Writing a Parent-Child Query Sorter in ColdFusion

by Rick Osborne, http://rickosborne.org/

This document was originally published as a series of blog entries on my web site. The first page, which includes an index for all of the related parts, is available here:

http://rickosborne.org/blog/?p=3

Part 1: Goals

One of the Stupid-Web-Programming-Tricks I end up recoding quite often is a sorter for queries with Parent-Child relationships. Trees, basically. You know the situation: some yahoo wants theoretically infinite levels of depth to his navigational menus or his product categories for his e-commerce system. What you end up with is a table in a database that looks like this:

Stuff		
ItemID	**Title**	**ParentID**
12	Coding	0
26	Web	12
15	Applications	12
9	Hybrid	12
7	Food	0
8	C++	15
19	Perl	9
16	ColdFusion	26
1	Cheese	7

This should magically yield a list that looks like this:

1. Coding
    A. Applications
        a. C++
    B. Hybrid
        a. Perl
    C. Web
        a. ColdFusion
2. Food
    A. Cheese

Getting from Point A to Point B isn't all that hard, especially when you have the time, foresight, and ability to plan ahead. There are dozens of different ways to accomplish this efficiently within the database. Most of the time, however, I find that I have read-only access to the database table and end up having to do the organization in code. In this document I'll be explaining my solution and how I get to it.

Part 2: Data Extraction

The first thing we need to worry about is how to get our data out of the database. A novice would probably start off with something like this:

```
SELECT ItemID, Name, ParentID
FROM Stuff
ORDER BY ParentID, ItemID
```

This makes the assumption that all of the items were entered in the order that they should come back out. This is very rarely the case. Similarly, this won't work:

```
SELECT ItemID, Name, ParentID
FROM Stuff
ORDER BY ParentID, Name
```

All that gains us is that we now know that we're almost in alphabetical order. Somewhat. But really, our same problem from last time comes up: we just can't trust the IDs to be in any sort of order. We really need to be ordering by the Name.

A more SQL-savvy coder might then try this:

```
SELECT s1.ItemID, s1.Name, s1.ParentID
FROM Stuff AS s1 LEFT OUTER JOIN STUFF AS s2 ON (s1.ParentID =
s2.ItemID)
ORDER BY s2.Name, s1.Name
```

A join has been added, so that we're now ordering by the Parent's name. This will work just fine ... if we're limited to only one level of parent-child relationships. Given our test data, we know that we're not. Trying to out-guess the user and add more joins is only going to get us into more trouble (and slow down the query).

Trying to do parent-child relationships in straight SQL is next to impossible given the open-ended scenario we've got here. There are solutions if your DB engine supports parent-child queries natively, as well as solutions that use stored procedures. But what if you don't have access to either of those?

For now, let's just get the data out in the order we'd like to see it if it had no parent-child relationships:

```
SELECT ItemID, Name, ParentID
FROM Stuff
ORDER BY Name
```

In the next section we'll look at how we can re-sort the query with ColdFusion.

Part 3: Starting With ColdFusion

Last time, we gave up on trying to get our data out of the database in a structured way and settled for just getting it out sorted alphabetically. Now we'll get into the juicy stuff: doing the sort in ColdFusion.

Before we start on the code, let's stop and think about what we're doing. Obviously, we need a place to start. We know from our test data that the first thing on our list should be "Coding". But how can we make the code see that?

Wrong Answer #1: *Look for anything with a ParentID of "0".* What makes "0" special? We could assume that the tree always starts at "0". But then we're screwed if someone doesn't start their list with "0". And what if instead of starting the list off with "0", they set the ParentID to NULL?

Wrong Answer #2: *Look for anything with a ParentID of NULL, or "0", or the lowest ParentID we can find.* This is a slightly more convoluted version of the first wrong answer. Sure, you're likely to catch more lists this way, but are you really gaining anything? We already know that we can't trust IDs for ordering, so we really shouldn't use the lowest ParentID, either.

So what are we left with?

