http://www.sqlservercentral.com/articles/SQL+Server+2008/62204/
Printed 2008/03/26 06:13PM

The HierarchyID Datatype in SQL Server 2008

By J P R, 2008/02/08

Introduction

In this article, I will present a new feature of SQL Server 2008: the HierarchyID data-type. We will see that this new data-type provides a new way to design trees in databases. It adds functionalities to T-SQL language and improves whole performance.

That article explains deeply the new type and gives you some samples, with a comparison with classic way, based on CTE.

Source code is available here: code source

Note: SQL scripts work with SQL Server 2008 CTP2 (july)

The Issue Description

Manage a hierarchy is a common issue for information systems. So classical that we can find many case studies on it, for example, the famous Employee table. Other examples can be pointed out as managing a tree of categories in a catalog or modeling a file system.

Employee hierarchy issue is simple, we have to store employees list and each employee knows its manager. We represent it with the following design:

Figure 1 – Classical Employee table

For object-oriented design gurus, that design is similar to Composite pattern (from Gang of Four). Even if the diagram is simple, query it can be a hard job. Here are some samples queries:

· Find all employees below another

· Know the level of an employee in the company

· etc.

SQL Server 2000 users used mechanics based on CURSOR or temporary tables to write these queries. But simplicity, maintenance or performances were sacrificed.

SQL Server 2005 brings up a smart way in T-SQL language with Common Table Expressions (CTE). CTE allow you to script recursive queries. Without writing a chapter on CTE syntax, here is a sample of CTE computing level in hierarchy for Employee table:

```
WITH UpperHierarchy(EmployeeId, LastName, Manager, HierarchyOrder)
AS
(
   SELECT emp.EmployeeId, emp.LoginId, emp.LoginId, 1 AS HierarchyOrder
   FROM HumanResources.Employee AS emp
```

```
      WHERE emp.ManagerId isNull
    UNION ALL
    SELECT emp.EmployeeId, emp.LoginId, Parent.LastName, HierarchyOrder + 1
    FROM HumanResources.Employee AS emp
           INNER JOIN UpperHierarchy AS Parent
                 ON emp.ManagerId = parent.EmployeeId
)
SELECT *
From UpperHierarchy
```

You can run that query against AdventureWorks database.

```
Select EmployeeId, LoginId, ManagerId, Title
From HumanResources.Employee
```

Results | Messages

	EmployeeId	LoginId	ManagerId	Title
1	1	adventure-works\guy1	16	Production Technician - WC60
2	2	adventure-works\kevin0	6	Marketing Assistant
3	3	adventure-works\roberto0	12	Engineering Manager
4	4	adventure-works\rob0	3	Senior Tool Designer
5	5	adventure-works\thierry0	263	Tool Designer
6	6	adventure-works\david0	109	Marketing Manager
7	7	adventure-works\jolynn0	21	Production Supervisor - WC60
8	8	adventure-works\ruth0	185	Production Technician - WC10
9	9	adventure-works\gail0	3	Design Engineer
10	10	adventure-works\barry0	185	Production Technician - WC10
11	11	adventure-works\jossef0	3	Design Engineer
12	12	adventure-works\terri0	109	Vice President of Engineering
13	13	adventure-works\sidney0	185	Production Technician - WC10
14	14	adventure-works\taylor0	21	Production Supervisor - WC50
15	15	adventure-works\jeffrey0	185	Production Technician - WC10

Figure 2 - Table Employee de la base AdventureWorks

CTE was a great evolution for tables representing trees but some problems remain. By the way, query complexity drops down but what about performance? Even if CTE execution plans are optimized, you'll fall out in a case where no index is possible.

The HierarchyID

To provide a real support of hierarchies, SQL Server 2008 introduces a new type of data: HierarchyID.

It is a managed type (.NET), handled by the SQLCLR of SQL Server.

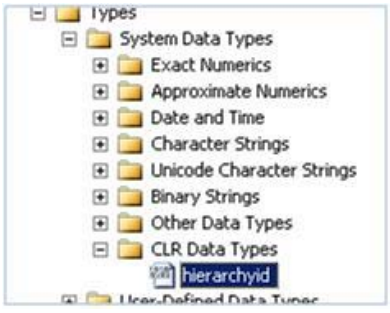

Figure 3 - System Data Types

It does not store the identifier of the parent element but a set of information to locate the element in the hierarchy. This type represents a node in the tree structure.

