Horizontal Trees

R Glen Cooper
r.glen.cooper@gmail.com

Introduction

The **sp_HorizontalTree** procedure introduced here may be used to audit tables with dependent columns.

A column is dependent on another if the meaning of the first column depends on the meaning of the second. For example, in a table containing accounting information, the meaning of a Fiscal Quarter column depends on the Fiscal Year to which it belongs.

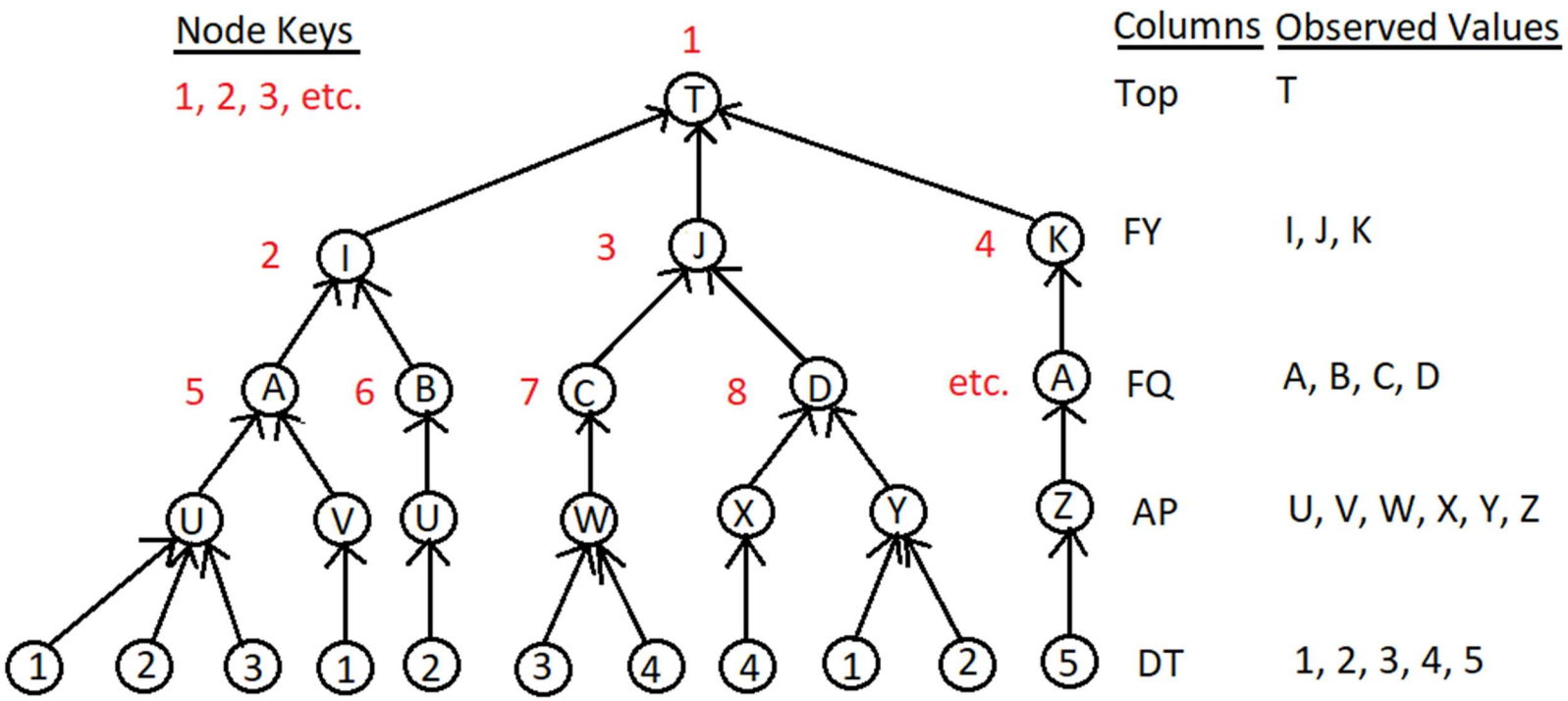
When manually building tables with dependent columns, you may wish to audit the values that one column has for a given value of another. For example, does every Fiscal Year in your table have four Fiscal Quarters? Do they have the same number of dates? Which Accounting Periods span multiple Fiscal Quarters? Which ones have the largest number of dates? What are those dates?