Best Guess: *Start with items that have parents that we can't find.* Logically, there should be at least one item in the list with a ParentID that we can't match to an ItemID. So, if we go though the list and find any items that have parents that we can't find,

they should be top-level items. (I've also seen lists that set the root item's ParentID to its own ItemID, but we can special-case that.)

Let's get some of this in code. Assume that we've already run the query and called it "Stuff". We'll start off by looping over the query to build an index of where (which row number) each item (by ID) is within the query. We can then find the root elements by seeing where the holes are in that index.

```
<cfset RowFromID=StructNew()>
<cfloop query="Stuff">
    <cfset RowFromID[ItemID]=CurrentRow>
</cfloop>
<cfset RootItems=ArrayNew(1)>
<cfloop query="Stuff">
    <cfif NOT StructKeyExists(RowFromID, ParentID)>
       <cfset ArrayAppend(RootItems, ItemID)>
    </cfif>
</cfloop>
```

In our `RootItems` array, we now have all of the ItemIDs for the items for which we can't find the parent. Since our query was sorted alphabetically when we ran it, the array is already in the right order.

Next, we'll see what we can do with this.

Part 4: Brute Force

After our first attempt, we're left with two things: an index of `ItemID`s to Rows (`RowFromID`), and an array of the items that don't have parents (within the chunk of the tree that we have) and are therefore logically root nodes (`RootItems`). Where do we go from here?

Let's look at the Brute Force approach. We could loop through the query for each root node and build a Struct tree with the same structure as the result that we want. We could then look for the children of each node recursively by looping over the query to find them. Of course, we'd then have to write a recursive function to walk through our struct tree. Recursive functions are possible in ColdFusion, but they are also rather slow. Also, since we're looping over the entire query for each item, we end up with quadratic slowdown as the query gets larger. A five-item query takes 25 loops, 10 takes 100, 20 takes 400, and so on. For those of you fresh out of CompSci who can still remember your Big O Notation, that's $O(n^2)$. Since we started off with just a linear slowdown, $O(n)$, that's hopefully not the right track.

Since the `ParentID`s don't change we could optimize this approach by building

another index, this time relating items to their children. In fact, we can just integrate this with the first loop in the code we already have. Since we know that parents will almost always have multiple children, we use an array to store them by simply appending the child's `ItemID`. Throw that into a struct by `ParentID` and you've got another index:

```
<cfset RowFromID=StructNew()>
<cfset ChildrenFromID=StructNew()>
<cfloop query="Stuff">
    <cfset RowFromID[ItemID]=CurrentRow>
    <cfif NOT StructKeyExists(ChildrenFromID, ParentID)>
        <cfset ChildrenFromID[ParentID]=ArrayNew(1)>
    </cfif>
    <cfset ArrayAppend(ChildrenFromID[ParentID], ItemID)>
</cfloop>
```

Building this index allows us to eliminate that pesky inner loop that we were talking about and get us back to linear time instead of quadratic. Now we can just loop through our root items' children, and their children recursively, and build our tree.

But there's that word again: *recursive*. As a general rule, as a ColdFusion programmer, you'll want to shy away from a solution that looks recursive. It can probably be done, and I've done my fair share of recursive CF, but odds are that there's a better answer. Sometimes there aren't better answers, but 99% of the time it's worth the extra effort to try to find one.

In this situation, unraveling the recursion turns out to be surprisingly easy. That's what we'll look at next.

Part 5: Unraveling the Recursion

Our second attempt left us with two indexes (items to rows and parents to children) and an array of the root nodes for the tree. Our next task is to figure out how we can *efficiently* take these things and build a tree out of them. I emphasize *efficiently*, because we've already discussed the perils of the most obvious[1] solution: recursion. Instead, we'll do our best to implement the solution iteratively.

