<https://docs.microsoft.com/en-us/sql/relational-databases/hierarchical-data-sql-server>

# Hierarchical Data (SQL Server)

2017-3-14 13 min to read Contributors

* [BYHAM](https://github.com/BYHAM)

* [Craig Guyer](https://github.com/craigg-msft)

The built-in **hierarchyid** data type makes it easier to store and query hierarchical data. **hierarchyid** is optimized for representing trees, which are the most common type of hierarchical data.

Hierarchical data is defined as a set of data items that are related to each other by hierarchical relationships. Hierarchical relationships exist where one item of data is the parent of another item. Examples of the hierarchical data that is commonly stored in databases include the following:

* An organizational structure
* A file system
* A set of tasks in a project
* A taxonomy of language terms
* A graph of links between Web pages

Use [hierarchyid](https://docs.microsoft.com/en-us/sql/t-sql/data-types/hierarchyid-data-type-method-reference) as a data type to create tables with a hierarchical structure, or to describe the hierarchical structure of data that is stored in another location. Use the [hierarchyid functions](http://msdn.microsoft.com/library/01a050f5-7580-4d5f-807c-7f11423cbb06) in Transact-SQL to query and manage hierarchical data.

## Key Properties of hierarchyid

A value of the **hierarchyid** data type represents a position in a tree hierarchy. Values for **hierarchyid** have the following properties:

* 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\*logAn 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.

* Support for arbitrary insertions and deletions

By using the [GetDescendant](https://docs.microsoft.com/en-us/sql/t-sql/data-types/getdescendant-database-engine) 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.

## Limitations of hierarchyid

The **hierarchyid** data type has the following limitations:

* A column of type **hierarchyid** does not automatically represent a tree. It is up to the application to generate and assign **hierarchyid** values in such a way that the desired relationship between rows is reflected in the values. Some applications might have a column of type **hierarchyid** that indicates the location in a hierarchy defined in another table.
* It is up to the application to manage concurrency in generating and assigning **hierarchyid** values. There is no guarantee that **hierarchyid** values in a column are unique unless the application uses a unique key constraint or enforces uniqueness itself through its own logic.
* Hierarchical relationships represented by **hierarchyid** values are not enforced like a foreign key relationship. It is possible and sometimes appropriate to have a hierarchical relationship where A has a child B, and then A is deleted leaving B with a relationship to a nonexistent record. If this behavior is unacceptable, the application must query for descendants before deleting parents.

## When to Use Alternatives to hierarchyid

Two alternatives to **hierarchyid** for representing hierarchical data are:

* Parent/Child
* XML

**hierarchyid** is generally superior to these alternatives. However, there are specific situations detailed below where the alternatives are likely superior.

### Parent/Child

When using the Parent/Child approach, each row contains a reference to the parent. The following table defines a typical table used to contain the parent and the child rows in a Parent/Child relationship:

Copy

USE AdventureWorks2012 ;

GO

CREATE TABLE ParentChildOrg

(

BusinessEntityID int PRIMARY KEY,

ManagerId int REFERENCES ParentChildOrg(BusinessEntityID),

EmployeeName nvarchar(50)

) ;

GO

Comparing Parent/Child and **hierarchyid** for Common Operations

* Subtree queries are significantly faster with **hierarchyid**.
* Direct descendant queries are slightly slower with **hierarchyid**.
* Moving non-leaf nodes is slower with **hierarchyid**.
* Inserting non-leaf nodes and inserting or moving leaf nodes has the same complexity with **hierarchyid**.

Parent/Child might be superior when the following conditions exist:

* The size of the key is critical. For the same number of nodes, a **hierarchyid** value is equal to or larger than an integer-family (**smallint**, **int**, **bigint**) value. This is only a reason to use Parent/Child in rare cases, because **hierarchyid** has significantly better locality of I/O and CPU complexity than the common table expressions required when you are using a Parent/Child structure.
* Queries rarely query across sections of the hierarchy. In other words, queries usually address only a single point in the hierarchy. In these cases co-location is not important. For example, Parent/Child is superior when the organization table is only used to process payroll for individual employees.
* Non-leaf subtrees move frequently and performance is very important. In a parent/child representation changing the location of a row in a hierarchy affects a single row. Changing the location of a row in a **hierarchyid** usage affects n rows, where n is number of nodes in the sub-tree being moved.

