

# Relational Data Modeling With Recursive Constructs: Part 2.

Dr. Tom Johnston  
MindfulData.com

By definition, any entity which has a relationship to itself is a recursive entity; and so any table implementing such an entity is a recursive table. In this chapter, specifically, I will describe the following kinds of recursive table constructs ("RT" constructs, for short):<sup>1</sup>

1. *Sequentially recursive tables*, used to model a 1:1 relationship among rows in a table, resulting in a linearly-structured set of rows;
2. *Hierarchically recursive tables*, used to model a 1:M relationship among rows in a table, resulting in a tree-structured set of rows; and
3. *Matrix-like recursive tables*, used to model a M:M relationship among rows in a table, resulting in a network-structured set of rows.

## **Hierarchically Recursive Tables: Modeling Tree-Structured Sets.**

A linear sequence is a pretty simple constraint. It's simple because it's strict. It is the constraint that (a) every code but the first has one (and only one) predecessor, (b) every code but the last has one (and only one) successor, (c) the first code has no predecessor, and (d) the last code has no successor.

A more complex form of constraint can be obtained by dropping requirement (a) or (b). It turns out that it doesn't matter which, so let it be (b).

This means that our constraint is relaxed so that every location type code but the last has one *or more* successors. Now as soon as we permit multiple successors, the notion of a last code becomes meaningless. For suppose we have one code with three successor codes. Which of the three is the *last* one? We could arbitrarily designate, for example, that the one in the lower right-most position on the illustration is the last one. But this is just an interpretation we place on an illustration. There is no foundation in the structure of the data itself that corresponds to this arbitrary convention.

A sequential RT construct does support the semantics of a set having a first and last element. It does by incorporating a syntactic structure that supports the semantics of a linear sequence. As we have just seen, that structure may be either (a) a predecessor and successor for each code, or (b) a sequence number for each code. But this more "relaxed" structure—a hierarchical RT construct--does not support the concept of a last element. It

---

<sup>1</sup> In this chapter, I will use the terminology of "tables" and "rows" as synonymous with the more awkward terminology of "entities" and "entity occurrences", or "entity instances".

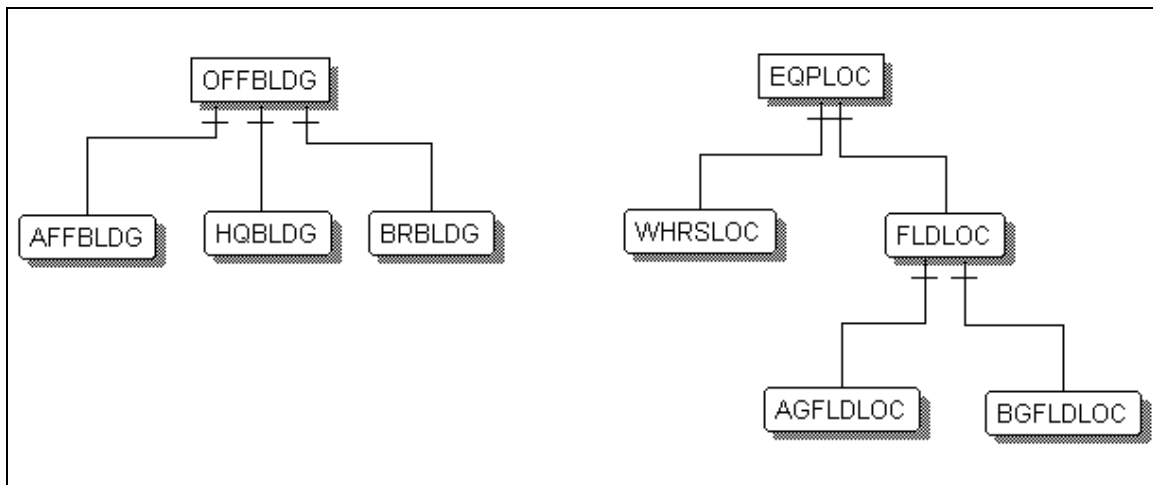
does, however, support the concept of a first element. As we shall see, the first element of a hierarchy is the “root node” of a tree structure.

The original sequential structure is like a hierarchy which isn’t allowed to branch out. This more relaxed structure is more relaxed because it’s a sequence in which branching is permitted.

It would be nice if we could use this insight into how the two structures differ to determine how we should model a tree structure. And, in fact, we can. But first, we must return, not to our sequence number approach to linear code tables, but to our predecessor/successor approach.