We know we need a way to list an item, its first child, it's first child's child, et cetera, in a depth-first approach. That is, we want to go down as many levels as we can before listing a second item. As we finish a level, we need to pop back up to the previous level and move on to the second, then the third, and so on. I can hear the

1 Well, I presume it would be the most obvious solution to the folks who started off on classic procedural languages. For those of you who are cutting your programming teeth with ColdFusion ... move along, nothing to see here.

CompSci crowd saying "Aha!" at that one. Yep, we'll need a stack. For the non-CompSci among you, we need a place to keep track of all the items we still need to work on. Luckily, ColdFusion arrays can provide exactly what we need with the `ArrayAppend`, `ArrayPrepend`, and `ArrayDeleteAt` functions.

Even better, we already have an array with a list of items we need to work on: `RootItems`. But here's where we need to make a decision. We can either treat the end or the beginning of the array as the top of the stack. While it may seem more logical to use the end of the array, it'll make the code profoundly easier to read if we use the beginning. You'll see why later, but this works out well since the first item on our stack is the first item we want to work with. Enough talk, let's code.

First we'll need to pop the current item off the top of the stack:

```
<cfloop condition="ArrayLen(RootItems) GT 0">
    <cfset ThisID=RootItems[1]>
    <cfset ArrayDeleteAt(RootItems, 1)>
```

For now, let's just output the name of the item. We'll change this to do something more interesting later. Of course, we may be looking at a parent node, so we need to make sure the item exists before we try to display it.

```
    <cfif StructKeyExists(RowFromID, ThisID)>
        <cfset RowID=RowFromID[ThisID]>
        <cfoutput>#Stuff.Name[RowID]#<br /></cfoutput>
    </cfif>
```

Look to see if we have any children:

```
    <cfif StructKeyExists(ChildrenFromID, ThisID)>
        <cfset ChildrenIDs=ChildrenFromID[ThisID]>
```

Then push any children onto the stack. We need to do this backwards so that the first child ends up on the top of the stack. Alternatively, we could have pushed them into the index backwards to begin with, but it's pretty much the same thing, and this looks more deliberate.

```
        <cfloop from="#ArrayLen(ChildrenIDs)#" to="1" step="-1"
index="i">
            <cfset ArrayPrepend(RootItems, ChildrenIDs[i])>
```

That's it. Close it up and finish it off:

```
        </cfloop>
    </cfif>
</cfloop>
```

That'll do for the basics of what we want. We should now have a list that is in the correct order, and since we should (hopefully) hit each point in the tree only once we're still at linear, `O(n)`, time. Of course, there's still quite a bit to add to make it sturdier and prettier, but we'll save that for the next part.

Part 6: Adding Depth

Even though we now have our list in the correct order, it's not yet pretty. We have a list, but we really don't have a tree, do we? We need to associate some kind of depth information with each level in the tree. There are a couple of ways we could do this, but the easiest way is to just keep another, parallel, stack with depth information. Going back to our second loop, the one where we figure out which items are the root items and push them onto the stack:

```
<cfset RootItems=ArrayNew(1)>
<cfset Depth=ArrayNew(1)>
<cfloop query="Stuff">
    <cfif NOT StructKeyExists(RowFromID, ParentID)>
        <cfset ArrayAppend(RootItems, ItemID)>
        <cfset ArrayAppend(Depth, 0)>
    </cfif>
</cfloop>
```

Why use `0` for the initial depth? We know that we'll be incrementing the depth for each level of children, so the first displayed children will have a depth of `1`, et cetera. So, starting at `0` seems pretty logical. Of course, if you want to start at `42` or something else, you are welcome to.

We then just need to keep our depth stack synchronized with the item stack in the third loop:

```
    <cfset ThisID=RootItems[1]>
    <cfset ArrayDeleteAt(RootItems, 1)>
    <cfset ThisDepth=Depth[1]>
    <cfset ArrayDeleteAt(Depth, 1)>
```

And again later on when we push the children onto the stack:

```
            <cfset ArrayPrepend(RootItems, ChildrenIDs[i])>
            <cfset ArrayPrepend(Depth, ThisDepth + 1)>
```

This allows us to do tricks like this back up in the display portion of our loop:

```
        <cfoutput>#RepeatString("–",ThisDepth)##Stuff.Name[RowID]#<br
/></cfoutput>
```

You can add in non-breaking spaces, spacer images, empty table cells, or whatever you want instead of the – part to make your display come out right. If you want to be really fancy and generate correctly-nested HTML lists with the data … you're going to have to wait for a later installment in this series. We'll get there, but it's trickier.

