

SQLskills Immersion Event

IETS: Advanced Transact-SQL Programming

Module 8: Alternative Structures

Hierarchies and Open Schema

Bob Beauchemin
BobB@SQLskills.com



Outline

- **Alternate design structures**
 - Designing hierarchies
 - Open schema designs
- **Querying Hierarchies with Recursive Common Table Expressions**
- **HierarchyID Data Type**
- **Querying Sparse Attribute tables with the PIVOT operator**
- **Sparse Columns**



Relational Benefits and Implications

- **The more a data design is normalized, the more tables it contains**
 - Normalization eliminates "special codes"
 - Normalization eliminates data redundancy, update anomalies
 - More tables must be JOINed to produce a result
- **"Special" designs may be useful for special purpose data**
 - Hierarchies
 - Sparse attributes

Hierarchical Data

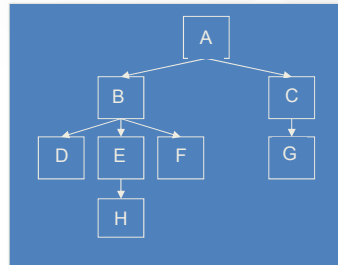
- **Hierarchical data consists of nodes and edges**
 - In employee-boss relationship, employee and boss are each nodes, the relationship between them is an edge
- **Hierarchical data can be modeled in relational as**
 - Adjacency model - separate column for edge
 - Column can either be in same or separate table
 - Most common
 - Path Enumeration model - column w/hierarchical path
 - Nested Set model - adds "left" and "right" columns to represent edges, which must be maintained separately

Adjacency Model

```
-- adjacent model, same table

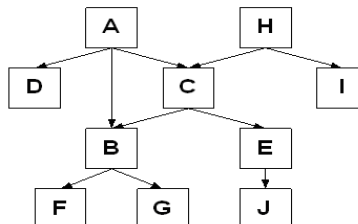
CREATE TABLE dbo.Personnel (
  empid INT PRIMARY KEY, -- node
  bossid INT NULL,       -- edge
  salary DECIMAL(6,2) NOT NULL
)
GO

ALTER TABLE dbo.Personnel
  ADD CONSTRAINT FK_BossEmp
  FOREIGN KEY(bossid)
  REFERENCES Personnel (empid)
```



Multi-Parent Hierarchies

- Hierarchies can be multi parent - BOM
 - No node can be the parent of itself
 - Adjacency data is defined in a separate table
 - One row for each "contains" relationship
 - Cycles prevented with constraints
 - Reference: AdventureWorks.Production.BillOfMaterials



Adjacency Model – Separate Tables

```
-- adjacency model, different table

CREATE TABLE BOM(
    BOMID int IDENTITY PRIMARY KEY,
    AssemblyID int NULL,      -- node
    ComponentID int NOT NULL, -- edge
    UnitMeasureCode nchar(3) NOT NULL,
    BOMLevel smallint NOT NULL,
    PerAssemblyQty decimal(8, 2) NOT NULL
);
GO

ALTER TABLE BOM ADD CONSTRAINT FK_BillofMaterialsP FOREIGN KEY(ComponentID)
REFERENCES Product (ProductID)

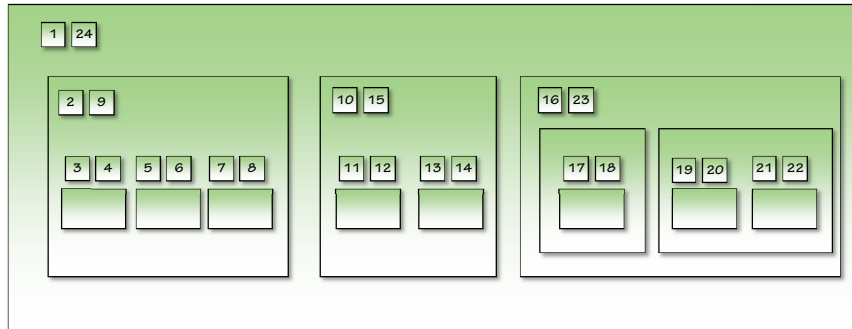
ALTER TABLE BOM ADD CONSTRAINT FK_BillofMaterialsA FOREIGN KEY(AssemblyID)
REFERENCES Product (ProductID)
```