If you look at values contained in a column of HierachyId type, you realize that they are binary values.

```
Select EmployeeId, EmployeeName From dbo.Organization
```

Results | Messages

	EmployeeId	EmployeeName
1	0x	djeepy1
2	0x	adventure-works\ken0
3	0x58	adventure-works\david0
4	0x58	adventure-works\terri0
5	0x58	adventure-works\jean0
6	0x58	adventure-works\laura1
7	0x58	adventure-works\james1
8	0x58	adventure-works\brian3
9	0x5AC0	adventure-works\stephen0
10	0x5AC0	adventure-works\amy0
11	0x5AC0	adventure-works\sued0

Figure 4 – HierarchyID Column values

We can represent the HierarchyID type in a string format. This format shows clearly information carried by this type. Indeed, string representation is formatted as is:

/<index level 1>/<index level 2>/…/<index level N>

This representation corresponds to the tree structure as shown in diagram below. Note that first child of a node does not have a value of 1 all the time but can have the /1.2/ value for example.

```
Select EmployeeId.ToString(), EmployeeName From dbo.Organization
```

Results | Messages

	(No column name)	EmployeeName
1	/	djeepy1
2	/	adventure-works\ken0
3	/1/	adventure-works\david0
4	/1/	adventure-works\terri0
5	/1/	adventure-works\jean0
6	/1/	adventure-works\laura1
7	/1/	adventure-works\james1
8	/1/	adventure-works\brian3
9	/1/1/	adventure-works\stephen0
10	/1/1/	adventure-works\amy0

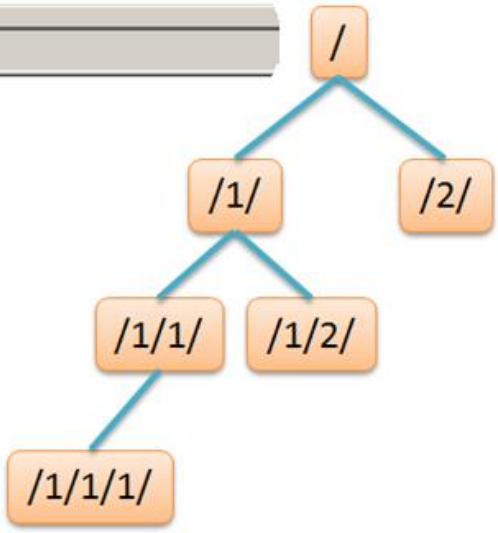

Figure 5 – String representation of HierarchyID values

You noticed that query used to display string representation uses the function ToString() directly on the column. HierarchyId can be manipulated through a set of functions described later.

Usage

HierarchyID is used like any other type in the DDL of table creation. In the examples below, we will work on a table called Organization. At this point, it contains only one column of the type HierarchyID and the name of the corresponding employee.

```
CREATE TABLE Organization
 (
      EmployeeID hierarchyid NOT NULL,
 EmployeeName nvarchar(50)NOT NULL
 )
```

We populate Organization table with data from Employee table of AdventureWorks database. We are going to use CTE

described above. To determine the value of the root node, we will use the GetRoot() function of HierarchyID type (notice that GetRoot, like other functions of HierarchyID type, is case-sensitive):

hierarchyid::GetRoot()

To determine the value of child nodes, on each recursion we use GetDescendant function on the parent node:

Parent.Node.GetDescendant(null,null)

Parameters of this function make it possible to place the node child at a certain position among the other nodes (siblings).

The modified CTE gives the following T-SQL script. It copies the hierarchy of the Employee table to the new Organization table using the new type.

```
WITH UpperHierarchy(EmpId, LastName, Node)
AS
(
   SELECT EmployeeId, LoginId,hierarchyid::GetRoot()
   FROM HumanResources.Employee
     WHERE ManagerId is Null
   UNION ALL
   SELECT Sub.EmployeeId, Sub.LoginId, Parent.Node.GetDescendant(null, null)
   FROM HumanResources.Employee AS Sub
          INNER JOIN UpperHierarchy AS Parent
                ON Sub.ManagerId = Parent.EmpId
)
Insert Into dbo.Organization(EmployeeId, EmployeeName)
Select Node, LastName
From UpperHierarchy
```

HierarchyId characteristics

Thanks to its binary format, the HierarchyId type has a variable size which makes it extremely compact compared to information it carries. As example, for a hierarchy of 100.000 people, the size will be of 40 bits, which is just 25% more than an integer. Of course, according to the way in which the hierarchy is filled (an average number children by node, density), space used will be more or less important.