If the non-leaf subtrees move frequently and performance is important, but most of the moves are at a well-defined level of the hierarchy, consider splitting the higher and lower levels into two hierarchies. This makes all moves into leaf-levels of the higher hierarchy. For instance, consider a hierarchy of Web sites hosted by a service. Sites contain many pages arranged in a hierarchical manner. Hosted sites might be moved to other locations in the site hierarchy, but the subordinate pages are rarely re-arranged. This could be represented via:

Copy

CREATE TABLE HostedSites

(

SiteId hierarchyid, PageId hierarchyid

) ;

GO

### XML

An XML document is a tree, and therefore a single XML data type instance can represent a complete hierarchy. In SQL Server when an XML index is created, **hierarchyid** values are used internally to represent the position in the hierarchy.

Using XML data type can be superior when all the following are true:

* The complete hierarchy is always stored and retrieved.
* The data is consumed in XML format by the application.
* Predicate searches are extremely limited and not performance critical.

For example, if an application tracks multiple organizations, always stores and retrieves the complete organizational hierarchy, and does not query into a single organization, a table of the following form might make sense:

Copy

CREATE TABLE XMLOrg

(

Orgid int,

Orgdata xml

) ;

GO

## Indexing Strategies for Hierarchical Data

There are two strategies for indexing hierarchical data:

* **Depth-first**

A depth-first index stores the rows in a subtree near each other. For example, all employees that report through a manager are stored near their managers' record.

In a depth-first index, all nodes in the subtree of a node are co-located. Depth-first indexes are therefore efficient for answering queries about subtrees, such as "Find all files in this folder and its subfolders".

* **Breadth-first**

A breadth-first stores the rows each level of the hierarchy together. For example, the records of employees who directly report to the same manager are stored near each other.

In a breadth-first index all direct children of a node are co-located. Breadth-first indexes are therefore efficient for answering queries about immediate children, such as "Find all employees who report directly to this manager".

Whether to have depth-first, breadth-first, or both, and which to make the clustering key (if any), depends on the relative importance of the above types of queries, and the relative importance of SELECT vs. DML operations. For a detailed example of indexing strategies, see [Tutorial: Using the hierarchyid Data Type](https://docs.microsoft.com/en-us/sql/relational-databases/tables/tutorial-using-the-hierarchyid-data-type).

### Creating Indexes

The GetLevel() method can be used to create a breadth first ordering. In the following example, both breadth-first and depth-first indexes are created:

wmimofCopy

USE AdventureWorks2012 ;

GO

CREATE TABLE Organization

(

BusinessEntityID hierarchyid,

OrgLevel as BusinessEntityID.GetLevel(),

EmployeeName nvarchar(50) NOT NULL

) ;

GO

CREATE CLUSTERED INDEX Org\_Breadth\_First

ON Organization(OrgLevel,BusinessEntityID) ;

GO

CREATE UNIQUE INDEX Org\_Depth\_First

ON Organization(BusinessEntityID) ;

GO

## Examples

### Simple Example

The following example is intentionally simplistic to help you get started. First create a table to hold some geography data.

Copy

CREATE TABLE SimpleDemo

(Level hierarchyid NOT NULL,

Location nvarchar(30) NOT NULL,

LocationType nvarchar(9) NULL);

Now insert data for some continents, countries, states, and cities.

