# Hierarchical Data

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.

## Key Properties of hierarchyid

**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.

**Support for arbitrary insertions and deletions**

By using the GetDescendant 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 slightlyless 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.

## 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:

CREATE TABLE ParentChildOrg

(

BusinessEntityID int PRIMARY KEY,

ManagerId int REFERENCES ParentChildOrg(BusinessEntityID),

EmployeeName nvarchar(50)

) ;

## 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.

## hierarchyid methods

The **hierarchyid** data type is a variable length, system data type. Use **hierarchyid** to represent position in a hierarchy. 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.

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

### GetAncestor

Returns a **hierarchyid** representing the *n*th ancestor of *this*.

child.GetAncestor ( n )

### GetDescendant

Returns a child node of the parent.

parent.GetDescendant ( child1 , child2 )

### GetLevel

Returns an integer that represents the depth of the node *this* in the tree.

node.GetLevel ( )

### GetRoot

Returns the root of the hierarchy tree. GetRoot() is a static method.

hierarchyid::GetRoot ( )

### IsDescendantOf (Database Engine)

Returns true if *this* is a descendant of parent.

child. IsDescendantOf ( parent )

### Parse (Database Engine)

* Parse 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. Parse() is a static method.

hierarchyid::Parse ( input )

-- This is functionally equivalent to the following syntax

-- which implicitly calls Parse():

CAST ( input AS hierarchyid )

### Read (Database Engine)

* Read reads binary representation of **SqlHierarchyId** from the passed-in **BinaryReader** and sets the **SqlHierarchyId** object to that value. Read cannot be called by using Transact-SQL. Use CAST or CONVERT instead

### GetReparentedValue (Database Engine)

* Returns a node whose path from the root is the path to *newRoot*, followed by the path from *oldRoot* to *this*.

node. GetReparentedValue ( oldRoot, newRoot )

### ToString

* Returns a string with the logical representation of *this*. ToString is called implicitly when a conversion from **hierarchyid** to a string type occurs.

-- Transact-SQL syntax

node.ToString ( )

-- This is functionally equivalent to the following syntax which implicitly calls ToString():

CAST(node AS nvarchar(4000))

### Write (Database Engine)

* Write writes out a binary representation of **SqlHierarchyId** to the passed-in **BinaryWriter**. Write cannot be called by using Transact-SQL. Use CAST or CONVERT instead.