This proc provides a table summarizing the dependencies between selected columns on any table, so questions like the above can be easily answered (often by just looking at the summary). This summary table makes no reference to the column names of the original table, so any SQL written for it won't use them either. Because of that, SQL written for the summary of one table can be applied to the summary of any other table. It will know about column values, however, so any SQL that references them would only be meaningful when comparing tables whose selected columns have similar content.

The program works by building a tree with the observed values of the first column sitting below the top node as children, and below each of them are the observed values of the second column, and below each of them are the observed values of the third column, and so on. At each depth of the tree, the observed values may repeat but below any parent they are always distinct.

This solution is reminiscent of the one developed in a previous article [1] where a table with a parent relationship between records is viewed as a tree whose node attributes **numChildren**, **numDescendants**, **numLeaves**, **Height**, **Depth** allows you to easily build sophisticated queries on the table. For such tables one migrates from node to node by scrolling up and down the table (vertical trees). Here, however, a tree is dynamically built from the table itself using its dependent columns across the top (horizontal trees).

The following tree represents the four columns FY, FQ, AP and DT of a simple accounting table with dummy data (the proc doesn't care much about data types):



Here, three values are observed for FY (I, J, K), four values for FQ (A, B, C, D), six values for AP (U, V, W, X, Y, Z) and five values for DT (1,2,3,4,5). Below each parent the names of its children (ie. the values of their common column) are always distinct, by way of how they're defined, but they may overlap with the values of another parent at the same depth (or even different depths).

The tree's underlying table has three columns: **NodeId** (primary key of the table), **Node** (name of the node) and **ParentId** (foreign key to a record's parent). The primary key values are denoted in red in the above diagram.

What does this tree mean?

One way to explain it is the following:

The top node represents the set of all records in the data set. The nodes directly below it represent the various (necessarily disjoint and non-empty) subsets of those records with a specific value on the first column. So, node 2 is the set of records whose FY value is I, node 3 is the set of records whose FY value is J and so on. The nodes below them are defined in the same way. So, node 5 is the set of records of node 2 whose FQ value is A, etc. Note that the children of any node always partition that node's set of records, and for a given depth they partition the entire data set.

Another way of viewing the tree is to recognize that each path from the top node to one of its leaves represents all the records whose column values agree with the node names (or values) on that path.

For example, the left-most path represents all the records in the data set whose column values for FY, FQ, AP, DT are I, A, U, 1 (respectively). If the column of the leaf nodes (DT) is a primary key of the table from which it was defined, every path represents a unique record (but it doesn't have to be that way in general). The arrows represent the subset relation of a set of records, but they can also be viewed in a different way. For example, there are two children of node 5 which means that for any record in the data set whose first two columns are I and A, the third column must be U or V. Furthermore, its child U has three children while V has only one. This may be of interest to you when examining how the value of one column (with its ancestors) affects the possible values of the next one. In other words, the tree describes the "shape" of the data.

The **sp_HorizontalTree** proc builds the tree's underlying table before passing it to the **sp_VerifyTree** proc [1] which verifies that it's a tree and computes its node attributes numChildren, numDescendants, numLeaves, Height, Depth. These attributes may be employed to study the shape of the data using simple SQL (which would be harder to write without these pre-computed values).

For example, the following query lists the Accounting Periods spanning multiple Fiscal Quarters for any accounting table:

```
SELECT Node FROM ##HorizTree WHERE Depth = 3
GROUP BY NODE
HAVING COUNT(*) > 1
```

The ##HorizTree table is the proc's global temporary table defining the tree. Its name has a random suffix generated by the proc at run-time to avoid clashing with other users (but is not generally shown here for simplicity).