Copy

INSERT SimpleDemo

VALUES

('/1/', 'Europe', 'Continent'),

('/2/', 'South America', 'Continent'),

('/1/1/', 'France', 'Country'),

('/1/1/1/', 'Paris', 'City'),

('/1/2/1/', 'Madrid', 'City'),

('/1/2/', 'Spain', 'Country'),

('/3/', 'Antarctica', 'Continent'),

('/2/1/', 'Brazil', 'Country'),

('/2/1/1/', 'Brasilia', 'City'),

('/2/1/2/', 'Bahia', 'State'),

('/2/1/2/1/', 'Salvador', 'City'),

('/3/1/', 'McMurdo Station', 'City');

Select the data, adding a column that converts the Level data into a text value that is easy to understand. This query also orders the result by the **hierarchyid** data type.

Copy

SELECT CAST(Level AS nvarchar(100)) AS [Converted Level], \*

FROM SimpleDemo ORDER BY Level;

Here is the result set.

Copy

Converted Level Level Location LocationType

/1/ 0x58 Europe Continent

/1/1/ 0x5AC0 France Country

/1/1/1/ 0x5AD6 Paris City

/1/2/ 0x5B40 Spain Country

/1/2/1/ 0x5B56 Madrid City

/2/ 0x68 South America Continent

/2/1/ 0x6AC0 Brazil Country

/2/1/1/ 0x6AD6 Brasilia City

/2/1/2/ 0x6ADA Bahia State

/2/1/2/1/ 0x6ADAB0 Salvador City

/3/ 0x78 Antarctica Continent

/3/1/ 0x7AC0 McMurdo Station City

Notice that the hierarchy is has a valid structure, even though it is not internally consistent. Bahia is the only state. It appears in the hierarchy as a peer of the city Brasilia. Similarly, McMurdo Station does not have a parent country. Users must decide if this type of hierarchy is appropriate for their use.

Add another row and select the results.

Copy

INSERT SimpleDemo

VALUES ('/1/3/1/', 'Kyoto', 'City'), ('/1/3/1/', 'London', 'City');

SELECT CAST(Level AS nvarchar(100)) AS [Converted Level], \* FROM SimpleDemo ORDER BY Level;

This demonstrates more possible problems. Kyoto can be inserted as level /1/3/1/ even though there is no parent level /1/3/. And both London and Kyoto have the same value for the **hierarchyid**. Again, users must decide if this type of hierarchy is appropriate for their use, and block values that are invalid for their usage.

Also, this table did not use the top of the hierarchy '/'. It was omitted because there is no common parent of all the continents. You can add one by adding the whole planet.

Copy

INSERT SimpleDemo

VALUES ('/', 'Earth', 'Planet');

## Related Tasks

### Migrating from Parent/Child to hierarchyid

Most trees are represented using Parent/Child. The easiest way to migrate from a Parent/Child structure to a table using **hierarchyid** is to use a temporary column or a temporary table to keep track of the number of nodes at each level of the hierarchy. For an example of migrating a Parent/Child table, see lesson 1 of [Tutorial: Using the hierarchyid Data Type](https://docs.microsoft.com/en-us/sql/relational-databases/tables/tutorial-using-the-hierarchyid-data-type).

### Managing a Tree Using hierarchyid

Although a **hierarchyid** column does not necessarily represent a tree, an application can easily ensure that it does.

* When generating new values, do one of the following:
  + Keep track of the last child number in the parent row.
  + Compute the last child. Doing this efficiently requires a breadth-first index.
* Enforce uniqueness by creating a unique index on the column, perhaps as part of a clustering key. To ensure that unique values are inserted, do one of the following:
  + Detect unique key violation failures and retry.
  + Determine the uniqueness of each new child node, and insert it as part of a serializable transaction.