The semantic difference between a linear sequence and a hierarchy is precisely the difference between our original definition of a linear sequence and the definition of a hierarchy which was obtained by removing constraint (b) above from that definition. And to preserve the one-to-one correspondence between a semantic requirement and a syntactic, or structural, implementation of the requirement, this change in the semantics of what we are required to model should be reflected in a modification to the structure of our table. Therefore, since the semantic constraint which was removed was the constraint of having at most one successor, we can reflect that change by removing the syntactical construct which expressed that constraint. That construct is the loc-type-succ-eid column itself, of Figures 4.2 and 4.3.

But before we remove the successor column itself, let’s describe a hierarchical relationship among our codes, so we have something to model. So let us suppose that that hierarchical relationship among our location codes is as follows:



**Figure 4.8. A Set of Hierarchically Related Codes: Two Trees.**

With the successor column removed from our sequential RT construct, we still have the predecessor column left. This column will enforce the constraint that every code can have at most one predecessor. But without a successor column, there is no syntactical construct

to enforce the rule that every code can have at most one successor, and so any code can have as many successors as it wants.

The result of these modifications is the hierarchical RT construct, shown in Figure 4.9.

**LOCATION TYPES**

<i>loc-type-<u>eid</u></i>	<i>loc-type-<u>pred-<u>eid</u></u></i>	<i>loc-type-code</i>	<i>loc-type-desc</i>
6021	{null}	OFFBLDG	Office Building
2718	6021	AFFBLDG	Affiliate Building
3234	6021	HQBLDG	Headquarters
4440	6021	BRBLDG	Branch Office Building
2356	{null}	EQPLOC	Equipment Location
0582	2356	WRHSLOC	Equipment Warehouse
5678	2356	FLDLOC	Equipment-in-Field Location
1448	5678	AGFLDLOC	Equipment-in-Field Location, Above-Ground
3899	5678	BGFLDLOC	Equipment-in-Field Location, Below-Ground

**Figure 4.9. A Hierarchical RT Construct--Sample Extension Showing Two Trees.**

Note that location type code #6021 has three successors, and codes #2356 and 5678 have two successors each. This is shown in the illustration in Figure 4.8, and modeled as shown in the hierarchical RT construct illustrated in Figure 4.9.

We can read this table either "up" or "down"--up from a leaf node to its root node, or down from a root node to its leaf nodes. Figure 4.10 illustrates how the RT construct can be read *up*.

We'll start with the below-ground equipment node (#3899). We can see that this is a leaf node because it is not the predecessor of any other node, since it does not appear anywhere in the loc-type-pred-eid column. Next, arrows #1 and #2 take us to the equipment-in-field location--the immediate predecessor node (#5678) to the below-ground equipment node (#3899).

Finally, as the last step in tracing up from a leaf node to a root node, arrows #3 and #4 take us to the equipment location--the immediate predecessor node (#2356) to the equipment-in-field node (#5678). And we know that the equipment location is a root node because its loc-type-pred-eid eid is null.

**LOCATION TYPES**

Loc-Type-Eid	Loc-Type-Pred-Eid	Loc-Type-Code	Loc-Type-Desc
058	300	WRHSLOC	Equipment Warehouse
144	885	AGFLDLOC	Equipment-in-Field Location, Above-Ground
300	{null}	EQPLOC	Equipment Location
313	947	AFFBLDG	Affiliate Building
475	885	BGFLDLOC	Equipment-in-Field Location, Below-Ground
706	947	HQBLDG	Headquarters
721	947	BRBLDG	Branch Office Building
885	300	FLDLOC	Equipment-in-Field Location
947	{null}	OFFBLDG	Office Building

**Figure 4.10. A Path Up to a Root Node. {{{{9/25/97: needs to be updated.}}}}**

Figure 4.11 illustrates how the RT construct can be read *down*.

We'll start with the equipment location--the node in the hierarchy with EID 30075. We can see that this is a root node because its loc-type-pred-eid is null. Next, arrows 1a and 1b take us to its two immediate successor nodes--the equipment warehouse location (node #0582) and the equipment-in-field location (node #5678). The equipment warehouse location is a leaf node, and consequently is the end of that branch of the tree; we can see that this is a leaf node because it is not the predecessor of any other node, since it does not appear anywhere in the loc-type-pred-eid column.

However, the equipment-in-field location is *not* a leaf node, since it is the predecessor of two other nodes. Arrows 2a and 2b point to those nodes--the above-ground and below-ground equipment locations (nodes #1448 and #3899).