Path Enumeration Model

- **Define all paths as a separate table**
 - Starting node of the path
 - Ending node of the path
 - Path length
- **Table is built recursively until the set of paths are complete**
 - Separates nodes and edges in separate table
 - Nodes are entities, edges are relationships
- **You can use node enumeration or edge enumeration**
- **XML and HierarchyID data types uses variation of this model known as ORDPATH**

Nested Set Model

- **Nested Set Model models hierarchies as containment hierarchy**
 - LEFT and RIGHT (or top to bottom) columns added to each row to represent traversal through the hierarchy



Tree Operations

- **In all models, tree operations are modeled as relational operations**
 - SELECT operations
 - Get descendants (subordinates)
 - Get number of descendants (subordinates)
 - Search up management chain (ancestors)
 - Get all leaf nodes (workers, no subordinates)
 - INSERT, UPDATE DELETE operations
 - On individuals
 - On trees

Recursive Calculations

- **Hierarchy may be of inconsistent depth**
 - chart of accounts and parts list are typical
- **Calculations require traversal of hierarchy**
- **Many useful recursive calculations possible**
 - aggregates, *e.g.* sum of leaves is rollup value of account
 - leaves, *e.g.* bill of materials for parts list

Recursive CTE

- **CTE can do recursive calculation**
- **Recursive CTE has three parts**
 - anchor precedes UNION ALL does initialization
 - recursive member after UNION ALL; recurses until no results
 - outer select; selects results to be returned

anchor; executed once	→	WITH descendant(parent, id, amount) AS
		(SELECT parent, id, amount
		FROM partsTree WHERE id = @start
		UNION ALL
recursive member; repeated	→	SELECT P.parent, P.id, P.amount
joined with previous recursion	→	FROM partsTree AS P INNER JOIN
		descendant AS A ON P.parent = A.id
		)
outer select; id's returned	→	SELECT id FROM descendant

Logic

- **Recursive CTE Logic**

- Anchor member is activated. Set R0 (R for Results) is generated
- Recursive member is activated, retrieving set Ri (i = step number) as input to RecursiveCTE, producing Set Ri + 1
- Step 2 is run repeatedly, incrementing the step number each time, until an empty set is returned
- The outer query is executed, retrieving a cumulative (UNION ALL) result of all of the previous steps when referring to the RecursiveCTE

Recursive Queries - Rules and Limits

- **Rules for anchor and recursive part**

- Anchor cannot self-reference CTE
- Recursive part always self-references CTE

- **Some limits on anchor and recursive part**

- Recursive part cannot contain "SELECT DISTINCT" behavior
 - GROUP BY
 - HAVING
 - scalar aggregation
 - TOP
 - outer joins
- Anchor and recursive must be separated by UNION ALL
 - not UNION

Traversing Up The Tree

- Both examples have shown traversing the tree downwards
 - Employees for a given manager
 - Subparts for a given part
- To traverse up the tree, reverse the roles of parent and child in the recursive component

```
WITH ancestor(parent, id, amount) AS
(SELECT parent, id, amount
 FROM partsTree WHERE id = @start
 UNION ALL
 SELECT P.parent, P.id, P.amount
 FROM partsTree AS P INNER JOIN
 ancestor AS A ON P.id = A.Parent
 )
SELECT id FROM ancestor
```

Limiting the Levels Returned

- A level indicator can be used to limit the levels of hierarchy returned
 - Initialize level in anchor
 - Increment and check level in recursive query

```
WITH descendant(parent, id, amount, level) AS
(SELECT parent, id, amount, level = 1
 FROM partsTree WHERE id = @start
 UNION ALL
 SELECT P.parent, P.id, P.amount, level = level + 1
 FROM partsTree AS P INNER JOIN
 descendant AS A ON A.id = P.parent
 WHERE level < 2
 )
SELECT id FROM descendant
```


SQL Server 2008+ HierarchyID

- **Implementation of path enumeration model**
 - Uses ORDPATH internally for speed
- **Variable length to 900 bytes**
- **Comparable type - can be used as index key**
- **SQLCLR based system type**
 - Use '.' syntax for properties
 - Use '.' syntax for instance methods
 - Use '::' syntax for static methods
 - Methods and Properties are case-sensitive
 - Useable on .NET clients directly as SqlHierarchyId