#### Example Using Error Detection

In the following example, the sample code computes the new child **EmployeeId** value, and then detects any key violation and returns to **INS\_EMP** marker to recompute the **EmployeeId** value for the new row:

Copy

USE AdventureWorks ;

GO

CREATE TABLE Org\_T1

(

EmployeeId hierarchyid PRIMARY KEY,

OrgLevel AS EmployeeId.GetLevel(),

EmployeeName nvarchar(50)

) ;

GO

CREATE INDEX Org\_BreadthFirst ON Org\_T1(OrgLevel, EmployeeId)

GO

CREATE PROCEDURE AddEmp(@mgrid hierarchyid, @EmpName nvarchar(50) )

AS

BEGIN

DECLARE @last\_child hierarchyid

INS\_EMP:

SELECT @last\_child = MAX(EmployeeId) FROM Org\_T1

WHERE EmployeeId.GetAncestor(1) = @mgrid

INSERT Org\_T1 (EmployeeId, EmployeeName)

SELECT @mgrid.GetDescendant(@last\_child, NULL), @EmpName

-- On error, return to INS\_EMP to recompute @last\_child

IF @@error <> 0 GOTO INS\_EMP

END ;

GO

#### Example Using a Serializable Transaction

The **Org\_BreadthFirst** index ensures that determining **@last\_child** uses a range seek. In addition to other error cases an application might want to check, a duplicate key violation after the insert indicates an attempt to add multiple employees with the same id, and therefore **@last\_child** must be recomputed. The following code uses a serializable transaction and a breadth-first index to compute the new node value:

Copy

CREATE TABLE Org\_T2

(

EmployeeId hierarchyid PRIMARY KEY,

LastChild hierarchyid,

EmployeeName nvarchar(50)

) ;

GO

CREATE PROCEDURE AddEmp(@mgrid hierarchyid, @EmpName nvarchar(50))

AS

BEGIN

DECLARE @last\_child hierarchyid

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRANSACTION

UPDATE Org\_T2

SET @last\_child = LastChild = EmployeeId.GetDescendant(LastChild,NULL)

WHERE EmployeeId = @mgrid

INSERT Org\_T2 (EmployeeId, EmployeeName)

VALUES(@last\_child, @EmpName)

COMMIT

END ;

The following code populates the table with three rows and returns the results:

Copy

INSERT Org\_T2 (EmployeeId, EmployeeName)

VALUES(hierarchyid::GetRoot(), 'David') ;

GO

AddEmp 0x , 'Sariya'

GO

AddEmp 0x58 , 'Mary'

GO

SELECT \* FROM Org\_T2

Here is the result set.

Copy

EmployeeId LastChild EmployeeName

---------- --------- ------------

0x 0x58 David

0x58 0x5AC0 Sariya

0x5AC0 NULL Mary

### Enforcing a tree

The above examples illustrate how an application can ensure that a tree is maintained. To enforce a tree by using constraints, a computed column that defines the parent of each node can be created with a foreign key constraint back to the primary key id.

Copy

CREATE TABLE Org\_T3

(

EmployeeId hierarchyid PRIMARY KEY,

ParentId AS EmployeeId.GetAncestor(1) PERSISTED

REFERENCES Org\_T3(EmployeeId),

LastChild hierarchyid,

EmployeeName nvarchar(50)

)

GO

This method of enforcing a relationship is preferred when code that is not trusted to maintain the hierarchical tree has direct DML access to the table. However this method might reduce performance because the constraint must be checked on every DML operation.

### Finding Ancestors by Using the CLR

A common operation involving two nodes in a hierarchy is to find the lowest common ancestor. This can be written in either Transact-SQL or CLR, because the **hierarchyid** type is available in both. CLR is recommended because performance will be faster.

Use the following CLR code to list ancestors and to find the lowest common ancestor:

Copy

using System;

using System.Collections;

using System.Text;

using Microsoft.SqlServer.Server;

using Microsoft.SqlServer.Types;

public partial class HierarchyId\_Operations

{

[SqlFunction(FillRowMethodName = "FillRow\_ListAncestors")]

public static IEnumerable ListAncestors(SqlHierarchyId h)

{

while (!h.IsNull)

{

yield return (h);

h = h.GetAncestor(1);

}

}

public static void FillRow\_ListAncestors(Object obj, out SqlHierarchyId ancestor)

{

ancestor = (SqlHierarchyId)obj;

}

public static HierarchyId CommonAncestor(SqlHierarchyId h1, HierarchyId h2)