As for the additional robustness, we'll work on that next.

Part 7: Building a Function

Let's start looking at turning what we've done into a custom tag and UDF. I'm actually going to start with the UDF and work backwards to an old-school custom tag. I find that the structure imposed by a proper UDF helps me think more about what is going on, and then I can de-structure it later for deployment on older servers. For simplicity's sake, my UDF will be named `QueryTreeSort`.

Now let's talk about component design. What do we want out of this component? There are a few ways to come at a component like this:

1. Return a new query, which is already correctly sorted and has all of the original query's data plus a new column with depth information.
2. Return a sorted list of the row numbers that the calling code can then loop over.
3. Return a sorted list of the row numbers *and* a list of depth measurements.

The first option is the more complete, black-box way to do it. However, for large datasets it may not be efficient to make a copy of them just to add a column for depth information. The second option is more memory-friendly, but the calling code is then responsible for figuring out depth information via the `ParentID` data. The third option returns both order and depth information, but is a bit trickier to work into a UDF since you are returning multiple distinct chunks of data. (You would most likely return a struct of arrays, as does `REFindNoCase`.) I'm going to just work on the first option for now, leaving the other two as exercises for the reader.

Our first step is to UDF-ize what we've already written. I've also added in a few comments, as we're starting to grow beyond plainly-obvious.

First we need to start with a basic function signature:

```
<cffunction name="QueryTreeSort" returntype="query" output="No">
    <cfargument name="Stuff" type="query" required="Yes">
    <cfargument name="ParentID" type="string" required="No"
default="ParentID">
    <cfargument name="ItemID" type="string" required="No"
default="ItemID">
    <cfargument name="BaseDepth" type="numeric" required="No"
default="0">
    <cfargument name="DepthName" type="string" required="No"
default="TreeDepth">
```

We start off with the query to be sorted, which is the only required argument. The next two arguments are provided for convenience so that we don't have to make sure that the queries we want to sort conform to some random naming scheme. Instead, we can pass in the name of our Item and Parent ID columns, thus allowing us to sort queries where the IDs are named `ID` or `ItemID` or `Item` or even `TheItemID_int_4bytes_ricko_was_here`.

The `BaseDepth` argument allows us to specify that we'd prefer that the depth numbering start at 1 or -1 or 42 or whatever. Similarly, we can specify the name of the new column with the `TreeDepth` argument. In this way if our query already has a column named `Depth` or `Level` then we can have the depth returned in a new column called `MyTreeDepthInfo` if we wanted.

We have essentially taken out *anything* that might have been hard-coded or assumed, and left it up to the caller to specify. This makes for code later that is a bit more interesting, but it makes the function *disturbingly* portable.

Next we need to declare our local variables. This function is truly black-box, so it should neither assume anything about the outside world, nor reveal anything to the outside about its inner workings.

```
    <cfset var RowFromID=StructNew()>
    <cfset var ChildrenFromID=StructNew()>
    <cfset var RootItems=ArrayNew(1)>
    <cfset var Depth=ArrayNew(1)>
    <cfset var ThisID=0>
    <cfset var ThisDepth=0>
    <cfset var RowID=0>
    <cfset var ChildrenIDs="">
    <cfset var ColName="">
    <cfset var Ret=QueryNew(ListAppend(Stuff.ColumnList,
Arguments.DepthName))>
```

Most of the variables here are simply moved up to the top of the function from where they were before. The only new ones are the last two. We'll need the `ColName`

variable later when we loop over the columns in the query since we assume nothing about the structure of the query. The `Ret` variable is what will be returned, and it is a query with the same columns as what was passed into the function, plus the new depth information.

