One of the coolest features SQL Server 2008 has to offer is the new data type called hierarchyid **(or** SqlHierarchyId in the CLR**)**. As you can begin to imagine this data type addresses a common issue when dealing with hierarchical data. You’ve probably been confronted with a scenario in witch you needed to data tree of some sort, and you’ve probably struggled with joins, recursive operations, and other strategies in the past. With this new type you can implement tree structures with ease, making use of the best indexing strategy. This article aims to explain what this data type is, what it can do for you, its main advantages and a sample.

Hierarchyid is a variable length data type, and it represents a node in your data tree. By itself, it knows nothing of its siblings, so it’s up to the developer to provide such connection in order to create the necessary relationship between rows.

Memory-wise, it encodes information about a single node in the hierarchy tree by encoding the path from the root of the tree to the node. This path is nothing but a sequence of node labels of all children from the root to the node. The root itself is represented by a slash, so a path that aims to the root will only have “\”. From there on, slashes are used to separate levels. Each node label is encoded as a sequence of integers separated by dots. Consider this ordered example of hierarchyid’s:

\  (*root level*)

\1  (*depth 1 node, with label “1”*)

\0.5  (*depth 1 node, with label “0.5”*)

\0.5.1  (*depth 1 node, with label “0.5.1”*)

\0.6  (*depth 1 node, with label “0.6”*)

\1\1  (*depth 2 node, with label “1”*)

So, you’re probably wondering how do we know how to order nodes in the same level. This is accomplished by comparing node labels, like versioning. 0.5.1 comes after 0.5 and before 0.6. If we wished to insert a new node between 0.5 and 0.5.1, we could use 0.5.0.1 or 0.5.0.2, etc. Pretty simple actually.

Now, performance-wise, here’s what you can expect:

**Extremely compact:**

The average number of bits that are required to represent a node in a tree with *n* nodes depends on the average fanout (the average number of children of a node). For small fanouts (0-7), the size is about 6\*logA*n* bits, where A is the average fanout. A node in an organizational hierarchy of 100,000 people with an average fanout of 6 levels takes about 38 bits. This is rounded up to 40 bits, or 5 bytes, for storage.

**Comparison is in depth-first order**

Given two hierarchyid values a and b, a<b means a comes before b in a depth-first traversal of the tree. Indexes on hierarchyid data types are in depth-first order, and nodes close to each other in a depth-first traversal are stored near each other. For example, the children of a record are stored adjacent to that record. For more information, check out [Using hierarchyid Data Types (Database Engine)](http://technet.microsoft.com/en-us/library/bb677173.aspx).

**Support for arbitrary insertions and deletions**

By using the [GetDescendant](http://technet.microsoft.com/en-us/library/bb677209.aspx) method, it is always possible to generate a sibling to the right of any given node, to the left of any given node, or between any two siblings. The comparison property is maintained when an arbitrary number of nodes is inserted or deleted from the hierarchy. Most insertions and deletions preserve the compactness property. However, insertions between two nodes will produce hierarchyid values with a slightly less compact representation. The encoding used in the hierarchyid type is limited to 892 bytes. Consequently, nodes which have too many levels in their representation to fit into 892 bytes cannot be represented by the hierarchyid type.

SQL Server 2008 database engine has some new methods that go along with it. I’ll describe them a bit:

[GetAncestor(*n*)](http://technet.microsoft.com/en-us/library/bb677202.aspx) : Returns a hierarchyid representing the *n*th ancestor of the affected node.

[GetDescendant(*child1, child2*)](http://technet.microsoft.com/en-us/library/bb677209.aspx) : Return a child of the affected node, depending on child 1 and 2.

* If affected node is NULL, returns NULL.
* If affected node is not NULL, and both child1 and child2 are NULL, returns a child of the affected node.
* If affected node and child1 are not NULL, and child2 is NULL, returns a child of the affected node greater than child1.
* If affected node and child2 are not NULL and child1 is NULL, returns a child of the affected node less than child2.
* If affected node , child1, and child2 are not NULL, returns a child of the affected node greater than child1 and less than child2.
* If child1 is not NULL and not a child of the affected node, an exception is raised.
* If child2 is not NULL and not a child of the affected node, an exception is raised.
* If child1 >= child2, an exception is raised.

