for the ParentSalesAreaID column in the traditional hierarchy can be compared using the query in Listing 12-7.

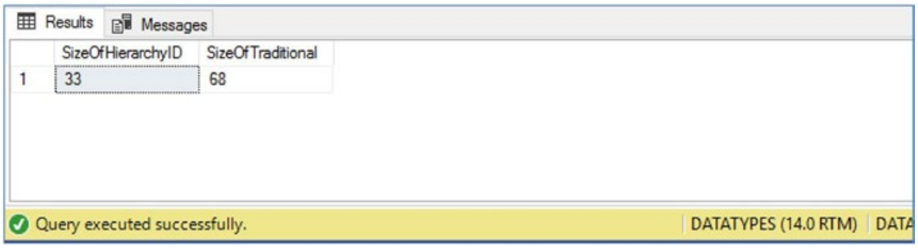
Listing 12-7. Comparing the Size of a Traditional Hierarchy to `HierarchyID`

```
USE WideWorldImporters
GO

SELECT
          SUM(DATALENGTH(salesareahierarchy)) AS SizeOf
          HierarchyID
        , SUM(DATALENGTH(parentsalesareaid)) AS SizeOf
        Traditional
FROM Sales.SalesAreaHierarchyID SalesAreaHierarchy
```

```
INNER JOIN sales.SalesAreaTraditionalHierarchy SalesAreaTraditional
        ON SalesAreaHierarchy.SalesAreaID = SalesArea
        Traditional.SalesAreaID ;
```

The results of this query are shown in Figure 12-4. You can see that the `HierarchyID` column is less than half the size of the INT column used for the ParentSalesAreaID.



	SizeOfHierarchyID	SizeOfTraditional
1	33	68

Query executed successfully. | DATATYPES (14.0 RTM) | DATA

Figure 12-4. *Results of size comparison*

Note In Chapter 1, you learned that it is important to use the correct data type, and if I had chosen to use a `SMALLINT` for the ParentSalesAreaID column, the two columns would be about the same size. This is, however, a minor example, provided for the purpose of explaining `HierarchyID`, but if you are implementing hierarchies on a large scale, this example is a fair representation.

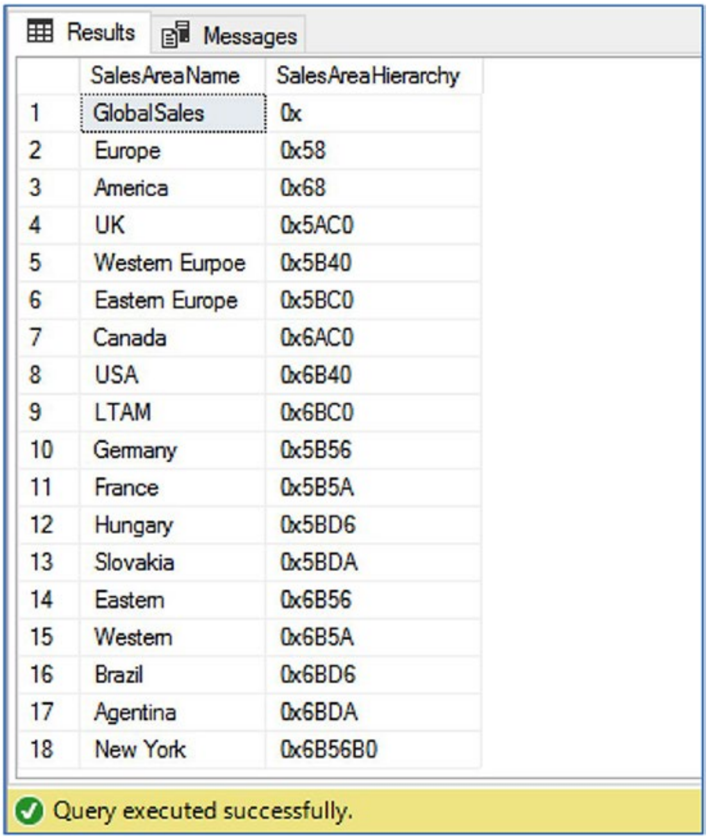
If we run a normal `SELECT` statement against the SalesAreaHierarchyID table, we can see the hexadecimal values in their raw form. For example, consider the query in Listing 12-8.

Listing 12-8. `SELECT` Statement Against `HierarchyID` Column

```
USE WideWorldImporters
GO

SELECT
          SalesAreaName
        , SalesAreaHierarchy
FROM Sales.SalesAreaHierarchyID ;
```

The results of this query are displayed in Figure 12-5. You will notice that the contents of the SalesAreaHierarchy column are returned as hexadecimal values, instead of human-readable strings. In order to view the human-readable strings that we entered, we must use the `ToString()` method, which is discussed in the "Working with `HierarchyID` Methods" section of this chapter.



	SalesAreaName	SalesAreaHierarchy
1	GlobalSales	0x
2	Europe	0x58
3	America	0x68
4	UK	0x5AC0
5	Western Eurpoe	0x5B40
6	Eastern Europe	0x5BC0
7	Canada	0x6AC0
8	USA	0x6B40
9	LTAM	0x6BC0
10	Germany	0x5B56
11	France	0x5B5A
12	Hungary	0x5BD6
13	Slovakia	0x5BDA
14	Eastern	0x6B56
15	Western	0x6B5A
16	Brazil	0x6BD6
17	Agentina	0x6BDA
18	New York	0x6B56B0

Query executed successfully.

Figure 12-5. *Results of `SELECT` statement against `HierarchyID` column*

HierarchyID Methods

A number of methods are exposed against the `HierarchyID` data type, allowing developers to quickly write efficient code when working with hierarchies. Table 12-3 details these methods.