Next, as the comment says, we set up our indexes as before. This time, however, we've needed to substitute out our hard-coded column names for the dynamic ones passed into the function.

```
    <!--- Set up all of our indexing --->
    <cfloop query="Stuff">
        <cfset RowFromID[Stuff[
Arguments.ItemID][Stuff.CurrentRow]]=CurrentRow>
        <cfif NOT StructKeyExists(ChildrenFromID,
Stuff[Arguments.ParentID][Stuff.CurrentRow])>
            <cfset ChildrenFromID[Stuff[
Arguments.ParentID][Stuff.CurrentRow]]=ArrayNew(1)>
        </cfif>
        <cfset ArrayAppend(ChildrenFromID[Stuff[
Arguments.ParentID][Stuff.CurrentRow]],
Stuff[Arguments.ItemID][Stuff.CurrentRow])>
    </cfloop>
```

The `Stuff[Arguments.ItemID][Stuff.CurrentRow]` stuff is just nasty-looking, isn't it? But if you pick it apart it's not all that bad. We're looking into the `Stuff` query, in the column whose name was specified by our `ItemID` argument, for the data in the current row. Long-winded and ugly, but really no big deal.

Finding the parents requires similar find-and-replace gymnastics:

```
    <!--- Find parents without rows --->
    <cfloop query="Stuff">
        <cfif NOT StructKeyExists(RowFromID,
Stuff[Arguments.ParentID][Stuff.CurrentRow])>
            <cfset ArrayAppend(RootItems,
Stuff[Arguments.ItemID][Stuff.CurrentRow])>
            <cfset ArrayAppend(Depth, Arguments.BaseDepth)>
        </cfif>
    </cfloop>
```

Second verse, same as the first. And since we're not going to loop over the query again, no more nasty-looking code. Up next is the sorting part, the preamble to which is unchanged.

```
    <!--- Do the deed --->
    <cfloop condition="ArrayLen(RootItems) GT 0">
        <cfset ThisID=RootItems[1]>
```

```
        <cfset ArrayDeleteAt(RootItems, 1)>
        <cfset ThisDepth=Depth[1]>
        <cfset ArrayDeleteAt(Depth, 1)>
```

Look! New code:

```
        <cfif StructKeyExists(RowFromID, ThisID)>
            <!--- Add this row to the query --->
            <cfset RowID=RowFromID[ThisID]>
            <cfset QueryAddRow(Ret)>
            <cfset QuerySetCell(Ret, Arguments.DepthName,
ThisDepth)>
            <cfloop list="#Stuff.ColumnList#" index="ColName">
                <cfset QuerySetCell(Ret, ColName,
Stuff[ColName][RowID])>
            </cfloop>
        </cfif>
```

Okay, so it's not all that exciting. We add a row to the return query, set the depth, and then loop over the original query's columns to copy the data to our return query.

The rest is unchanged, except that we also end our function, returning our resultant query.

```
        <cfif StructKeyExists(ChildrenFromID, ThisID)>
            <!--- Push children into the stack --->
            <cfset ChildrenIDs=ChildrenFromID[ThisID]>
            <cfloop from="#ArrayLen(ChildrenIDs)#" to="1" step="-1"
index="i">
                <cfset ArrayPrepend(RootItems, ChildrenIDs[i])>
                <cfset ArrayPrepend(Depth, ThisDepth + 1)>
            </cfloop>
        </cfif>
    </cfloop>
    <cfreturn Ret>
</cffunction>
```

The following code can be used to test the function, presuming the table data we started with:

```
<cfquery datasource="#Request.DSN#" name="Stuff">
SELECT ItemID, Name, ParentID
FROM Stuff
ORDER BY Name
</cfquery>
<cfset TreeStuff=QueryTreeSort(Stuff)>
```

Dump out `TreeStuff` and you get:

ItemID	Name	ParentID	TreeDepth
12	Coding	0	0
15	Applications	12	1
8	C++	15	2
9	Hybrid	12	1
19	Perl	9	2
26	Web	12	1
16	ColdFusion	26	2
7	Food	0	0
1	Cheese	7	1

As you can see, we now have a function that can take a query, tree-sort it, and return a new query that is not only in the right order, but also includes depth information to help us figure out how to format it. In the next section we'll do the fun stuff: formatting.