Finally, both of these last two locations are leaf nodes. As always, what shows that they are leaf nodes is that their EIDs do not appear as predecessor EIDs to any other nodes.

## LOCATION TYPES

Loc-Type-Eid	Loc-Type-Pred-Eid	Loc-Type-Code	Loc-Type-Desc
058	300	WRHSLOC	Equipment Warehouse
144	885	AGFLDLOC	Equipment-in-Field Location, Above-Ground
300	{null}	EQPLOC	Equipment Location
313	947	AFFBLDG	Affiliate Building
475	885	BGFLDLOC	Equipment-in-Field Location, Below-Ground
706	947	HQBLDG	Headquarters
721	947	BRBLDG	Branch Office Building
885	300	FLDLOC	Equipment-in-Field Location
947	{null}	OFFBLDG	Office Building

**Figure 4.11. A Path Down to Leaf Nodes. {{{{9/25/97: needs to be updated.}}}}**

A couple of observations can be made about these structures. The first is that the hierarchies which they represent can be as broad and as deep as you like. Indeed, in most real-world situations, the trees will be considerably larger than the ones used here for illustrations.

The second is that much of the power and value of this structure comes from the fact that it is so simple, stable and mathematically well-defined a structure that processes to navigate it can be provided by vendors, and will not have to be written again and again by IS professionals. Once the root node of a tree is specified, it should be possible to write a query to get all its branch and leaf nodes which is as simple as this:

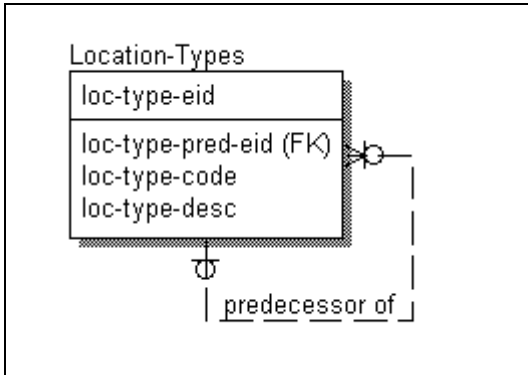
*Select, recursively, all members of the tree whose root node is node EID=x.*

If each row in a recursively-structured table contained counts, amounts or any arithmetically-manipulable attributes, it should be possible to also specify aggregation functions such as summing, counting, taking an average or standard deviation, etc., from the set of rows specified in the recursive query.

The data model diagram for this hierarchically recursive table is shown in Figure 4.12. Observe that there is something different about this 1:M relationship, other than the fact that it relates rows in a table to other rows in the same table. The other difference is that the relationship is optional on both ends.

The relationship, we can see, is from a predecessor row to its successor rows. Now suppose that the relationship was mandatory at the one-end, which is the typical minimum cardinality for a one-to-many relationship. With a recursive relationship, however, this won't do, because it would entail that the table has an infinite number of

rows!<sup>2</sup> For given any row you care to start with, it will have a predecessor row, since all rows must have one. But then its predecessor row will itself have a predecessor row, and so on ad infinitum. A similar line of reasoning easily demonstrates that the relationship must also be optional on the many-end.



**Figure 4.12. A Hierarchical RT Construct--Diagram.**

And the generic DDL for this hierarchically recursive table is shown in Figure 4.13.

```
CREATE TABLE Location-Types
(
  loc-type-eid
  loc-type-code
  loc-type-desc

  PRIMARY KEY (loc-type-eid)
  FOREIGN KEY (loc-type-pred-eid) REFERENCES TABLE
    Location-Types
)
```

**Figure 4.13. A Hierarchical RT Construct--Generic DDL.**

**Multiple Hierarchies: Pruning and Grafting Trees.**

There is one more significant point to make about hierarchically recursive tables. They can contain *multiple* hierarchies--any number of them. For consider the set of rows shown in Figure 4.9. Two of those rows have a {null} value for their predecessor—the rows for Office Buildings and for Equipment Locations. This models the fact that there are actually two distinct hierarchical structures in the table, whose root nodes are these two rows. And Figure 4.8 clearly shows those two distinct hierarchies.

<sup>2</sup> Actually, this conclusion follows only given the (quite reasonable) assumption that no row can be a predecessor to itself.

Obviously, any number of additional trees could be added to this table. To start a new one, we need only to add a row whose loc-type-pred-eid is {null}. To extend it, we need only to add rows whose loc-type-pred-eid is that row's loc-type-eid. We can continue to add new rows to any tree by putting some existing row's identifier into the new row's loc-type-pred-eid.