Table 12-3. *Methods Exposed Against the `HierarchyID` Data Type*

Method	**Description**
`GetAncestor`	Returns the ancestor of a hierarchy node. Accepts a parameter that defines how many levels up the hierarchy the ancestor should be returned from. For example, `GetAncestor(1)` will return the node's parent, while `GetAncestor(2)` will return the node's grandparent.
`GetDescendant`	Returns a child node ID for a given node in the hierarchy. The `GetDescendant()` method is generally used in the creation of two nodes. Therefore, the method accepts two parameters, both of type `HierarchyID`. The generated node will sit between the two nodes specified.
`GetLevel`	Returns the hierarchical level of the node
`GetRoot`	A static method that returns the root level of a hierarchy
`IsDescendantOf`	The `IsDescendantOf()` method accepts a single parameter, of type `HierarchyID`, and returns `1` if a given node is a descendant of the node passed as a parameter.
`Parse`	Parses the string representation of a node, which is passed as a parameter, to ensure it is valid. If valid, it returns the hexadecimal representation. If invalid, it will throw an error.
`Read`	Reads the binary representation of `SqlHierarchyId` from the BinaryReader and sets the `SqlHierarchyId` object to that value. The `Read()` method can only be called from SQLCLR. It cannot be called from T-SQL. When using T-SQL, you should use `CAST` or `CONVERT` instead.

(*continued*)

Table 12-3. *(continued)*

Method	Description
`GetReparentedValue`	Used to move a node to a new parent. Accepts two parameters, the first being the original parent and the second being the new parent
`ToString`	Returns a string-formatted representation of a node within the hierarchy
`Write`	Writes out a binary representation of `SqlHierarchyId` to the BinaryWriter. For use with SQLCLR only. When using T-SQL, use `CAST` or `CONVERT` instead.

Tip `HierarchyID` methods are case-sensitive. For example, calling `tostring()` will throw an error; calling `ToString()` will succeed.

Working with HierarchyID Methods

The following sections describe how to use the methods exposed against the `HierarchyID` data type.

Using ToString( )

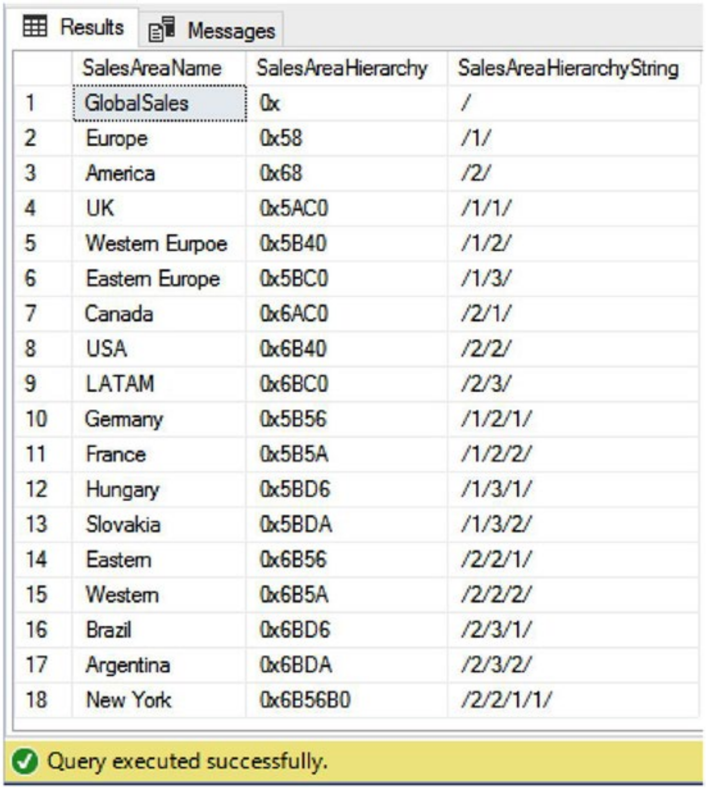
If you run a `SELECT` statement against a column with the `HierarchyID` data type, the value returned will be a hexadecimal representation of the node. To see a textual representation of the node, you must use the `ToString()` method. For example, consider the query in Listing 12-9.

Listing 12-9. Using `ToString()`

```
USE WideWorldImporters
GO

SELECT
          SalesAreaName
        , SalesAreaHierarchy
        , SalesAreaHierarchy.ToString() AS SalesArea
        HierarchyString
FROM Sales.SalesAreaHierarchyID ;
```

The results of this query are displayed in Figure 12-6. You will see that the column becomes human-readable, once the `ToString()` method is called against it.



Results Messages

	SalesAreaName	SalesAreaHierarchy	SalesAreaHierarchyString
1	GlobalSales	0x	/
2	Europe	0x58	/1/
3	America	0x68	/2/
4	UK	0x5AC0	/1/1/
5	Western Eurpoe	0x5B40	/1/2/
6	Eastern Europe	0x5BC0	/1/3/
7	Canada	0x6AC0	/2/1/
8	USA	0x6B40	/2/2/
9	LATAM	0x6BC0	/2/3/
10	Germany	0x5B56	/1/2/1/
11	France	0x5B5A	/1/2/2/
12	Hungary	0x5BD6	/1/3/1/
13	Slovakia	0x5BDA	/1/3/2/
14	Eastern	0x6B56	/2/2/1/
15	Western	0x6B5A	/2/2/2/
16	Brazil	0x6BD6	/2/3/1/
17	Argentina	0x6BDA	/2/3/2/
18	New York	0x6B56B0	/2/2/1/1/

Query executed successfully.

Figure 12-6. *Results of using `ToString()`*

Using Parse( )

The `Parse()` method is called implicitly when a string representation of a node is inserted into a `HierarchyID` column. Essentially, the `Parse()` method performs the reverse function of the `ToString()` method. It attempts to convert a string formatted representation to the `HierarchyID` representation. If it fails, an error is thrown. For example, consider the script in Listing 12-10.