Part 8: Table-Based Display

In this installment we're going to write a function to turn our tree-sorted query into an HTML table. A function may not be the best place to do this, as your presentation is very likely tied to your data. However, for simplicity let's assume that you just want a function to dump the data in a table-based tree-like format, and you can take that function and tweak it to your specific needs.

Let's dive right in with our function signature, shall we?

```
<cffunction name="TableFromQueryTree" returntype="string"
output="No">
    <cfargument name="Query" type="query" required="Yes">
    <cfargument name="Columns" type="array" required="Yes">
    <cfargument name="Titles" type="array" required="Yes">
    <cfargument name="Classes" type="array" required="No">
    <cfargument name="SpannedColumn" type="string" required="No">
    <cfargument name="ParentID" type="string" required="No"
default="ParentID">
    <cfargument name="ItemID" type="string" required="No"
default="ItemID">
    <cfargument name="BaseDepth" type="numeric" required="No"
default="0">
    <cfargument name="DepthName" type="string" required="No"
default="TreeDepth">
    <cfargument name="EvenOdd" type="array" required="No"
default="#ListToArray('even,odd')#">
```

Many of these arguments are the same as last time. First, we pass in the query to

be displayed. The next 3 arguments are arrays that tell the function which columns to display and in what order, what to use for the column titles, and what classes to assign to the cells for each of the columns. This is a half-hearted attempt to abstract the function a little bit from the presentation details. The CSS classes will come in handy later for a bit of post-processing.

The `SpannedColumn` argument is the name of the column that will have the tree formatting assigned to it, and will be pushed to the right depending on the depth of the row. The rest are copied straight from the previous function up to `EvenOdd`. I prefer my tables to be zebra-striped, so passing in the class names that you use for even and odd rows will save time later.

Next, we declare our local variables:

```
    <cfset var MinDepth=999>
    <cfset var MaxDepth=0>
    <cfset var Ret="">
    <cfset var Levels=0>
    <cfset var Q=Arguments.Query>
    <cfset var ColID=0>
```

Because we can start the tree depth with any level, we'll need to calculate the minimum and maximum depths in the query so that we can tell what the range of depths are. Since we'll be using the passed-in query so much, I found the code easier to read when I copied a reference of it to `Q`. We'll use `ColID` as a loop variable.

```
    <cfloop query="Q">
        <cfif Q[Arguments.DepthName][Q.CurrentRow] GT MaxDepth>
            <cfset MaxDepth=Q[Arguments.DepthName][Q.CurrentRow]>
        </cfif>
        <cfif Q[Arguments.DepthName][Q.CurrentRow] LT MinDepth>
            <cfset MinDepth=Q[Arguments.DepthName][Q.CurrentRow]>
        </cfif>
    </cfloop>
    <cfset Levels=MaxDepth-MinDepth+1>
```

The table header is fairly straightforward:

```
    <cfsavecontent variable="Ret">
        <cfoutput>
<table>
    <thead>
        <tr>
            <cfloop from="1" to="#ArrayLen(Columns)#" index="ColID">
            <th<cfif Classes[ColID] NEQ "">
class="#Classes[ColID]#"</cfif><cfif (Columns[ColID] EQ
```

```
Arguments.SpannedColumn) AND (Levels GT 1)>
colspan="#Levels#"</cfif>><cfif Titles[ColID] NEQ
"">#Titles[ColID]#<cfelse>#Columns[ColID]#</cfif></th>
            </cfloop>
        </tr>
    </thead>
```

We're going to pass back a string instead of outputting directly, hence the `cfsavecontent` tag. We loop over the column array that was passed in and output any class information that we might have been given. That next `cfif` block sets up the span if the current column is our tree column and is supposed to be spanned. Last, we try to output a column title if one was given, but output the SQL column name if not.