The result for the accounting table in the upcoming sample audit is:

```
2019-04
2019-07
2019-10
2020-04
2020-07
2020-10
2021-04
2021-07
2021-10
```

In fact, this query can be used on the horizontal tree of any table since it knows nothing about column names. It simply lists the names of those nodes of depth = 3 that occur more than once. So, it can be applied to any accounting table, regardless of its column names. In this manner a library of auditing queries for all future tables may be built. For non-accounting tables this query may, or may not, be of interest.

Sample Audit

The **D_DATE** dimension table that's commonly found in a database warehouse (but with more columns) will now be audited. A script to generate this table can be found in the resource section.

First, run the proc to generate its tree. You'll need to specify the server, database, schema, table and list of dependent columns ordered by dependency. That is, the second column is dependent on the first, the third is dependent on the second, and so on. As well, another parameter (@FirstNodes) lists the values of the first column that will filter the data set before processing begins. The proc doesn't really care about column data types, but the first column must be character based. Extensive error checking ensures that the parameters and incoming data make sense.

The following invocation of the proc examines **Fiscal_Year**, **Fiscal_Yr_Qtr**, **Accounting_Period**, **Date_Dim_Key** of D_DATE:

```
DECLARE @ERROR_NUMBER   INT
DECLARE @ERROR_MESSAGE  VARCHAR(MAX)
DECLARE @ROW_COUNT      INT

EXEC @ERROR_NUMBER      =
[dbo].[sp_HorizontalTree]
@Server           = (Your Server)
,@Database        = (Your Database)
,@Schema          = 'dbo'
,@Table           = 'D_DATE'
,@Columns         = 'Fiscal_Year, Fiscal_Yr_Qtr, Accounting_Period, Date_Dim_Key'
,@FirstNodes      = '2019/2020,2020/2021, 2021/2022'
,@DEBUG           = 0
,@ERROR_NUMBER    = @ERROR_NUMBER OUTPUT
,@ERROR_MESSAGE   = @ERROR_MESSAGE OUTPUT
,@ROW_COUNT       = @ROW_COUNT OUTPUT

SELECT @ERROR_NUMBER AS ERROR_NUMBER, @ERROR_MESSAGE AS ERROR_MESSAGE, @ROW_COUNT AS ROW_COUNT -- Optional
```

The output is the following tree:

NodeId	Node	ParentId	NumChildren	NumDescendants	NumLeaves	Height	Depth
1	Top_Node	NULL	3	1159	1096	4	0
2	2019/2020	1	4	386	366	3	1
3	2020/2021	1	4	385	365	3	1
4	2021/2022	1	4	385	365	3	1
5	2020 Q2	3	4	96	92	2	2
6	2021 Q3	4	4	96	92	2	2
7	2020 Q3	3	4	96	92	2	2

8	2021 Q1	4	4	95	91	2	2
9	2019 Q2	2	4	96	92	2	2
10	2020 Q4	3	4	94	90	2	2
11	2019 Q4	2	4	95	91	2	2
12	2021 Q2	4	4	96	92	2	2
13	2021 Q4	4	4	94	90	2	2
14	2020 Q1	3	4	95	91	2	2
15	2019 Q1	2	4	95	91	2	2
16	2019 Q3	2	4	96	92	2	2
17	2019-01	15	32	32	32	1	3
18	2019-02	15	28	28	28	1	3
19	2019-03	15	28	28	28	1	3
20	2019-04	9	25	25	25	1	3
21	2019-04	15	3	3	3	1	3
22	2019-05	9	28	28	28	1	3
23	2019-06	9	28	28	28	1	3
24	2019-07	9	11	11	11	1	3
25	2019-07	16	17	17	17	1	3
26	2019-08	16	28	28	28	1	3
27	2019-09	16	28	28	28	1	3
28	2019-10	11	9	9	9	1	3
29	2019-10	16	19	19	19	1	3
30	2019-11	11	28	28	28	1	3
31	2019-12	11	28	28	28	1	3
32	2019-13	11	26	26	26	1	3
33	2020-01	14	30	30	30	1	3
34	2020-02	14	28	28	28	1	3
35	2020-03	14	28	28	28	1	3
36	2020-04	5	23	23	23	1	3
37	2020-04	14	5	5	5	1	3
38	2020-05	5	28	28	28	1	3
39	2020-06	5	28	28	28	1	3
40	2020-07	5	13	13	13	1	3
41	2020-07	7	15	15	15	1	3
42	2020-08	7	28	28	28	1	3
43	2020-09	7	28	28	28	1	3
44	2020-10	7	21	21	21	1	3
45	2020-10	10	7	7	7	1	3
46	2020-11	10	28	28	28	1	3