Listing 12-10. Using the `Parse()` Method

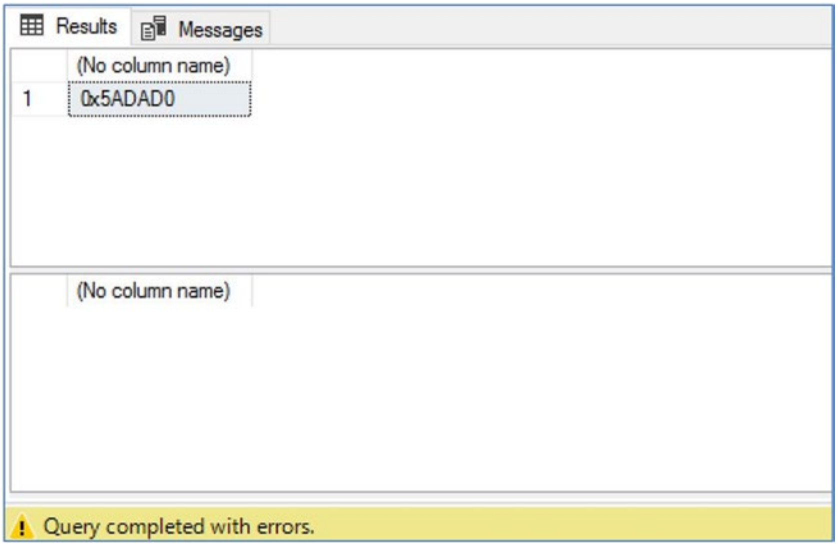
```
--Returns Hexidecimal Representation Of Node

SELECT HierarchyID::Parse('/1/1/2/2/') ;

--Throws An Error Because Trailing / Is Missing

SELECT HierarchyID::Parse('/1/1/2/2') ;
```

The Results tab displayed by running this script can be seen in Figure 12-7. While the first query displays the expected result, the second query returns no results.



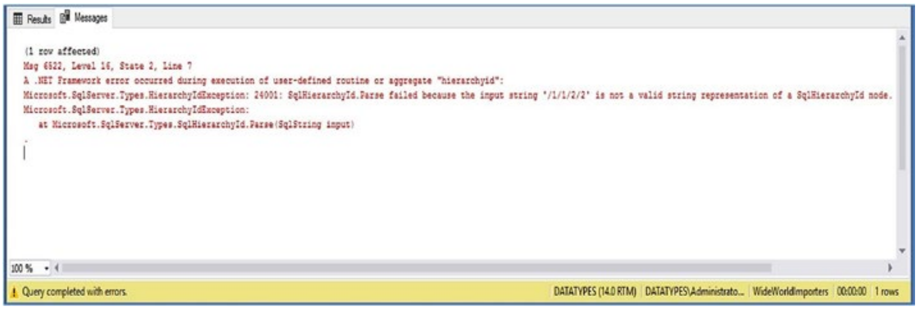
```
Results   Messages
   (No column name)
1  0x5ADAD0

   (No column name)

Query completed with errors.
```

Figure 12-7. *Using `Parse()` Results tab*

Checking the Messages tab, displayed in Figure 12-8, will detail the error thrown by the .NET framework.



```
(1 row affected)
Msg 6522, Level 16, State 2, Line 7
A .NET Framework error occurred during execution of user-defined routine or aggregate "hierarchyid":
Microsoft.SqlServer.Types.HierarchyIdException: 24001: SqlHierarchyId.Parse failed because the input string '/1/1/2/2' is not a valid string representation of a SqlHierarchyId node.
Microsoft.SqlServer.Types.HierarchyIdException:
   at Microsoft.SqlServer.Types.SqlHierarchyId.Parse(SqlString input)
.
```

```
Query completed with errors.   DATATYPES (14.0 RTM) | DATATYPES\Administrato... | WideWorldImporters | 00:00:00 | 1 rows
```

Figure 12-8. Error thrown by .NET framework

Using GetRoot( )

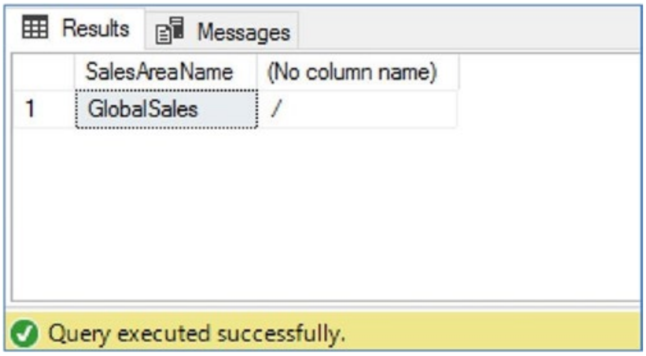
The `GetRoot()` method will return the root node of a hierarchy, as demonstrated in Listing 12-11.

Listing 12-11. Using `GetRoot()`

```
USE WideWorldImporters
GO

SELECT
           SalesAreaName
        ,  SalesAreaHierarchy.ToString()
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy = HierarchyID::GetRoot() ;
```

The results of this query can be viewed in Figure 12-9.



```
   SalesAreaName   (No column name)
1  GlobalSales     /
```

Query executed successfully.

***Figure 12-9.** Results of using `GetRoot()`*

Using GetLevel( )

The `GetLevel()` method allows you to determine at what level of the hierarchy a particular node resides. For example, the query in Listing 12-12 will return all nodes that reside on the bottom level of the hierarchy. In our case, this is just `New York`.

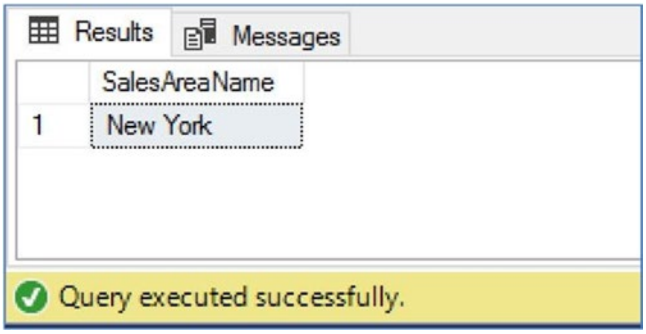
The subquery will determine the maximum level within the hierarchy, and the outer query will return all sales areas that are at that level.