As just described, multiple trees in a hierarchical RT construct can be combined into one tree with ease. To illustrate, let's combine the two Location Type trees shown in Figures 4.8 and 4.9 into one tree. Let's make the root node of this one tree a node whose code is "LEGLOC", standing for "Legacy Location Types". The meaning of this new single Location Type tree, then, will be that it contains all the location type codes used in the enterprise's legacy systems.<sup>3</sup>

Figure 4.14 is a graphical representation of this new single-tree set of codes.

Insert Figure 4.14 here.

***Figure 4.14. A Set of Hierarchically Related Codes: One Tree.***

Now let's see how easy it is to represent this change in our hierarchical RT construct. Figure 4.15 shows the new row added to the original table from Figure 4.14. At this point, it has been assigned an EID, along with the code and description already mentioned. And, since we already know that it will be the root node of the new tree, we know that its loc-type-pred-eid will be {null}.

---

<sup>3</sup> In a later chapter, we will then add a new set of "standard" location codes, and illustrate how to map legacy codes to canonical "standard" codes.

**LOCATION TYPES**

<i>loc-type-eid</i>	<i>loc-type-pred-eid</i>	<i>loc-type-code</i>	<i>loc-type-desc</i>
0582	2356	WRHSLOC	Equipment Warehouse
1448	5678	AGFLDLOC	Equipment-in-Field Location, Above-Ground
2356	{null}	EQPLOC	Equipment Location
2718	6021	AFFBLDG	Affiliate Building
3899	5678	BGFLDLOC	Equipment-in-Field Location, Below-Ground
6215	{null}	LEGLOG	Legacy Location Types
3234	6021	HQBLDG	Headquarters
4440	6021	BRBLDG	Branch Office Building
5678	2356	FLDLOC	Equipment-in-Field Location
6021	{null}	OFFBLDG	Office Building

**Figure 4.15. A Hierarchical RT Construct--Sample Extension Showing Three Trees.**

But at this point, all we have done is add a third tree to our Location Type table. We can see that we have three trees because we have three rows with {null} as a loc-type-pred-eid value.

The second step is to link the two original trees to this root node. We do so by making the new row's loc-type-eid the loc-type-pred-eid of the root nodes of the original two trees. Doing so results in the single tree shown in Figure 4.16.

**LOCATION TYPES**

<i>loc-type-eid</i>	<i>loc-type-pred-eid</i>	<i>loc-type-code</i>	<i>loc-type-desc</i>
0582	2356	WRHSLOC	Equipment Warehouse
1448	5678	AGFLDLOC	Equipment-in-Field Location, Above-Ground
2356	6215	EQPLOC	Equipment Location
2718	6021	AFFBLDG	Affiliate Building
3899	5678	BGFLDLOC	Equipment-in-Field Location, Below-Ground
6215	{null}	LEGLOG	Legacy Location Types
3234	6021	HQBLDG	Headquarters
4440	6021	BRBLDG	Branch Office Building
5678	2356	FLDLOC	Equipment-in-Field Location
6021	6215	OFFBLDG	Office Building

**Figure 4.16. A Hierarchical RT Construct--Sample Extension Showing One Tree.**

We can now see that we have one tree, because we have only one row with {null} as a loc-type-pred-eid value.

We have just seen how to merge two tree structures into one. It is a simple, two-step process. First, add one row for the new root node. Second, replace the {null} loc-type-pred-eid of the trees to be merged with the loc-type-eid of the new root node.

It is obvious that we can merge any number of tree structures into one structure, in this manner. It is equally obvious that we can “cut” a tree at any branch, creating two or more trees. To make a cut, we need only set the loc-type-pred-eid of the node to be “pruned” from the tree, to {null}. Doing so creates a new tree, whose root node is the node whose loc-type-pred-eid was just set to {null}.

In this way, we can both “prune” and “graft” hierarchical structures. So as a final illustration of these operations on tree structures, let’s suppose that our enterprise has decided to revise their hierarchical model of locations as follows:

1. Introduce a high-level distinction, under Legacy Location Types, between (a) Enterprise Location Types, and (b) Other Location Types.
2. Move all of the other rows in the table under Enterprise Location Types. In other words, prune them from Legacy Location Types, and graft them onto Enterprise Location Types.
3. Create an Office Building location type under Other Location Types.
4. Move AFFBLDG under the Other Locations’ Office Building type.
5. Add a Competitor’s Building entry under the Other Location Types’ Office Building.