47	2020-12	10	28	28	28	1	3
48	2020-13	10	27	27	27	1	3
49	2021-01	8	29	29	29	1	3
50	2021-02	8	28	28	28	1	3
51	2021-03	8	28	28	28	1	3
52	2021-04	8	6	6	6	1	3
53	2021-04	12	22	22	22	1	3
54	2021-05	12	28	28	28	1	3
55	2021-06	12	28	28	28	1	3
56	2021-07	6	14	14	14	1	3
57	2021-07	12	14	14	14	1	3
58	2021-08	6	28	28	28	1	3
59	2021-09	6	28	28	28	1	3
60	2021-10	6	22	22	22	1	3
61	2021-10	13	6	6	6	1	3
62	2021-11	13	28	28	28	1	3
63	2021-12	13	28	28	28	1	3
64	2021-13	13	28	28	28	1	3
65	20190401	17	0	0	1	0	4
66	20190402	17	0	0	1	0	4

etc.
etc.

ERROR_NUMBER	ERROR_MESSAGE	ROW_COUNT
0		1160

Note that the SELECT @ERROR_NUMBER query in the proc's invocation is optional, but without it you will receive no error messages when running the proc. At the very least, check that @ERROR_NUMBER is 0 before proceeding further in your calling application. Errors in the parameter values, for example, will prevent any tree building but the SELECT @ERROR_NUMBER query will tell you why. ROW_COUNT is the number of nodes in the tree.

Now that you have your tree, you can begin to audit its underlying table for accuracy.

Assuming you don't know anything about leap years, the first thing you might ask is: why does node '2019/2020' have 366 days? How did that happen in the model you're auditing? Are duplicate dates present?

Looking at the node's children: **SELECT * FROM ##HorizTree WHERE ParentId = 2** shows there are four Fiscal Quarters (one of which must contain the extra date):

NodeId	Node	ParentId	NumChildren	NumDescendants	NumLeaves	Height	Depth
9	2019 Q2	2	4	96	92	2	2
11	2019 Q4	2	4	95	91	2	2
15	2019 Q1	2	4	95	91	2	2
16	2019 Q3	2	4	96	92	2	2

Looking at their children: **SELECT * FROM ##HorizTree WHERE ParentId IN (9, 11, 15, 16)** shows there are 16 Accounting Periods (one of which must contain the extra date)::

NodeId	Node	ParentId	NumChildren	NumDescendants	NumLeaves	Height	Depth
17	2019-01	15	32	32	32	1	3
18	2019-02	15	28	28	28	1	3
19	2019-03	15	28	28	28	1	3
20	2019-04	9	25	25	25	1	3
21	2019-04	15	3	3	3	1	3
22	2019-05	9	28	28	28	1	3
23	2019-06	9	28	28	28	1	3
24	2019-07	9	11	11	11	1	3
25	2019-07	16	17	17	17	1	3
26	2019-08	16	28	28	28	1	3
27	2019-09	16	28	28	28	1	3
28	2019-10	11	9	9	9	1	3
29	2019-10	16	19	19	19	1	3
30	2019-11	11	28	28	28	1	3
31	2019-12	11	28	28	28	1	3
32	2019-13	11	26	26	26	1	3

Browsing through their children **SELECT * FROM ##HorizTree WHERE ParentId IN(17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32)** you find the date you never expected:

NodeId	Node	ParentId	NumChildren	NumDescendants	NumLeaves	Height	Depth
399	20200229	31	0	0	1	0	4