Listing 12-12. Using `GetLevel()`

```
USE WideWorldImporters
GO

SELECT
        SalesAreaName
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy.GetLevel() =
        (
        SELECT
                MAX(SalesAreaHierarchy.GetLevel())
        FROM Sales.SalesAreaHierarchyID
        ) ;
```

The results of this query can be found in Figure 12-10.



	SalesAreaName
1	New York

Query executed successfully.

Figure 12-10. *Results of using* `GetLevel()`

Read( ) and Write( )

Because this chapter focuses on how to use `HierarchyID` within your T-SQL code and `the Read()` and `Write()` methods are only applicable to using SQLCLR (the technology that allows for managed objects to be created inside SQL Server), a full description of the `Read()` and `Write()` methods is beyond the scope of this book.

Using GetDescendant( )

Of course, a developer could insert a new node into the hierarchy, between existing nodes, but the `GetDescendant()` method helps a developer do this pragmatically. The method accepts two parameters, both of which can be `NULL` and represent existing children. The method will then generate a node value, using the following rules:

- If the parent is `NULL`, then a `NULL` value will be returned.
- If the parent is not `NULL`, and both parameters are `NULL`, the first child of the parent will be returned.

- If the parent and first parameter are not `NULL`, but the second parameter is `NULL`, a child of parent greater than the first parameter will be returned.
- If the parent and second parameter are not `NULL` but the first parameter is `NULL`, a child of parent smaller than the second parameter will be returned.
- If parent and both parameters are not `NULL`, a child of parent between the two parameters will be returned.
- If the first parameter is not `NULL` and not a child of the parent, an error is thrown.
- If the second parameter is not `NULL` and not a child of the parent, an error is thrown.
- If the first parameter is greater than or equal to the second parameter, an error is thrown.

For example, imagine that we want to create with the parent of America a new sales area called Spain. We want the node value to be between Canada and USA. We could achieve this with the query in Listing 12-13.

Tip Obviously, Spain should be included under Western Europe, not under America. Don't worry, this is a deliberate error, which we will resolve in the "Using `GetReparentedValue`" section of this chapter.

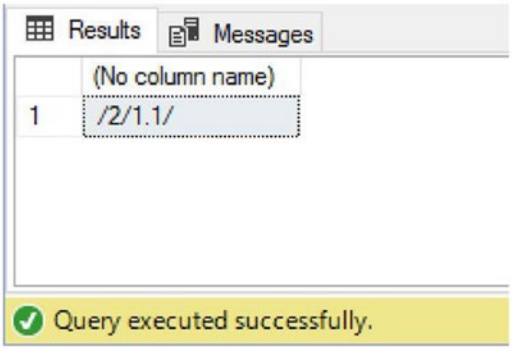
Listing 12-13. Generating a New Hierarchy Node

```
USE WideWorldImporters
GO

SELECT NewNode.ToString()
FROM
```

```
(
SELECT
        SalesAreaHierarchy.GetDescendant(0x6AC0,0x6B40) AS
NewNode
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaName = 'America'
) NewNode ;
```

The results produced by this query are illustrated in Figure 12-11. You will see that the lowest level contains a period (`1.1`). This is because `Canada` has a value of `1` and `USA` has a value of `2`. Therefore, to generate a node value between the two, an integer cannot be used. This guarantees that a new node can always be inserted between two existing nodes.



```
   (No column name)
1  /2/1.1/
```

Query executed successfully.

Figure 12-11. *Results of using `GetDescendant()`*

To programmatically add the Spain sales area to the hierarchy, we could use the query in Listing 12-14.

Listing 12-14. Insert a New Node into the Hierarchy

```
USE WideWorldImporters
GO

INSERT INTO Sales.SalesAreaHierarchyID
        (
                  SalesAreaID
                , SalesAreaHierarchy
                , SalesAreaName
                , CountOfSalesPeople
                , SalesYTD
                )
SELECT
        (SELECT MAX(SalesAreaID) + 1 FROM Sales.SalesArea
        HierarchyID)
              ,  SalesAreaHierarchy.GetDescendant
              (0x6AC0,0x6B40)
        , 'Spain'
        , 2
        , 200000
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaName = 'America' ;
```

Using GetReparentedValue( )

As you probably noticed in the "Using `GetDescendant()`" section, the Spain sales area was incorrectly created under the America aggregation area, as opposed to the Western Europe aggregation area. We can resolve this issue by using the `GetReparentedValue()` method.

Consider the script in Listing 12-15. First, we declare two variables, with the type `HierarchyID`. These will be passed as parameters into the `GetReparentedValue()` method. The `@America` variable is populated with the sales area hierarchy node pertaining to the original parent sales area, and the `@WesternEurope` variable is populated with the sales area hierarchy node pertaining to the target parent sales area.

Listing 12-15. Use `GetReparentedValue()`

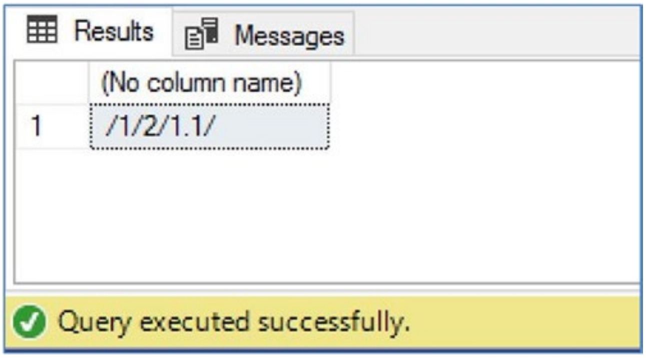
```
USE WideWorldImporters
GO

DECLARE @America HIERARCHYID =
(
    SELECT SalesAreaHierarchy
    FROM Sales.SalesAreaHierarchyID
    WHERE SalesAreaName = 'America'
) ;

DECLARE @WesternEurope HIERARCHYID =
(
    SELECT SalesAreaHierarchy
    FROM Sales.SalesAreaHierarchyID
    WHERE SalesAreaName = 'Western Europe'
) ;

SELECT Area.ToString()
FROM
(
SELECT SalesAreaHierarchy.GetReparentedValue(@America,
@WesternEurope) AS Area
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy = 0x6B16
) NewNodePath ;
```

The results of this script can be seen in Figure 12-12. You will notice that the leaf node value has remained the same, while the path (parent nodes) have been changed, so that the node sits under the Western Europe aggregation area.



```
   (No column name)
1  /1/2/1.1/
```

Query executed successfully.