The body is a bit more convoluted, but not too much worse:

```
    <tbody>
            <cfloop query="Q">
        <tr class="#Arguments.EvenOdd[IncrementValue(Q.CurrentRow MOD
2)]#">
                <cfloop from="1" to="#ArrayLen(Columns)#"
index="ColID">
                    <cfif (Columns[ColID] EQ
Arguments.SpannedColumn)><cfloop from="#IncrementValue(MinDepth)#"
to="#Q[Arguments.DepthName][Q.CurrentRow]#"
index="i"><td>   </td></cfloop></cfif>
            <td<cfif Classes[ColID] NEQ "">
class="#Classes[ColID]#"</cfif><cfif (Columns[ColID] EQ
Arguments.SpannedColumn) AND (Int(MaxDepth -
Q[Arguments.DepthName][Q.CurrentRow]) GT 0)>
colspan="#IncrementValue(MaxDepth-
Q[Arguments.DepthName][Q.CurrentRow])#"</cfif>>#HTMLEditFormat(Q[Colu
mns[ColID]][Q.CurrentRow])#</td>
                </cfloop>
        </tr>
            </cfloop>
    </tbody>
```

Of course, we start by looping over the query, then again over each column. In between we have that even/odd magic that I mentioned earlier. The next `cfif` block inside the inner loop is essentially the same as before: if we're looking at the tree-spanned column and the current depth is greater than the minimum depth, then output as many padding columns as needed. The next `cfif` block is the same class-generating stuff we saw before.

The third `cfif` block is a bit less obvious. Yet again, we're only interested if the current column is the spanned one. If so, and the current depth is *less* than the

maximum, then we need to generate a `colspan` attribute to make sure this cell fills all the space it needs to. After that, output the data and close off the loops.

The rest of the function just closes out all of the HTML, loops, etc, that need to be closed, and returns our data.

```
</table>
        </cfoutput>
    </cfsavecontent>
    <cfreturn Ret>
</cffunction>
```

That's it. We now have a function to dump tree-based tables. Let's call it:

```
<cfoutput>#
TableFromQueryTree(
    TreeStuff,
    ListToArray("ItemID,Name,ParentID,TreeDepth"),
    ListToArray("ID,Name,Parent,Depth"),
    ListToArray("num,string,num,num"),
    "Name"
)#</cfoutput>
```

Vis:

ID	Name	Parent	Depth
12	Coding	0	0
15	Applications	12	1
8	C++	15	2
9	Hybrid	12	1
19	Perl	9	2
26	Web	12	1
16	ColdFusion	26	2
7	Food	0	0
1	Cheese	7	1

Of course, I sprinkled in some extra formatting *after* the function returned by creative use of `ReplaceNoCase` and the class attributes that I provided.

After all of the acrobatics to get padded tables, you would think that doing a correctly-nested HTML list would be cake, right? Wrong. Nested lists are actually a bit trickier, and I'll be going over them in the next section.

Part 9: List-Based Display

Finally we get to the endgame: formatting a tree-sorted query as an HTML list using ColdFusion. We'll build upon what we've already gone through, of course. This time we'll be building `ListFromQueryTree`, a UDF that takes a tree-sorted query and returns a correctly-nested HTML list.

It starts, of course, with a signature:

```
<cffunction name="ListFromQueryTree" returntype="string" output="No">
    <cfargument name="Query" type="query" required="Yes">
    <cfargument name="TitleColumn" type="string" required="No"
default="Name">
    <cfargument name="DepthColumn" type="string" required="No"
default="TreeDepth">
    <cfargument name="DepthPrefix" type="string" required="No"
default="depth">
    <cfargument name="ListTag" type="string" required="No"
default="ul">
```

Nothing in here should be surprising. We start with the query, then pass in the names of the columns that are the title and the depth information. We'll also pass in a prefix to be used in our `ul` tags as CSS classes that help select sub-level lists of a certain depth. Lastly, we have the option of overriding the default list tag, `ul`, and using something else, such as `ol` or whatever you like.