HierarchyID

- **Type can represent position in hierarchy**
 - Including order among siblings
- **"Level" property - for breadth-first indexing**
- **Methods for common hierarchical operations**
 - GetRoot
 - GetLevel
 - GetAncestor
 - IsDescendantOf
 - GetDescendant
 - GetReparentedValue

GetDescendant

- **GetDescendant** used to add nodes
 - Used to get new child node for an existing parent
- **GetDescendant(NULL, NULL)**
 - Gets child number 1
- **GetDescendant(sibling_hid, NULL)**
 - Gets child to the right of sibling_hid
- **GetDescendant(NULL, sibling_hid)**
 - Gets child to the left of sibling_hid
- **GetDescendant(sibling_hid1, sibling_hid2)**
 - Gets child between siblings

Constructing the Hierarchy

- **To populate root use the GetRoot method**
- **To insert a node after last child**
 - Use parent.GetAncestor(1) to get children
 - Select lastchild = max from children
 - Insert after last child using parent.GetDescendent(lastchild)
- **Does not enforce tree structure**
 - Can enforce tree using constraints

Querying Hierarchies

- HierarchyID is a position in a hierarchy
- Querying hierarchy shape combines HierarchyID methods and relational joins
- IsDescendantOf used on child node
 - Gets all levels of hierarchy
- GetAncestor(n) used on child node
 - Gets specific level of hierarchy
- GetLevel property gets the level of hierarchy relative to the (1-based) root

Querying a Hierarchy

```
-- All subordinates of employee 3
SELECT C.BusinessEntityId, C.LoginID, C.OrganizationLevel,
       C.OrganizationNode.ToString() as OrgNode
FROM HumanResources.Employee AS P
JOIN HumanResources.Employee AS C ON P.BusinessEntityId = 3
AND C.OrganizationNode.IsDescendantOf(P.OrganizationNode) = 1;
```

```
-- All managers of employee 11
SELECT P.BusinessEntityId, P.LoginID, P.OrganizationLevel,
       P.OrganizationNode.ToString() as OrgNode
FROM HumanResources.Employee AS P
JOIN HumanResources.Employee AS C ON C.BusinessEntityId = 11
AND C.OrganizationNode.IsDescendantOf(P.OrganizationNode) = 1;
```

```
-- All direct subordinates of employee 9
SELECT C.BusinessEntityId, C.LoginID,
       C.OrganizationNode.ToString() as OrgNode
FROM HumanResources.Employee AS P
JOIN HumanResources.Employee AS C ON P.BusinessEntityId = 9
AND C.OrganizationNode.GetAncestor(1) = P.OrganizationNode;
```

HierarchyID and Indexing

- **HID data type is depth-first index**
 - where `hid.IsDescendantOf(@parentid) = 1` can use index seek
- **Level, HID is breadth-first index**
 - where `hid.GetAncestor(1) = @parentid` can use index seek

Moving Nodes And Trees

- **Use GetReparentedValue**
 - Method called on node to be moved
 - Parameters - old parent, new parent
 - It does not update the nodes, you must update
- **To move an entire tree (including root)**
 - Use GetDescendant to get a new node under the new parent
 - Update HierarchyID of all the nodes in the subtree to the new node

Extending HierarchyID

- **HierarchyID is exposed as SQLCLR type**
 - Additional methods using SQLCLR functions
 - Functions available on server (UDF) or client
- **HierarchyID can be extended**
 - Encapsulate the type
 - Inheritance not possible, HierarchyID is a struct
 - Add instance methods and properties