Figure 12-12. *Results of using* `GetReparentedValue()`

We could update the ancestry of the Spain sales area, in the Sales. SalesAreaHierarchyID table, by using the script in Listing 12-16.

Listing 12-16. Updating the Ancestry of the Spain Sales Area

```
USE WideWorldImporters
GO

DECLARE @America HIERARCHYID =
(
    SELECT SalesAreaHierarchy
    FROM Sales.SalesAreaHierarchyID
    WHERE SalesAreaName = 'America'
) ;

DECLARE @WesternEurope HIERARCHYID =
(
    SELECT SalesAreaHierarchy
```

```
        FROM Sales.SalesAreaHierarchyID
        WHERE SalesAreaName = 'Western Europe'
) ;

UPDATE Sales.SalesAreaHierarchyID
SET SalesAreaHierarchy =
(
SELECT SalesAreaHierarchy.GetReparentedValue(@America,
@WesternEurope)
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy = 0x6B16
)
WHERE SalesAreaHierarchy = 0x6B16 ;
```

Using GetAncestor( )

The `GetAncestor()` method can be used to return the ancestor node of a given hierarchical node at the number of levels based on an input parameter to the method. For example, consider the query in Listing 12-17. This query will return the grandparent of Spain, the parent of Spain, and Spain itself, by passing different parameters into the `GetAncestor()` method.

Caution For this query to work as expected, you first must have run the previous examples in the chapter. Specifically, the insert and update queries in the “Using `GetDescendant()`” and “Using `GetReparentedValue()`” sections, as well as Listing 12-2, which creates and populates the table.

Listing 12-17. Using `GetAncestor()`

```
USE WideWorldImporters
GO

SELECT
                CurrentNode.SalesAreaName AS SalesArea
        , ParentNode.SalesAreaName AS ParentSalesArea
        , GrandParentNode.SalesAreaName AS GrandParentSalesArea
FROM Sales.SalesAreaHierarchyID Base
INNER JOIN Sales.SalesAreaHierarchyID CurrentNode
        ON CurrentNode.SalesAreaHierarchy = Base.
        SalesAreaHierarchy.GetAncestor(0)
INNER JOIN Sales.SalesAreaHierarchyID ParentNode
        ON ParentNode.SalesAreaHierarchy = Base.
        SalesAreaHierarchy.GetAncestor(1)
INNER JOIN Sales.SalesAreaHierarchyID GrandParentNode
        ON GrandParentNode.SalesAreaHierarchy = Base.
        SalesAreaHierarchy.GetAncestor(2)
WHERE Base.SalesAreaName = 'Spain' ;
```

Using IsDescendantOf( )

The `IsDescendantOf()` method evaluates if a node within the hierarchy is a descendant (at any level) of a node that is passed to it as a parameter. It is this method that we can use to rewrite the query in Listing 12-5, which rolled up the SalesYTD for all sales areas under the America aggregation area, using a traditional hierarchy.

You will remember, that when using a traditional hierarchy, we had to implement a recursive CTE, which rolled up the SalesYTD column, for all hierarchical levels, which are descendants of America. When using a `HierarchyID` column to maintain the hierarchy, however, our code is greatly simplified, as demonstrated in Listing 12-18.

Listing 12-18. Using `IsDescendantOf()`

```
USE WideWorldImporters
GO

SELECT
    SUM(SalesYTD) AS TotalSalesYTD
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy.IsDescendantOf(0x68) = 1 ;
```

Instead of a recursive CTE, the functionally equivalent code is a simple query with a `WHERE` clause that filters hierarchical nodes, based on whether they are descendants of the America aggregation area. Listing 12-19 shows a more complex example. Here, we are parameterizing the sales area and calculating not only the total SalesYTD but also the TotalSalesPeople and the regions AverageSalesPerSalesPerson.

Listing 12-19. Parameterizing `IsDescendantOf()` Queries

```
USE WideWorldImporters
GO

DECLARE @Region NVARCHAR(20) = 'America' ;

DECLARE @RegionHierarchy HIERARCHYID =
    (
         SELECT SalesAreaHierarchy
         FROM Sales.SalesAreaHierarchyID
         WHERE SalesAreaName = @Region
        ) ;

SELECT
      SUM(SalesYTD) AS TotalSalesYTD
    , SUM(CountOfSalesPeople) AS TotalSalesPeople
    , SUM(SalesYTD) / SUM(CountOfSalesPeople) AS
    AverageSalesPerSalesPerson
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy.IsDescendantOf(@RegionHierarchy) = 1 ;
```

You can clearly see how the Region parameter could be passed into a stored procedure, so that this code could be accessed by an application.

Let us now add a SalesAreaName to the SELECT list, and group by this column, as demonstrated in Listing 12-20.