You've just discovered leap years. You'll probably want to write some SQL to audit any future accounting tables to ensure that leap years are included for every Fiscal Year divisible by 4. Of course, your SQL will need to assume that the node directly below the top node represents Fiscal Year so it can perform the necessary computations (recall that the column names are unknown). And your SQL will be restricted to tables whose first column (whatever it's called) looks like Fiscal Year,

The actual name of the global temporary table holding the tree (for a given session) may be found in the SELECT @ERROR_NUMBER query or the messages tab of SSMS whenever @DEBUG = 1 (eg. **##HorizTree51407**). The tree's name ends in a random 5-character suffix (eg. 51407) which, like all the global temporary tables created by the proc, is randomly generated when the proc begins to avoid clashing with other sessions. Therefore, you can run the proc simultaneously in other panes of SSMS or write @SQL on the global temporary tables (including the source table). But note that when @DEBUG = 0 all such tables are dropped when the proc concludes. In any event, dropping the pane from which the proc was run will drop all tables that the proc created (such as the temporary lookup tables for the columns in the original table).

Special feature of horizontal trees

Because horizontal trees are built from data sets, some interesting attributes may be added to the tree. For example, each path in the tree represents the set of records in the data set whose column values agree with the nodes on that path. An attribute related to this would be **numRecords**, which counts the number of records passing through each node. That way, one could compute the probability of realizing the possible values of any parent's children. In other words, show which children are more favoured to appear in the data set (given their genealogy). The computation for this would be nearly identical to that which computes numDescendants (simply compute the volume of records for each leaf and work upwards).

How does the proc work?

First, it copies the table @Table into a global temporary table **##Table** upon which filtering occurs. It then builds lookup global temporary tables for the observed values of each column. After that it builds a global temporary table **##TempStage** containing just the columns appearing in the @Columns parameter, along with lookup columns for each of them that reference the column lookup tables. From here it builds the tree in the table **##HorizTree**, one column at a time.

Set @DEBUG = 1 to view these tables as the tree is being built, where these views include a new column (Column 0, Column1, etc.) displaying the name of the lookup table.

For the above example:

Lookup table for top node:

Column 0	NodeId	Node
##TopNode	1	TOP

Lookup table for first column:

Column 1	NodeId	Node
##Fiscal_Year	2	2019/2020
##Fiscal_Year	3	2020/2021

##Fiscal_Year	4	2021/2022

At this point the first four nodes of the tree may be created using the original NodeId values, where the top node is the parent:

NodeId	Node	ParentId
1	Top_Node	NULL
2	2019/2020	1
3	2020/2021	1
4	2021/2022	1

But the lookup tables for the remaining columns have NodeId starting at 1, which will clash with the nodes already inserted. The script has no idea what these values should be until the preceding insertions have occurred. That's where ##TempTable comes in. Assigning the start value for its NodeId as one more than the current maximum NodeId in the tree, the current column is inserted into a fresh copy of ##TempTable without its NodeId column. Those NodeId values are automatically computed by the IDENTITY specification of ##TempTable before being inserted into the tree. After the insertion, the lookup values for the current column are updated in ##TempStage using the tree's newly created key values, so the foreign key of the next column will be correct when it's processed. As mentioned above, the foreign key for any column is the primary key of the column preceding it.

Lookup table for second column:

Column 2	NodeId	Node
##Fiscal_Yr_Qtr	1	2021 Q2
##Fiscal_Yr_Qtr	2	2020 Q2
##Fiscal_Yr_Qtr	3	2021 Q3
##Fiscal_Yr_Qtr	4	2021 Q4
##Fiscal_Yr_Qtr	5	2021 Q1
##Fiscal_Yr_Qtr	6	2019 Q3
##Fiscal_Yr_Qtr	7	2019 Q4
##Fiscal_Yr_Qtr	8	2019 Q2
##Fiscal_Yr_Qtr	9	2019 Q1
##Fiscal_Yr_Qtr	10	2020 Q3
##Fiscal_Yr_Qtr	11	2020 Q1
##Fiscal_Yr_Qtr	12	2020 Q4

Lookup table for third column:

Column 3	NodeId	Node
##Accounting_Period	1	2019-01
##Accounting_Period	2	2019-02
##Accounting_Period	3	2019-03
##Accounting_Period	4	2019-04
##Accounting_Period	5	2019-05
##Accounting_Period	6	2019-06
##Accounting_Period	7	2019-07
##Accounting_Period	8	2019-08
##Accounting_Period	9	2019-09
##Accounting_Period	10	2019-10
##Accounting_Period	11	2019-11
##Accounting_Period	12	2019-12
##Accounting_Period	13	2019-13
##Accounting_Period	14	2020-01
##Accounting_Period	15	2020-02

etc.
etc.