Local variable declarations come next:

```
    <cfset var Ret="">
    <cfset var MinDepth=999>
    <cfset var Q=Arguments.Query>
    <cfset var LastDepth=0>
    <cfset var ThisDepth=0>
    <cfset var d=0>
```

Again, there are no surprises here. We'll use `d` as a loop variable later on. Then we do our loop to find the minimum depth:

```
    <cfloop query="Q">
        <cfset ThisDepth=Q[Arguments.DepthColumn][Q.CurrentRow]>
        <cfif ThisDepth LT MinDepth>
            <cfset MinDepth=ThisDepth>
        </cfif>
    </cfloop>
    <cfset LastDepth=MinDepth-1>
```

We also set `LastDepth` to be one less that the minimum depth. Why?

Since we can't cheat like we did with tables and just pad to the left the number of empty spaces, we'll need to keep track of how deep we are in the list, so that we can then pop up or down the appropriate number of levels before and after each list item. In theory, we should never dig down more than one level at a time, but we may pop back up more than level at a time. If we're on the last item of some deeply-nested list, once it ends we'll need to close off the number of lists that are between that item and the next. By setting `LastDepth` to one less than the minimum, we know that we'll always have cleanly-closed lists.

Then the main loop begins:

```
    <cfloop query="Q">
        <cfset ThisDepth=Q[Arguments.DepthColumn][Q.CurrentRow]>
```

We'll use `ThisDepth` as a shortcut for the depth of the current item. Using that longer version each time is just silly.

```
        <cfif LastDepth LT ThisDepth>
            <cfloop from="#IncrementValue(LastDepth)#"
to="#ThisDepth#" index="d">
                <cfset Ret=Ret & '<#Arguments.ListTag#
class="#Arguments.DepthPrefix##d#">'>
            </cfloop>
        <cfelse>
            <cfif LastDepth GT ThisDepth>
                <cfset Ret=Ret &
RepeatString("</li></ul>",LastDepth-ThisDepth)>
            </cfif>
            <cfset Ret=Ret & "</li>">
        </cfif>
```

This is the meat of our function. We need to look at the depth of the current item as compared to the depth of the item before it. If it's greater then we've dug down a bit and need to open a new list. Otherwise, it could be at the same level or upwards. If we've moved upwards, the last depth being greater than the current depth, then we need to close off the lists in between. Either way, we'll also need to close off the previous item. In the first scenario, digging down, we *didn't* close the current item, because it'll need to be closed after all of its children are closed.

The rest is cake:

```
        <cfset Ret=Ret & "<li>" &
HTMLEditFormat(Q[Arguments.TitleColumn][Q.CurrentRow])>
        <cfset LastDepth=ThisDepth>
    </cfloop>
```

We output the current item, but *do not* close it, as we know it will be closed in a later iteration of the loop. But what about the dangling items at the end of the loop, you ask?

```
    <cfif Q.RecordCount GT 0>
        <cfset Ret=Ret & RepeatString("</li></ul>",LastDepth-
(MinDepth-1))>
    </cfif>
    <cfreturn Ret>
</cffunction>
```

We close them in one fell swoop at the end. Notice that we use `MinDepth` here, as that's the depth we started with.

Calling our function is similar to the table version:

```
<cfoutput>#ListFromQueryTree(TreeStuff)#</cfoutput>
```

Yields:

- Coding
    - Applications
        - C++
    - Hybrid
        - Perl
    - Web
        - ColdFusion
- Food
    - Cheese

Again, I added a bit of color at the end by replacing the CSS classes with inline styles with `ReplaceNoCase`.