Listing 12-20. Adding a GROUP BY

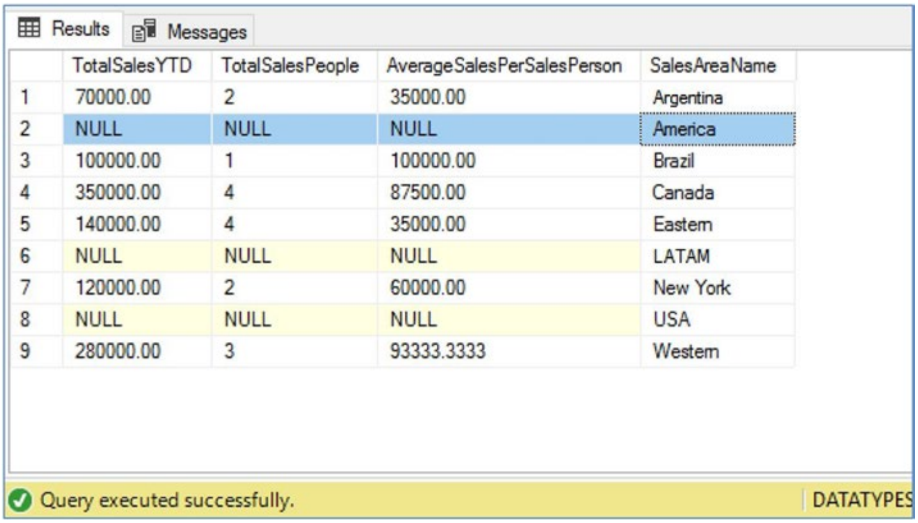
```
USE WideWorldImporters
GO

DECLARE @Region NVARCHAR(20) = 'America' ;

DECLARE @RegionHierarchy HIERARCHYID =
    (
         SELECT SalesAreaHierarchy
         FROM Sales.SalesAreaHierarchyID
         WHERE SalesAreaName = @Region
        ) ;

SELECT
      SUM(SalesYTD) AS TotalSalesYTD
    , SUM(CountOfSalesPeople) AS TotalSalesPeople
    , SUM(SalesYTD) / SUM(CountOfSalesPeople) AS
    AverageSalesPerSalesPerson
    , SalesAreaName
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy.IsDescendantOf(@RegionHierarchy) = 1
GROUP BY SalesAreaName ;
```

The results of this query are illustrated in Figure 12-13.



	TotalSalesYTD	TotalSalesPeople	AverageSalesPerSalesPerson	SalesAreaName
1	70000.00	2	35000.00	Argentina
2	NULL	NULL	NULL	America
3	100000.00	1	100000.00	Brazil
4	350000.00	4	87500.00	Canada
5	140000.00	4	35000.00	Eastern
6	NULL	NULL	NULL	LATAM
7	120000.00	2	60000.00	New York
8	NULL	NULL	NULL	USA
9	280000.00	3	93333.3333	Western

Query executed successfully. DATATYPES

Figure 12-13. *Results of using* `IsDescendantOf()` *with* `GROUP BY`

The interesting behavior exposed by the results of this query is that America is included. This is because `HierarchyID` regards America as a descendant of itself. This does not create an issue for us, because the aggregations are not pre-calculated. However, in some instances, you may have to exclude America from the result set. This can easily be achieved by adding an additional filter to the `WHERE` clause, as demonstrated in Listing 12-21.

Listing 12-21. Filtering the Current Node from Descendants

```
USE WideWorldImporters
GO

DECLARE @Region NVARCHAR(20) = 'America' ;

DECLARE @RegionHierarchy HIERARCHYID =
    (
         SELECT SalesAreaHierarchy
         FROM Sales.SalesAreaHierarchyID
         WHERE SalesAreaName = @Region
        ) ;

SELECT
      SUM(SalesYTD) AS TotalSalesYTD
    , SUM(CountOfSalesPeople) AS TotalSalesPeople
    , SUM(SalesYTD) / SUM(CountOfSalesPeople) AS
    AverageSalesPerSalesPerson
    , SalesAreaName
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy.IsDescendantOf(@RegionHierarchy) = 1
        AND SalesAreaHierarchy <> @RegionHierarchy
GROUP BY SalesAreaName ;
```

This query filters the result set to exclude the sales area, the hierarchical node of which is equal to the hierarchical node that is being passed to the `IsDescendantOf()` method. This technique allows us to use the `WITH ROLLUP` clause on the `GROUP BY`, in conjunction with wrapping `SalesAreaName` in an `ISNULL()` function, to produce a subtotal row for the whole of America. This is demonstrated in Listing 12-22.

Listing 12-22. Producing a Total Row for America

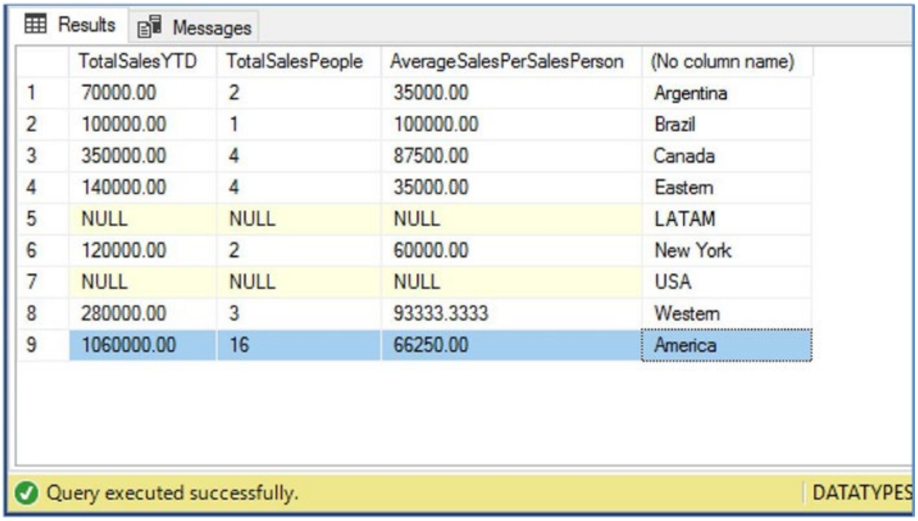
```
USE WideWorldImporters
GO

DECLARE @Region NVARCHAR(20) = 'America' ;

DECLARE @RegionHierarchy HIERARCHYID =
    (
         SELECT SalesAreaHierarchy
         FROM Sales.SalesAreaHierarchyID
         WHERE SalesAreaName = @Region
        ) ;

SELECT
      SUM(SalesYTD) AS TotalSalesYTD
    , SUM(CountOfSalesPeople) AS TotalSalesPeople
    , SUM(SalesYTD) / SUM(CountOfSalesPeople) AS
    AverageSalesPerSalesPerson
    , ISNULL(SalesAreaName, @Region)
FROM Sales.SalesAreaHierarchyID
WHERE SalesAreaHierarchy.IsDescendantOf(@RegionHierarchy) = 1
        AND SalesAreaHierarchy <> @RegionHierarchy
GROUP BY SalesAreaName  WITH ROLLUP ;
```

The results of this query can be seen in Figure 12-14.