The type supports comparison and it is important to understand the way in which the tree structure is traversed to know how the elements are compared. The comparison take place initially on the depth in the tree (depth-first) then over the sibling nodes as described in the diagram below.

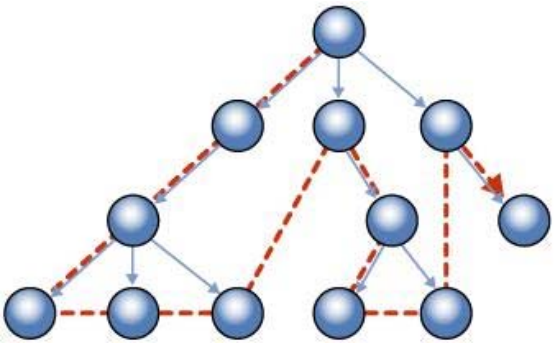

Figure 6 - Depth-first (SQL Server 2008 CTP 2 Books On Line)

We will see that we can index the table to allow traversing tree breadth-first.

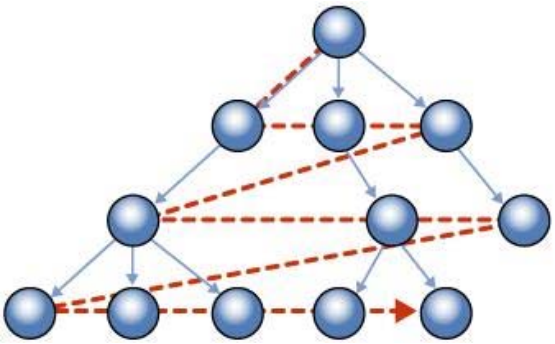

Figure 7 - Breadth-First (SQL Server 2008 CTP 2 Books On Line)

To set up that kind of indexing, we need to know the level of each record in table. We can get this information directly from the HierarchyID column with GetLevel() function. So we add a computed column in our table to provide employee level.

```
Alter Table dbo.Organization
Add HierarchyLevel As EmployeeID.GetLevel()
```

Once the new column is created, we can index it and so get a breadth-first way to traverse the hierarchy.

```
CREATE INDEX IX_ParLargeur
ON Organization(HierarchyLevel,EmployeeID);
```

Limitations

Uniqueness

Uniqueness is not supported natively by the HierarchyId type. For example, it possible to have two roots in the same table. You will obviously run into integrity problems in your application but also, it will be impossible to uniquely index the nodes and thus the whole tree in clustered way.

To work arround that limitation, we can add a primary key (or a unique index) on HierarchyId column:

```
ALTER TABLE dbo.Organization ADD CONSTRAINT
      PK_Organization PRIMARY KEY
      (
      EmployeeID
      )
```

Primary key or unique index on the HierarchyId allows a depth-first indexing of the table.

With the data populated previously, this DDL will raise an error. Indeed, the children of each node have the same index, which does not allow uniqueness. To fix that problem, we need to reorganize the tree by ordering children at each level. To do it, it is necessary to pass parameters to GetDescendant() function. This operation is explained later in the article.

Foreign Keys

Contrary to traditional modeling described above, the foreign key referencing the parent record is not supported natively. Indeed, the HierarchyId type stores the path of the node in the tree, not the parent node.

Nevertheless, it is possible to retrieve easily the identifier of the parent with the GetAncestor() function, as shown in the example below:

```
Select EmployeeId.GetAncestor(1), EmployeeName
From dbo.Organization
```

GetAncestor() returns a HierarchyID. If the HierarchyID column is the primary key of the table (as in our sample), it is then possible to add a foreign key referencing itself.

```
Alter Tabledbo.Organization
Add ParentId AS EmployeeId.GetAncestor(1)PERSISTED
     REFERENCES dbo.Organization(EmployeeId)
```

Now, our table has the same integrity rules as initial modeling.

HierarchyID functions

HierarchyID data type can be manipulated through a set of functions.

· GetAncestor

· GetDescendant

· GetLevel

· GetRoot

· ToString

· IsDescendant

· Parse

· Read

· Reparent

· Write

We saw the first 5 in previous examples. The next 5 are explained in the table below:

Function	Description
IsDescendant	Allow to know if a record is a child of another in the hierarchy
Parse	It is the opposite function of ToString(); it makes it possible to obtain a HierarchyID value from a string
Read	Same as Parse but for varbinary values
Write	Same as ToString but for varbinary values
Reparent	Allows to move a node in the hierarchy modifying its parent

Warning, all functions are case sensitive.