{

while (!h1.IsDescendant(h2))

h1 = h1.GetAncestor(1);

return h1;

}

}

To use the **ListAncestor** and **CommonAncestor** methods in the following Transact-SQL examples, build the DLL and create the **HierarchyId\_Operations** assembly in SQL Server by executing code similar to the following:

Copy

CREATE ASSEMBLY HierarchyId\_Operations

FROM '<path to DLL>\ListAncestors.dll'

GO

### Listing Ancestors

Creating a list of ancestors of a node is a common operation, for instance to show position in an organization. One way of doing this is by using a table-valued-function using the **HierarchyId\_Operations** class defined above:

Using Transact-SQL:

Copy

CREATE FUNCTION ListAncestors (@node hierarchyid)

RETURNS TABLE (node hierarchyid)

AS

EXTERNAL NAME HierarchyId\_Operations.HierarchyId\_Operations.ListAncestors

GO

Example of usage:

Copy

DECLARE @h hierarchyid

SELECT @h = OrgNode

FROM HumanResources.EmployeeDemo

WHERE LoginID = 'adventure-works\janice0' -- /1/1/5/2/

SELECT LoginID, OrgNode.ToString() AS LogicalNode

FROM HumanResources.EmployeeDemo AS ED

JOIN ListAncestors(@h) AS A

ON ED.OrgNode = A.Node

GO

### Finding the Lowest Common Ancestor

Using the **HierarchyId\_Operations** class defined above, create the following Transact-SQL function to find the lowest common ancestor involving two nodes in a hierarchy:

Copy

CREATE FUNCTION CommonAncestor (@node1 hierarchyid, @node2 hierarchyid)

RETURNS hierarchyid

AS

EXTERNAL NAME HierarchyId\_Operations.HierarchyId\_Operations.CommonAncestor

GO

Example of usage:

Copy

DECLARE @h1 hierarchyid, @h2 hierarchyid

SELECT @h1 = OrgNode

FROM HumanResources.EmployeeDemo

WHERE LoginID = 'adventure-works\jossef0' -- Node is /1/1/3/

SELECT @h2 = OrgNode

FROM HumanResources.EmployeeDemo

WHERE LoginID = 'adventure-works\janice0' -- Node is /1/1/5/2/

SELECT OrgNode.ToString() AS LogicalNode, LoginID

FROM HumanResources.EmployeeDemo

WHERE OrgNode = dbo.CommonAncestor(@h1, @h2) ;

The resultant node is /1/1/

### Moving Subtrees

Another common operation is moving subtrees. The procedure below takes the subtree of **@oldMgr** and makes it (including **@oldMgr**) a subtree of **@newMgr**.

Copy

CREATE PROCEDURE MoveOrg(@oldMgr nvarchar(256), @newMgr nvarchar(256) )

AS

BEGIN

DECLARE @nold hierarchyid, @nnew hierarchyid

SELECT @nold = OrgNode FROM HumanResources.EmployeeDemo WHERE LoginID = @oldMgr ;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRANSACTION

SELECT @nnew = OrgNode FROM HumanResources.EmployeeDemo WHERE LoginID = @newMgr ;

SELECT @nnew = @nnew.GetDescendant(max(OrgNode), NULL)

FROM HumanResources.EmployeeDemo WHERE OrgNode.GetAncestor(1)=@nnew ;

UPDATE HumanResources.EmployeeDemo

SET OrgNode = OrgNode.GetReparentedValue(@nold, @nnew)

WHERE OrgNode.IsDescendantOf(@nold) = 1 ;

COMMIT TRANSACTION

END ;

GO

## See Also

[hierarchyid Data Type Method Reference](http://msdn.microsoft.com/library/01a050f5-7580-4d5f-807c-7f11423cbb06)  
[Tutorial: Using the hierarchyid Data Type](https://docs.microsoft.com/en-us/sql/relational-databases/tables/tutorial-using-the-hierarchyid-data-type)  
[hierarchyid (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/data-types/hierarchyid-data-type-method-reference)