	TotalSalesYTD	TotalSalesPeople	AverageSalesPerSalesPerson	(No column name)
1	70000.00	2	35000.00	Argentina
2	100000.00	1	100000.00	Brazil
3	350000.00	4	87500.00	Canada
4	140000.00	4	35000.00	Eastern
5	NULL	NULL	NULL	LATAM
6	120000.00	2	60000.00	New York
7	NULL	NULL	NULL	USA
8	280000.00	3	93333.3333	Western
9	1060000.00	16	66250.00	America

Query executed successfully. DATATYPES

Figure 12-14. *Results of adding a total row*

Indexing HierarchyID Columns

There are no “special” index types that support `HierarchyID`, as there are for `XML` or geospatial data types. Instead, the performance of `HierarchyID` columns can be improved by using traditional clustered and nonclustered indexes. When creating indexes to support `HierarchyID`, there are two strategies that can be employed, depending on the nature of the queries that will use the indexes.

By default, creating an index on a `HierarchyID` column will create a depth-first index. This means that descendants will be stored close to their parents. In our example, New York would be stored close to Eastern, which in turn would be stored close to USA, and so on. The script in Listing 12-23 demonstrates how to create a clustered index on the SalesAreaHierarchy column, which uses a depth-first approach.

Caution The script first drops the primary key on the SalesAreaID column, which implicitly drops the clustered index on this column. The primary key name, in this case, is system-generated, however. Therefore, to run this script, you must change the name of the constraint, to reflect your own system.

Listing 12-23. Creating a Depth-First Clustered Index

```
USE WideWorldImporters
GO

ALTER TABLE Sales.SalesAreaHierarchyID
        DROP CONSTRAINT PK__SalesAre__DB0A1ED5D7B258FB ;
GO

CREATE CLUSTERED INDEX SalesAreaHierarchyDepthFirst
        ON Sales.SalesAreaHierarchyID(SalesAreaHierarchy) ;
GO
```

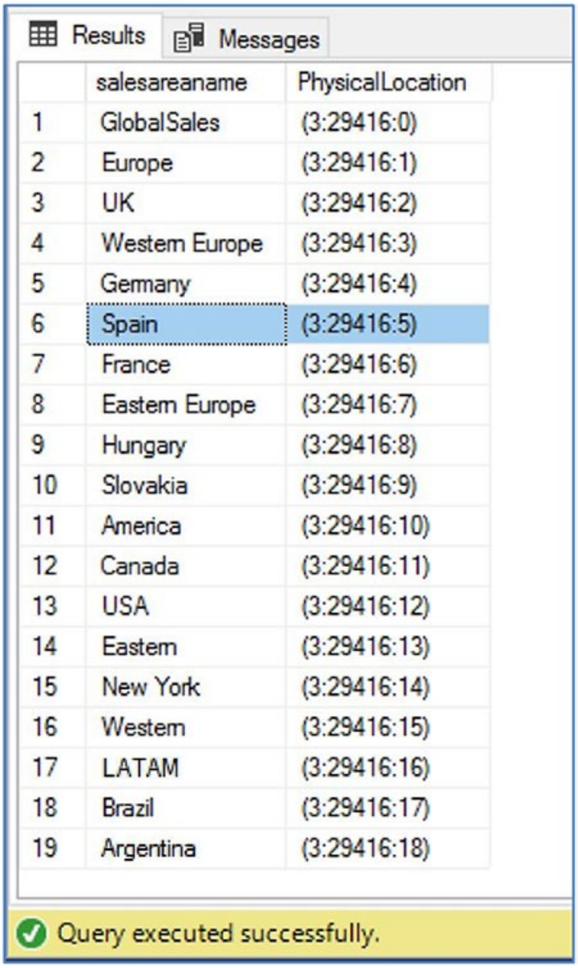
We can see how SQL Server has organized this data, by running the query in Listing 12-24. This query uses the undocumented `sys.physlocformatter()` function to return the exact location of each record, in the format `FileID:PageID:SlotID`.

Listing 12-24. View Location of Rows

```
USE WideWorldImporters
GO

SELECT
      SalesAreaName
    , sys.fn_PhysLocFormatter(%%physloc%%) AS PhysicalLocation
FROM Sales.SalesAreaHierarchyID ;
```

This query returns the results shown in Figure 12-15. You can see that each node is stored under its parent. This is even true for Spain, despite us adding it after the other regions, which means its original location would have been the final used slot in the page.



	salesareaname	PhysicalLocation
1	GlobalSales	(3:29416:0)
2	Europe	(3:29416:1)
3	UK	(3:29416:2)
4	Westem Europe	(3:29416:3)
5	Germany	(3:29416:4)
6	Spain	(3:29416:5)
7	France	(3:29416:6)
8	Eastem Europe	(3:29416:7)
9	Hungary	(3:29416:8)
10	Slovakia	(3:29416:9)
11	America	(3:29416:10)
12	Canada	(3:29416:11)
13	USA	(3:29416:12)
14	Eastem	(3:29416:13)
15	New York	(3:29416:14)
16	Westem	(3:29416:15)
17	LATAM	(3:29416:16)
18	Brazil	(3:29416:17)
19	Argentina	(3:29416:18)

Query executed successfully.

Figure 12-15. *Results of viewing row locations*

Tip The row's physical location is likely to be different when you run the query yourself.

The other possible indexing strategy is a breadth-first technique. Here, sibling nodes will be stored close to each other, instead of storing descendants close to each other. To implement a breadth-first indexing strategy, we must add to our table an additional column that stores the hierarchical level of each node. This column can be created and populated by using the script in Listing 12-25.