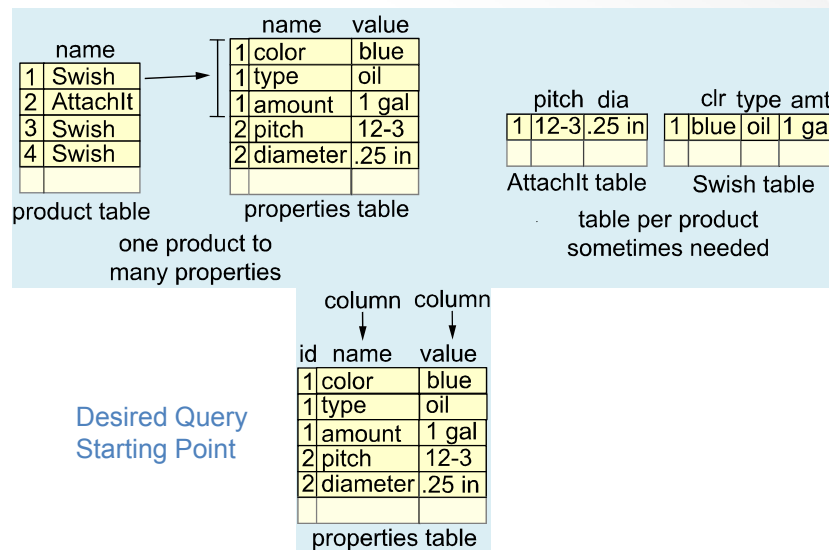
Open Schema

- **Many designs require sparse properties**
 - Hardware store has different attributes for each product
 - Lab tests have different readings for each test
 - Directory systems have different attributes for each item
- **These are name-value pairs (property bags)**
- **Because they don't appear on each tuple (row) they are difficult to model**

Modeling Sparse Properties

- **Sparse Properties often modeled as separate table**
 - Base table has one row per item - common properties
 - Property table has N rows per item - one per property
 - Known as Entity-Attribute-Value (EAV)
- **Can be modeled as sparse tables**
 - 256 table limit in SQL Server JOIN
- **Can be modeled as sparse columns**
 - 1024 column limit in SQL Server tables
- **Can be modeled as XML**
 - Common properties are elements, sparse are attributes

Sparse Properties



EAV Design

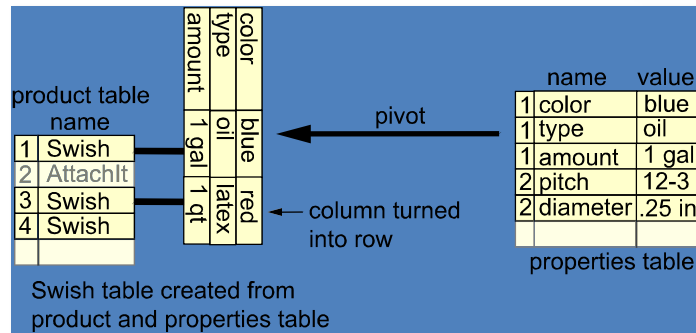
- This design is known as open schema
 - Or entity-attribute-value (EAV)
 - Used for modeling products with many different attributes
 - auction items
 - hardware store items
 - Must be *extremely* vigilant about editing
 - Difficult to query because you must turn values into columns

SQL Server 2005+ Additions

- SQL Server 2005+ adds PIVOT and UNPIVOT
- PIVOT changes column values into columns
 - Makes EAV property table into sparse column table
- UNPIVOT changes columns to column values
 - Sparse column table to EAV property table

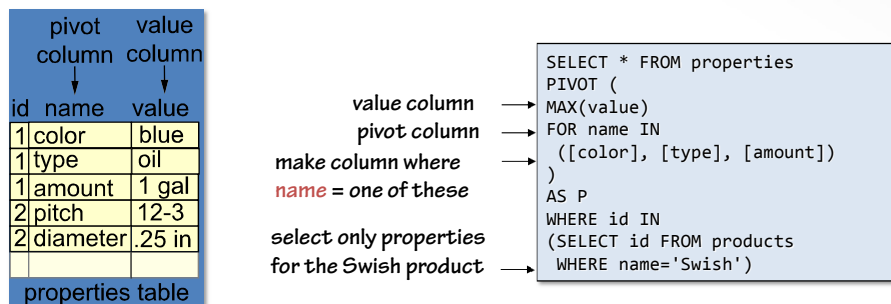
Pivot and EAV

- Pivot turns column values into rows
 - in effect it synthesizes a table
- Widens table by adding columns to it
 - rotate "many" table from 1-to-many solution