Our first step is to add two rows to the Location Type table—ENTPRLOCS and OTHRLOCS. These will represent enterprise location types and other location types, respectively. To add the rows, we assign them an EID, and then enter their name and description. Since both have been defined as legacy location types, we “graft” them onto the tree structure just under the Legacy Locations node, by making that node’s loc-type-eid their loc-type-pred-eid.

Then, to carry out the second step, we find all the rows whose loc-type-pred-eid links them to the LEGLOG row, and change it so that it links them to the new ENTPRLOC row. The result of carrying out these two steps is shown in Figure 4.17. (Changes to the table have been italicized.)

**LOCATION TYPES**

<i>loc-type-<u>eid</u></i>	<i>loc-type-<u>pred-<u>eid</u></u></i>	<i>loc-type-code</i>	<i>loc-type-desc</i>
0582	2356	WRHSLOC	Equipment Warehouse
1448	5678	AGFLDLOC	Equipment-in-Field Location, Above-Ground
8325	6215	<i>OTHRLOCS</i>	<i>Other Location Types</i>
2356	7318	EQPLOC	Equipment Location
2718	6021	AFFBLDG	Affiliate Building
7318	6215	<i>ENTPRLOCS</i>	<i>Enterprise Location Types</i>
3899	5678	BGFLDLOC	Equipment-in-Field Location, Below-Ground
6215	{null}	LEGLOG	Legacy Location Types
3234	6021	HQBLDG	Headquarters
4440	6021	BRBLDG	Branch Office Building
5678	2356	FLDLOC	Equipment-in-Field Location
6021	7318	OFFBLDG	Office Building

**Figure 3.17. A Hierarchical RT Construct—Pruning and Grafting Example, Step 1.**

Our third step is to add an Office Building row. To place it, in the tree structure, under OTHRLOCS, we set its loc-type-pred-*eid* to OTHRLOCS. Since code values must be unique, and since there is already a code with the value OFFBLDG, linked to Enterprise Location Types, we will assign this new one the code value OOFFBLDG, standing for “Other Office Building”.

The fourth step is to change the loc-type-pred-*eid* of AFFBLDG from its current value to the code-type-*eid* for OOFFBLDG. This is *both* a pruning and a grafting operation. By removing the loc-type-*eid* for OFFBLDG from the loc-type-pred-*eid* of AFFBLDG, we are pruning that node from its current place in the tree. Then, by putting the loc-type-*eid* for OOFFBLDG into its loc-type-pred-*eid*, we are grafting AFFBLDG onto that new node of the tree.

The last step is to add a CMPBLDG row, representing competitors’ buildings, and graft it onto the Location tree by setting its loc-type-pred-*eid* to OOFFBLDG.

The results of these last three steps are shown in Fig 4.18. (Again, changes to the table have been italicized.)

**LOCATION TYPES**

<i>loc-type-<u>eid</u></i>	<i>loc-type-<u>pred-<u>eid</u></u></i>	<i>loc-type-code</i>	<i>loc-type-desc</i>
7716	8325	OOFFBLDG	Other Office Building
0582	2356	WRHSLOC	Equipment Warehouse
8381	7716	CMPBLDG	Competitors' Building
1448	5678	AGFLDLOC	Equipment-in-Field Location, Above-Ground
8325	6215	OTHRLOCS	Other Location Types
2356	7318	EQPLOC	Equipment Location
2718	7716	AFFBLDG	Affiliate Building
7318	6215	ENTPRLOCS	Enterprise Location Types
3899	5678	BGFLDLOC	Equipment-in-Field Location, Below-Ground
6215	{null}	LEGLOG	Legacy Location Types
3234	6021	HQBLDG	Headquarters
4440	6021	BRBLDG	Branch Office Building
5678	2356	FLDLOC	Equipment-in-Field Location
6021	7318	OFFBLDG	Office Building

**Figure 4.18. A Hierarchical RT Construct—Pruning and Grafting Example, Step 2.**

Observe that these rows do not exist in any particular sequence in this table. For readability, I have listed them in an approximate top-to-bottom sequence. But that sequence is entirely arbitrary. What is important is that the *loc-type-pred-eid* column fully represents the tree structure. It does so by indicating the “parent” node for each node in the hierarchy but the root node.

This completes our discussion of hierarchical RT constructs. The last, and most complex, RT construct which we will analyze is one in which there is a many-to-many relationship among a set of codes.