Listing 12-25. Adding a Level Column to Support Breadth-First Indexing

```
USE WideWorldImporters
GO

ALTER TABLE Sales.SalesAreaHierarchyID ADD
        SalesAreaLevel INT NULL ;
GO

UPDATE Sales.SalesAreaHierarchyID
SET SalesAreaLevel = SalesAreaHierarchy.GetLevel() ;
```

Using the script in Listing 12-26, we can now create a clustered index, which is first order by the hierarchical level of the node and then by the `HierarchyID` column. This will cause siblings to be stored close to one another.

Note The script first drops the existing clustered index, because a table can only support a single clustered index.

Listing 12-26. Creating a Breadth-First Index

```
USE WideWorldImporters
GO

DROP INDEX SalesAreaHierarchyDepthFirst ON Sales.
SalesAreaHierarchyID ;
GO

CREATE CLUSTERED INDEX SalesAreaHierarchyBredthFirst
        ON Sales.SalesAreaHierarchyID(SalesAreaLevel,
        SalesAreaHierarchy) ;
```

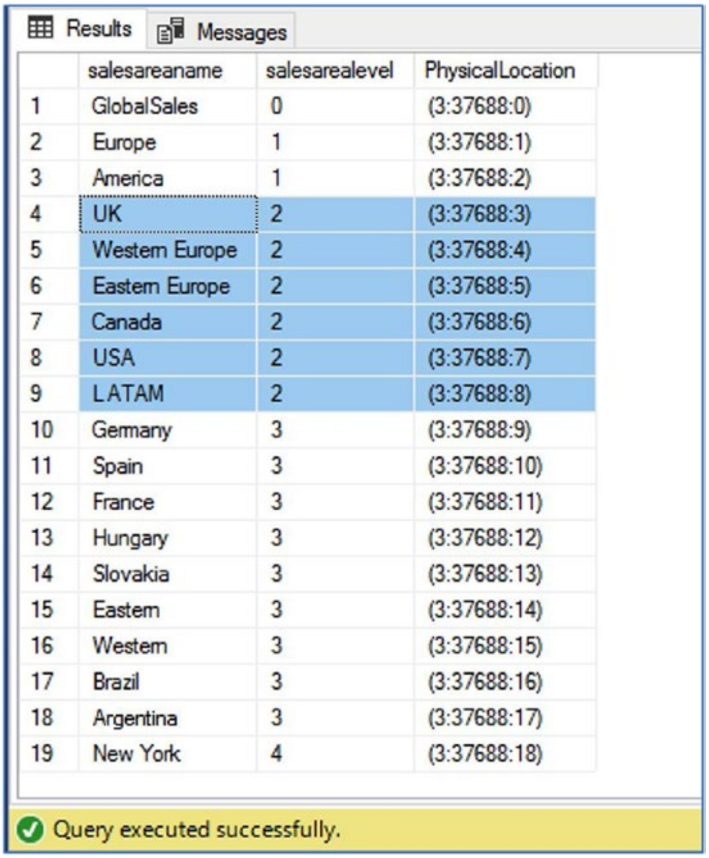
Listing 12-27 demonstrates how we can use the same technique as in Listing 12-24, to view the actual location of each node within the hierarchy. This time, as well as returning the sales area name, we will also return the SalesAreaLevel for easy analysis.

Listing 12-27. View Rows Location with a Breadth-First Strategy

```
USE WideWorldImporters
GO

SELECT
      SalesAreaName
    , SalesAreaLevel
    , sys.fn_PhysLocFormatter(%%physloc%%) AS PhysicalLocation
FROM Sales.SalesAreaHierarchyID ;
```

The results of this query can be found in Figure 12-16. You will notice that the order of rows has changed and that UK, Western Europe, Eastern Europe, Canada, USA, and LATAM are now next to one another, as they are all at Level 2 of the hierarchy.



	salesareaname	salesarealevel	PhysicalLocation
1	GlobalSales	0	(3:37688:0)
2	Europe	1	(3:37688:1)
3	America	1	(3:37688:2)
4	UK	2	(3:37688:3)
5	Western Europe	2	(3:37688:4)
6	Eastern Europe	2	(3:37688:5)
7	Canada	2	(3:37688:6)
8	USA	2	(3:37688:7)
9	LATAM	2	(3:37688:8)
10	Germany	3	(3:37688:9)
11	Spain	3	(3:37688:10)
12	France	3	(3:37688:11)
13	Hungary	3	(3:37688:12)
14	Slovakia	3	(3:37688:13)
15	Eastern	3	(3:37688:14)
16	Western	3	(3:37688:15)
17	Brazil	3	(3:37688:16)
18	Argentina	3	(3:37688:17)
19	New York	4	(3:37688:18)

Query executed successfully.

Figure 12-16. *Results of viewing row locations in a breadth-first hierarchy*

Summary

`HierarchyID` is created as a .NET class and implemented as a data type in SQL Server. Using `HierarchyID` over a traditional approach to modeling hierarchies in SQL Server has the benefits both of reducing code complexity and improving performance. The `HierarchyID` data type exposes several methods that can be used by developers to easily navigate a hierarchy, insert new hierarchical nodes, or update existing nodes so that they sit under a new parent.

The two most commonly used methods, in my experience, are the `ToString()` method, which allows a developer to format a hierarchical node as a human-readable string representation, and `IsDescendantOf()`, which performs an evaluation of hierarchical node lineage and returns 1 when a node is a descendant of an input parameter and 0 if it is not.

The `Read()` and `Write()` methods offer data type conversion functionality to SQLCLR, but these are not implemented in T-SQL, as the `CAST` and `CONVERT` functions can easily be used instead.

When indexing `HierarchyID` columns, either a depth-first strategy or a breadth-first strategy can be applied. Depth-first is the default option and stores child nodes close to their parents. A breadth-first strategy requires an additional column in the table, which stores the node's level with the hierarchy. This allows a multicolumn index to store sibling nodes close to one another. The indexing option that you choose should reflect the nature of the queries run against the `HierarchyID` column.