Insert nodes in the hierarchy

Since the HierarchyID type is more complex than a simple reference towards the parent record, it is thus more complicated to determine its value when inserting new elements. GetDescendant() function give us valid node values. However, in our example, the HierarchyID column has a unique constraint which does not make it possible to use GetDescendant as we did. By the way, we have to provide additional information: the index of the node in the list of children.

To do this, we pass sibling nodes as parameters to GetDescendant function. Of course we can pass NULL values to place the node in first or last position.

In the example below, we insert a node as the last child of an existing node. Before, there is a step where we retrieve the sibling node.

```
--finding sibling node
SELECT @sibling = Max(EmployeeID)
FROM dbo.Organization
WHERE EmployeeId.GetAncestor(1)= @Parent;
--inserting node
INSERT dbo.Organization(EmployeeId, EmployeeName)
VALUES(@Parent.GetDescendant(@sibling,NULL), @Name)
```

We do not always want to (or can) recover the sibling node to perform insertion. There is perhaps an implied policy to determine node position.

For example, let’s say we have an [order] column which position nodes among its siblings. We can compute node path as string:

```
Declare @Parent As HierarchyID = HierarchyID::GetRoot()
```

```
Declare @NewPath As varchar(10)= @Parent.ToString()+ CAST([Order] AS varchar(3))+ '/'
```

In this example, since the node @Parent is the root, that will give/<order>/. Thanks to the Parse() function, we can use this value to create the new node.

```
INSERT dbo.Organization(EmployeeId, EmployeeName)
VALUES(HierarchyId::Parse(@NewPath),'aChild')
```

You will have note the new syntax of SQL Server 2008 to declare and assign variables in only one line.

Querying

Find Referees

CTE is now useless. To retrieve a whole branch of the hierarchy, the query is simple:

```
Select *
From dbo.Organization
Where @BossNode.IsDescendant(EmployeeId)
```

Warning : a node is included in its descendants so @BossNode is its own descendant.

Find boss of somebody

To go up the chain of managers, the above query is reversed:

```
Select *
From dbo.Organization
Where EmployeeId.IsDescendant(@BossNode)
```

Find all employees of a given level

No more need to store a column with computed level:

```
Select *
From dbo.Organization
Where EmployeeId.GetLevel() = 3
```

Performance

We will compare performances of the new type compared to the ones of CTE. For this comparison, we will take as example the request recovering all referee of the manager 'adventure-works \ james1'. The tables are those which illustrated that article: **Employee** (traditional model) and **Organization** (HierarchyID).

CTE script is:

```
WITH UpperHierarchy(EmployeeId, LastName, Manager, HierarchyOrder)
AS
(
   SELECT emp.EmployeeId, emp.LoginId, emp.LoginId, 1 AS HierarchyOrder
   FROM HumanResources.Employee AS emp
     WHERE emp.LoginId ='adventure-works\james1'
   UNION ALL
   SELECT emp.EmployeeId, emp.LoginId, Parent.LastName, HierarchyOrder + 1
   FROM HumanResources.Employee AS emp
          INNER JOIN UpperHierarchy AS Parent
                ON emp.ManagerId = parent.EmployeeId
)
SELECT EmployeeId, LastName
From UpperHierarchy
```

The script using HierarchyID column is presented below. You will notice that there are 2 steps : one to retreive parent node, second to find referees.

```
Declare @BossNode As HierarchyId
Select @BossNode = EmployeeID From dbo.Organization Where EmployeeName = 'adventure-works\james1'
Select *
From dbo.Organization
Where @BossNode.IsDescendant(EmployeeId)= 1
```

Execution Plan, available through Management Studio, gives us information about performances.