Basic Pivot

- Pivot needs three basic values
 - columns that makeup rotated table
 - column that contains value for rotated table
 - pivot column, i.e. the many in the one to many relation



Basic Pivot Results

- Pivot selects all rows for a particular product
- Columns not mentioned in pivot used to group properties

id not mentioned
in pivot expression
↓

properties grouped by id →

id	color	type	amount
1	blue	oil	1 gal
3	red	latex	1 qt
4	white	oil	1 pt

pivoted properties of Swish product

UNPIVOT

- UNPIVOT
- Reverses the PIVOT operation
- Normalizes the pre-pivoted data
- Rotates columns into rows
- Eliminates NULLs in the results value column
- Usually is an exact reversal with open schema
 - Not an exact reversal of the PIVOT operator when aggregation used

UNPIVOT Using the PIVOT output

```
WITH VTable AS -- PIVOT using CTE as input
(
  SELECT * FROM properties
  WHERE id IN (SELECT id FROM products WHERE name = 'Swish')
)
SELECT * INTO painttab FROM VTable
PIVOT (
  MAX(value) FOR name IN ([color], [type], [amount])
) AS P
go
SELECT * FROM painttab
go

--results in original table - NO attributes with NULL values
SELECT id, attribute, value
FROM painttab
UNPIVOT
(
  value FOR attribute IN ([color], [type], [amount])
) AS U
go
```

SQL Server 2008 Sparse Columns

- Sparse Column extends column limit
- Still 1024 column limit for "non-sparse" columns
- Up to 30000 sparse columns
- Column marked as SPARSE in table definition
- Additional column represents all sparse column name value pairs as attributes in a single XML element

Inserts And Sparse Columns

```
-- Create a table with sparse properties and column_set
Create Table Products(
    Id int, Type nvarchar(16),
    Resolution nvarchar(8) SPARSE,
    ZoomLength nvarchar(8) SPARSE,
    WaistSize int SPARSE,
    Length int SPARSE,
    ProductProperties XML
    COLUMN_SET FOR ALL_SPARSE_COLUMNS);

-- Ordinary INSERT
INSERT INTO Products(Id, Type, Resolution, ZoomLength)
VALUES (101, 'Camera', '6 mb', '3x');

-- Insert a new product through sparse_column_set.
DECLARE @x XML =
    '<WaistSize>32</WaistSize><Length>32</Length>'
INSERT INTO Products (Id, Type, ProductProperties)
VALUES (5002, 'Pant', @x);
```

Sparse Column Operations

```
-- USING 'SELECT *'
-- selects non-sparse columns and column set
SELECT * FROM Products

-- Can SELECT Individual Columns
SELECT Id, Resolution, ZoomLength
FROM Products
WHERE Type='camera';

-- UPDATE through column set
-- sets omitted columns to NULL
DECLARE @x XML =
    '<Resolution>10mb</Resolution><ZoomLength>5x</ZoomLength>'
UPDATE Products
SET ProductProperties = @xml
WHERE Id = 101;
```

Filtered Indexes and Statistics

- **In a sparse column design, 95% of values for a column can be NULL**
 - SQL Server 2008 implements efficient storage
- **Filtered indexes can be used with sparse columns**
- **Filtered statistics can be kept on sparse columns**
 - Can keep track of only non-null value distribution

Summary

- **Alternate design structures**
 - Designing hierarchies
 - Open schema designs
- **Query alternate design structures with**
 - Recursive CTEs hierarchies
 - PIVOT - makes EAV look like relational form
 - UNPIVOT - changes back to EAV form
- **Sparse column features in SQL Server 2008+**

References

- Joe Celko's Trees and Hierarchies in SQL For Smarties, Morgan Kaufmann, 2004
- ORDPATHs: Insert-Friendly XML Node Labels
 - <http://www.cs.umb.edu/~poneil/ordpath.pdf>
- Model Your Data Hierarchies With SQL Server 2008
 - <http://msdn.microsoft.com/en-us/magazine/cc794278.aspx>
- SQL Server 2008's HierarchyID
 - <http://www.sqlmag.com/Articles/ArticleID/99369>
 - <http://www.sqlmag.com/Articles/ArticleID/99036>
- Webcast: T-SQL Enhancements in SQL Server 2008