[GetLevel](http://technet.microsoft.com/en-us/library/bb677197.aspx) : Returns an integer that represents the depth of the affected node in the tree.

[GetRoot](http://technet.microsoft.com/en-us/library/bb677194.aspx) : Static method. Returns the root of the hierarchy tree.

[IsDescendantOf(*parent*)](http://technet.microsoft.com/en-us/library/bb677203.aspx) : Returns true if the affected node is a descendant of parent.

[Parse(*input*)](http://technet.microsoft.com/en-us/library/bb677201.aspx) : Static method. Converts the canonical string representation of a hierarchyid to a hierarchyid value. **Parse** is called implicitly when a conversion from a string type to hierarchyid occurs. Acts as the opposite of [ToString](http://technet.microsoft.com/en-us/library/bb677195.aspx).

[Read](http://technet.microsoft.com/en-us/library/bb677205.aspx) : It’s a CLR method. **It** reads binary representation of SqlHierarchyId from the passed-in BinaryReader and sets the SqlHierarchyId object to that value. **As it** cannot be called by using Transact-SQL, use CAST or CONVERT instead.

[GetReparentedValue(*oldRoot, newRoot*)](http://technet.microsoft.com/en-us/library/bb677207.aspx) : Returns a node whose path from the root is the path to *newRoot*, followed by the path from *oldRoot* to the affected node.

[ToString](http://technet.microsoft.com/en-us/library/bb677195.aspx) : Returns a string with the logical representation of the affected node. **ToString** is called implicitly when a conversion from hierarchyid to a string type occurs. Acts as the opposite of Parse.

[Write(BinaryWriter)](http://technet.microsoft.com/en-us/library/bb677196.aspx) : CLR method. Writes out a binary representation of SqlHierarchyId to the passed-in BinaryWriter. Write cannot be called by using Transact-SQL so use CAST or CONVERT instead.

One of the first challenges I faced on a personal project when working with hierarchyid is the actual lack of support in Entity Framework. As far as I know, Microsoft has no time frame for this yet (I’ll keep you posted though!).

The workaround I found to this was to have an extra column on my tree table to hold the node’s varbinary representation of it’s hierarchyid. This way I can work with it (almost) normally. In the example, I will ad two extra columns, one for the binary representation and another to show each node label in string mode.

Now let’s assume we’re creating a database for a computer hardware online store. We need a Products table, and a Categories table. In order to provide a decent search and navigation experience across the product catalog in the UI, the best thing to do is to create a product category tree and afterwards relate each category with a product in our list. This way we will have an organized and hierarchical representation of our product catalog.

Let’s assume we already have the following Categories in our Categories table:

1. Products
2. Storage
3. Internal Storage
4. External Storage
5. IDE
6. Sata II
7. SSD
8. Peripherals
9. Mice
10. Keyboards

The idea of this tutorial is to represent a product category tree like so:

* Products (ID 1 – *root*)
  + Storage (ID 2)
    - Internal Storage (ID 3)
      * IDE (ID 4)
      * Sata II (ID 6)
      * SSD (ID 7)
    - External Storage (ID 4)
  + Peripherals (ID
    - Mice (ID 9)
    - Keyboards (ID 10)

This could be a sample of our Products table, with associated *CategoryID*’s:

1. Sunsung 1.5Tb Sata II 3.5” (*CategoryID 5*)
2. Sunsung 500Gb ESata + USB + Ethernet (*CategoryID 3*)
3. Maxedstor 1Gb IDE 3.5” (*CategoryID 4*)
4. ODZ SSD 250Gb (*CategoryID 6*)
5. Macrosoft Explorer Optical Mouse (*CategoryID 8*)
6. Logitak Ultra Flat Keyboard  (*CategoryID 9*)