Lookup table for fourth column:

Column 4	NodeId	Node
##Date_Dim_Key	1	20190401
##Date_Dim_Key	2	20190402
##Date_Dim_Key	3	20190403
##Date_Dim_Key	4	20190404
##Date_Dim_Key	5	20190405
##Date_Dim_Key	6	20190406
##Date_Dim_Key	7	20190407
##Date_Dim_Key	8	20190408
##Date_Dim_Key	9	20190409
##Date_Dim_Key	10	20190410
##Date_Dim_Key	11	20190411
##Date_Dim_Key	12	20190412

etc.
etc.

##TempStage before initial lookup values are computed:

I	Top_Node	Top_Node_Id	Fiscal_Year	Fiscal_Year_Id	Fiscal_Yr_Qtr	Fiscal_Yr_Qtr_Id	Accounting_Period	Accounting_Period_Id	Date_Dim_Key	Date_Dim_Key_Id
1	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190401	NULL
2	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190402	NULL
3	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190403	NULL
4	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190404	NULL
5	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190405	NULL
6	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190406	NULL
7	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190407	NULL
8	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190408	NULL
9	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190409	NULL
10	NULL	NULL	2019/2020	NULL	2019 Q1	NULL	2019-01	NULL	20190410	NULL

etc.
etc.

##TempStage after initial lookup values are computed:

I	Top_Node	Top_Node_Id	Fiscal_Year	Fiscal_Year_Id	Fiscal_Yr_Qtr	Fiscal_Yr_Qtr_Id	Accounting_Period	Accounting_Period_Id	Date_Dim_Key	Date_Dim_Key_Id
1	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190401	1
2	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190402	2
3	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190403	3
4	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190404	4
5	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190405	5
6	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190406	6
7	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190407	7
8	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190408	8
9	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190409	9
10	Top_Node	1	2019/2020	2	2019 Q1	9	2019-01	1	20190410	10

etc.
etc.

##TempStage after final lookup values are computed:

I	Top_Node	Top_Node_Id	Fiscal_Year	Fiscal_Year_Id	Fiscal_Yr_Qtr	Fiscal_Yr_Qtr_Id	Accounting_Period	Accounting_Period_Id	Date_Dim_Key	Date_Dim_Key_Id
1	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190401	65
2	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190402	66
3	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190403	67
4	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190404	68
5	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190405	69
6	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190406	70

7	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190407	71
8	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190408	72
9	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190409	73
10	Top_Node	1	2019/2020	2	2019 Q1	15	2019-01	17	20190410	74

etc.
etc.

Tree (see above listing for more detail):

NodeId	Node	ParentId	NumChildren	NumDescendants	NumLeaves	Height	Depth
1	Top_Node	NULL	3	1159	1096	4	0
2	2019/2020	1	4	386	366	3	1
3	2020/2021	1	4	385	365	3	1
4	2021/2022	1	4	385	365	3	1
5	2020 Q2	3	4	96	92	2	2
6	2021 Q3	4	4	96	92	2	2
7	2020 Q3	3	4	96	92	2	2
8	2021 Q1	4	4	95	91	2	2
9	2019 Q2	2	4	96	92	2	2
10	2020 Q4	3	4	94	90	2	2

etc.
etc.

Resources

The resource section contains a print-ready copy of this article, along with scripts to install the procs, build the D_DATE table, and run the demo.

References

[1] “Verifying Trees”, www.sqlservercentral.com, 20 Jan 2023 (https://www.sqlservercentral.com/articles/verifying-trees)