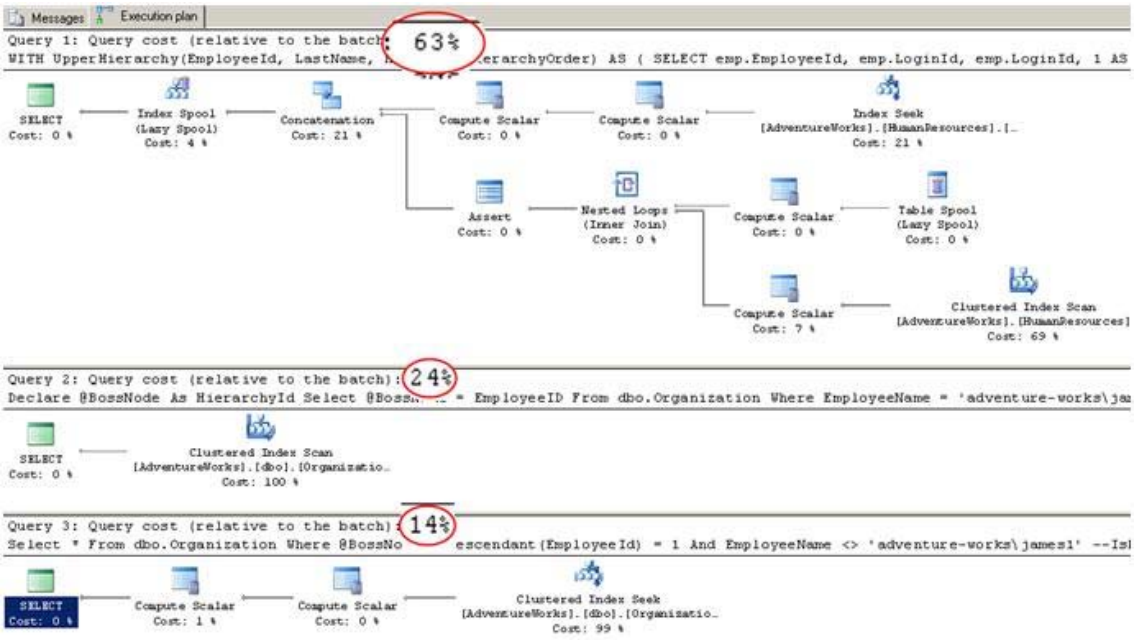

Figure 8 – Performances Benchmark

We can see that CTE takes 63% of the whole batch. That implies HierarchyID is 50% better

We can see that the step to retrieve parent node (james1) takes a major part of the query (with a Scan) because the column is not indexed. But, since it takes the same ratio in the 2 methods, we can ignore that point.

We can also see that the plan of execution of the CTE is much more complex than one of the HierarchyID type. This is because primary key allows a unique scan of the table.

If we look at systems resources used by these requests, the conclusions are catastrophic for the CTE. Here is a profiler trace showing multiple executions.

Untitled - 1 (LAPHROAIG)

EventClass	TextData	CPU	Reads	Writes	Duration
Trace Start					
SQL:BatchCompleted	--AVEC LA CTE---------- ...	131	3570	0	362
SQL:BatchCompleted	--AVEC LE CHAMP HIERARCHYID-- ...	30	30	0	174
SQL:BatchCompleted	--AVEC LA CTE---------- ...	90	3558	0	335
SQL:BatchCompleted	--AVEC LE CHAMP HIERARCHYID-- ...	0	12	0	74
SQL:BatchCompleted	--AVEC LA CTE---------- ...	60	3558	0	332
SQL:BatchCompleted	--AVEC LE CHAMP HIERARCHYID-- ...	10	12	0	111
SQL:BatchCompleted	--AVEC LA CTE---------- ...	90	3558	0	400
SQL:BatchCompleted	--AVEC LE CHAMP HIERARCHYID-- ...	10	12	0	117
Trace Stop					

Figure 9 – System ressources usage

We can see the 1/3 – 2/3 ration in duration column. However, ratio rises up to 300 for IO usage. CTE intensively uses temporary tables (Table Spool) which imply a lot of read. Also, CPU used is 9 times inferior.

HierarchyID wins by KO.

Conclusion

According performances, you can use the HierarchyID type without hesitation when modeling tree structures in a relational database. That new type fulfills its goal. It gives information which required complex requests (IsDescendant, GetDescendant, etc) and replaces the use of technical fields (GetLevel, GetAncestor, etc).

However, without being a “troublesome meddler”, I invite you to consider the use of this new type with wisdom (ie. only if it meets your needs). Indeed, the HierarchyID type introduces some disadvantages.

First of all, this type is more technical and thus more difficult to use and maintain. It remembers to me anonymous delegates in C# 2.0; everyone found that genius but a lot broke their teeth using it in real life. Also, HierarchyID can be inefficient; for example, insertions are more complex and need more CPU.

So, as every technology in IT, you should correctly evaluate your needs before choosing HierachyID. Here are some clues helping for your choice. Classic design would be prefer in those cases:

· If the size of key is big and you need to optimize storage; even if HierarchyID is a compact type, you will rapidly exceed 4bytes.

· If you query directly unique elements, primary key or unique index will be better.

· If you often move intermediate nodes ; hierarchy updates are slower with HierarchyID type

We made a complete overview of this new SQL Server type but some topics are missing, like use of HierarchyID in managed procedures or functions. Also, we could have dealt with use in an application using ADO.NET from Framework 3.5.