Sound like underground Chinese brands…

With all our base info set up, we need to map our category tree. But first, we need to create a table to store the hierarchy. So we must execute the following:

|  |  |
| --- | --- |
|  | CREATE TABLE [CategoryTree]( |
|  | [NodeID] [hierarchyid] NOT NULL, |

|  |  |
| --- | --- |
|  | [NodeString] [varchar](512) NOT NULL, |
|  | [NodeBinary] [varbinary](512) NOT NULL, |

|  |  |
| --- | --- |
|  | [CategoryID] [int] NOT NULL |
|  | ) ON [PRIMARY] |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  | ALTER TABLE [dbo].[CategoryTree] WITH CHECK ADD CONSTRAINT [FK\_CategoryTree\_Category] FOREIGN KEY([CategoryID]) |

|  |  |
| --- | --- |
|  | REFERENCES [dbo].[Category] ([ID]) |
|  | GO |

|  |  |
| --- | --- |
|  |  |
|  | ALTER TABLE [dbo].[CategoryTree] CHECK CONSTRAINT [FK\_CategoryTree\_Category] |

|  |  |
| --- | --- |
|  | GO |

Notice how I added a NodeString and NodeBinary columns. This is for me to be able to have a string representation of each node label and also be allowed to know their positioning in the tree. Now for the index:

|  |  |
| --- | --- |
|  | CREATE UNIQUE CLUSTERED INDEX [IX\_CategoryTree] ON [dbo].[CategoryTree] |
|  | ( |

|  |  |
| --- | --- |
|  | [NodeID], |
|  | [CategoryID] |

|  |  |
| --- | --- |
|  | ) |
|  | GO |

Table and index created, it’s time to make the tree grow! Here are the inserts! First, the root “*Produts*”:

|  |  |
| --- | --- |
|  | /\* Insert Root - Products \*/ |
|  | insert into [CategoryTree] |

|  |  |
| --- | --- |
|  | ([NodeID] |
|  | ,[NodeString] |

|  |  |
| --- | --- |
|  | ,[NodeBinary] |
|  | ,[CategoryID]) |

|  |  |
| --- | --- |
|  | values |
|  | (hierarchyid::GetRoot() |

|  |  |
| --- | --- |
|  | ,hierarchyid::GetRoot().ToString() |
|  | ,convert(varbinary, hierarchyid::GetRoot()) |

|  |  |
| --- | --- |
|  | ,11) |

Since I’m adding a root node, I must call *GetRoot()* to get the root node reference. Now, I’ll insert “Storage” and “*Peripherals*” sub-categories. They will both be children of the “Products” category, so I’ll have to make sure I generate the appropriate node reference:

|  |  |
| --- | --- |
|  | /\* Declarations \*/ |
|  | declare @rootNodeId hierarchyid |

|  |  |
| --- | --- |
|  | declare @categoryNodeId hierarchyid |
|  | declare @nodeBinary varbinary |

|  |  |
| --- | --- |
|  | declare @firstChildNodeId hierarchyid |
|  |  |

|  |  |
| --- | --- |
|  | /\* Variable assignments \*/ |
|  | select @rootNodeId = hierarchyid::GetRoot() from [CategoryTree] |

|  |  |
| --- | --- |
|  | select @categoryNodeId = @rootNodeId.GetDescendant(null,null) from [CategoryTree] |
|  | set @nodeBinary = convert(varbinary, @categoryNodeId) |

|  |  |
| --- | --- |
|  |  |
|  | /\* Insert first child - Storage \*/ |

|  |  |
| --- | --- |
|  | insert into [CategoryTree] |
|  | ([NodeID] |

|  |  |
| --- | --- |
|  | ,[NodeString] |
|  | ,[NodeBinary] |

|  |  |
| --- | --- |
|  | ,[CategoryID]) |
|  | values |

|  |  |
| --- | --- |
|  | (@categoryNodeId |
|  | ,@categoryNodeId.ToString() |

|  |  |
| --- | --- |
|  | ,@nodeBinary |
|  | ,2) |

Notice the call to *GetDescendant()* from the root node. Having both parameters set to null, I’m getting a new node reference that is a child of *root*. I then use that reference in the insert. Next, we’ll be inserting the “Peripherals” category, in the same level, right below “Products” and next to “Storage”:

|  |  |
| --- | --- |
|  | /\* Insert second child - Peripherals \*/ |
|  | select @firstChildNodeId = @rootNodeId.GetDescendant(NULL,NULL) |

|  |  |
| --- | --- |
|  | select @categoryNodeId = @rootNodeId.GetDescendant(@firstChildNodeId, null) from [FrontOffice].dbo.[CategoryTree] |
|  | set @nodeBinary = convert(varbinary, @categoryNodeId) |

|  |  |
| --- | --- |
|  |  |
|  | insert into [CategoryTree] |

|  |  |
| --- | --- |
|  | ([NodeID] |
|  | ,[NodeString] |

|  |  |
| --- | --- |
|  | ,[NodeBinary] |
|  | ,[CategoryID]) |

|  |  |
| --- | --- |
|  | values |
|  | (@categoryNodeId |

|  |  |
| --- | --- |
|  | ,@categoryNodeId.ToString() |
|  | ,@nodeBinary |

|  |  |
| --- | --- |
|  | ,8) |

Since we needed to add a new category, at the same level of the “Storage” category, we had to get a node reference next to it, hence the *@rootNodeId.GetDescendant()* call with *@firstChildNode* as the only not null parameter. It allowed us to get a node position after “Storage”. Now, both Storage sub-categories – Internal Storage and External Storage, are needed. Here’s what we must do (keep in mind that this SQL is continuous, so variables hold previous code snippet values):

|  |  |
| --- | --- |
|  | /\* Insert second level child of Storage - Internal Storage \*/ |
|  | /\* @categoryNodeId represents the "Peripherals" category node at this point \*/ |

|  |  |
| --- | --- |
| 0 | select @categoryNodeId = @categoryNodeId.GetAncestor(1).GetDescendant(null,@categoryNodeId).GetDescendant(null,null) |
|  | set @nodeBinary = convert(varbinary, @categoryNodeId) |

|  |  |
| --- | --- |
|  |  |
|  | insert into [FrontOffice].dbo.[CategoryTree] |

|  |  |
| --- | --- |
|  | ([NodeID] |
|  | ,[NodeString] |

|  |  |
| --- | --- |
|  | ,[NodeBinary] |
|  | ,[CategoryID]) |

|  |  |
| --- | --- |
|  | values |
|  | (@categoryNodeId |

|  |  |
| --- | --- |
|  | ,@categoryNodeId.ToString() |
|  | ,@nodeBinary |

|  |  |
| --- | --- |
|  | ,3) |

The reason we had to call *GetAncestor(1).).GetDescendant(null,@categoryNodeId)* was to get a reference to the previous node, the Storage category. It allowed us to go to the parent, and fetch the child right before the “Peripherals” category.

Now onto the “External Storage category:

|  |  |
| --- | --- |
|  | /\* Insert second level child of Storage - External Storage \*/ |
|  | /\* @categoryNodeId represents "Internal Storage" category node at this point \*/ |

|  |  |
| --- | --- |
|  | select @categoryNodeId = @categoryNodeId.GetAncestor(1).GetDescendant(@categoryNodeId,null) |
|  | set @nodeBinary = convert(varbinary, @categoryNodeId) |

|  |  |
| --- | --- |
|  |  |
|  | insert into [FrontOffice].dbo.[CategoryTree] |

|  |  |
| --- | --- |
|  | ([NodeID] |
|  | ,[NodeString] |

|  |  |
| --- | --- |
|  | ,[NodeBinary] |
|  | ,[CategoryID]) |

|  |  |
| --- | --- |
|  | values |
|  | (@categoryNodeId |

|  |  |
| --- | --- |
|  | ,@categoryNodeId.ToString() |
|  | ,@nodeBinary |

|  |  |
| --- | --- |
|  | ,5) |

And so on… There’s no need to explain the rest of the tree creation since it’s all much the same. At this point in time, our category tree table would look like this:

|  |  |  |  |
| --- | --- | --- | --- |
| NodeID | NodeString | NodeBinary | CategoryID |
| 0x | / | 0x | 1 |
| 0×58 | /1/ | 0×58 | 2 |
| 0x5AC0 | /1/1/ | 0x5A | 3 |
| 0x5B40 | /1/2/ | 0x5B | 4 |
| 0×68 | /2/ | 0×68 | 8 |