

Tables as Trees: Merging with Wildcards using Tree Traversal and Pruning

Matthew Nizol, Hewlett-Packard, Pontiac, MI

Abstract

One of the most fundamental operations in SAS is to merge two data sets. When the values of the key variables involved in the merge are fully specified, the code is straightforward. However, if one of the tables contains wildcards in the key variables, standard merge and look-up techniques do not work. This paper will explore an algorithm for merging two SAS data sets in which the key variables of the master data set are fully specified whereas the key variables of the look-up data set may either be fully specified or wildcard (e.g. "don't care") values. The algorithm works by interpreting the look-up table as a tree structure: each key variable becomes a level in the tree, and blocks of unique values for the key variables become nodes in the tree. As the algorithm visits each level in the tree, the tree is pruned to create a binary tree in which one child holds the wildcard value and the other child holds the fully specified key value. If any paths from the root to the leaves remain after all pruning is complete, the records in the look-up data set represented by those paths are returned as matches. This paper is intended for intermediate to advanced users of SAS. A basic understanding of tree-based data structures will be helpful but not necessary. The code presented in this paper will work with Base SAS versions 9.1.3 and later.

Introduction

As SAS programmers, we all know how to merge two data sets when the key variables are fully specified. But how can we approach a merge if one of the tables contains wildcard values in its key? Consider the following scenario. You are a real estate agent with a list of clients; each client is concerned about certain attributes in selecting a new home and is unconcerned about other attributes. As a real estate agent, you naturally have a listing of houses for sale as well. Samples of these two data sets are below:

Clients data set:

Client	num_beds	ext_color	price_pnt
Carl Clueless	*	*	*
Fred Familyman	2	*	C
Fred Familyman	2	R	B
Fred Familyman	3	*	C
Fred Familyman	3	R	B
H. I. Maintenance	2	R	A
H. I. Maintenance	2	R	B
Monte Moneybags	*	R	*
Monte Moneybags	*	W	*
Sam Saver	*	*	F
Sam Saver	2	*	D

Houses data set:

House	num_beds	ext_color	price_pnt
1	4	G	A
2	2	R	B
3	3	W	C
4	0	B	F
5	3	R	C

price_pnt

F = under \$75k
 E = \$75-\$100k
 D = \$100-\$125k
 C = \$125-\$200k
 B = \$200-\$300k
 A = over \$300k

ext_color

B = Blue
 G = Green
 R = Red
 W = White

Inspecting the Clients data set, we see that Carl Clueless has no particular requirements in mind, and would like to see every house available. Fred Familyman is on a budget, however – he has a specific price point in mind and needs a certain number of bedrooms to house his family; he's willing to pay a premium if the house is his wife's favorite color (red). H. I. Maintenance has very specific requirements and is willing to pay for them. For Monte Moneybags, price is no object, as long as the house is red or white. Finally, Sam Saver's primary concern is low price – if he can get a house in the F price point he doesn't care about any other attributes, but he gets a little pickier if he's going to move up to the next price point.

Given the list of available houses and the list of client preferences, how can we match our clients with houses that they might be interested in? Basic look-up techniques such as data step merges, formats, and hash look-ups will fail due to

the wild cards. Let's begin by looking at a straightforward solution using PROC SQL. This will serve as our baseline analysis of the problem against which we can benchmark our final algorithm.

A Simple SQL Approach

In a typical SQL join, we would use an ON expression in which the corresponding variables between the two data sets to be merged are compared for equality. However, PROC SQL is an extremely powerful tool for merging data, and the ON expression used for a join can be tailored to perform additional comparisons for more customized joins. In our present scenario, we can write an ON expression in which the variables from the Clients data set are checked for equality with either the corresponding variables in the Houses data set or with a wild card. Consider the following SAS Code:

```
proc sql;
  create table test_sql as
  select clients.client, houses.*
  from clients as C inner join houses as H
  on (C.price_pnt = H.price_pnt OR C.price_pnt = '*' ) and
     (C.num_beds = H.num_beds OR C.num_beds = '*' ) and
     (C.ext_color = H.ext_color OR C.ext_color = '*' )
  order by client, house;
quit;
```

Result:

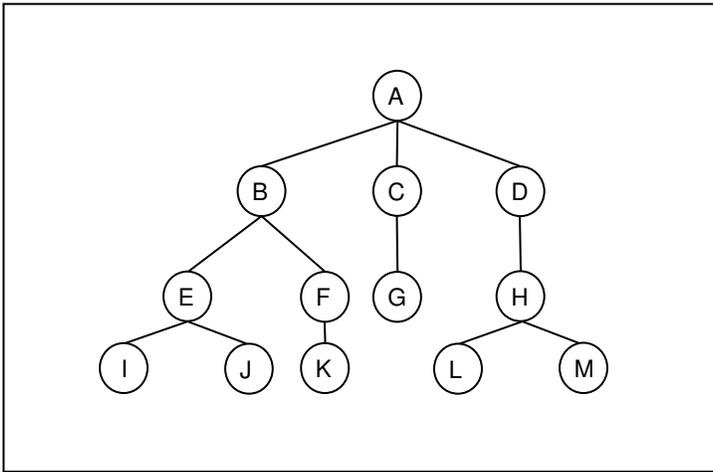
client	house	num_beds	ext_color	price_pnt
Carl Clueless	1	4	G	A
Carl Clueless	2	2	R	B
Carl Clueless	3	3	W	C
Carl Clueless	4	0	B	F
Carl Clueless	5	3	R	C
Fred Familyman	2	2	R	B
Fred Familyman	3	3	W	C
Fred Familyman	5	3	R	C
H. I. Maintenance	2	2	R	B
Monte Moneybags	2	2	R	B
Monte Moneybags	3	3	W	C
Monte Moneybags	5	3	R	C
Sam Saver	4	0	B	F

PROC SQL has successfully merged the data sets, providing us with a list of houses for each client that matches that client's preferences. This certainly works. However, let's examine an alternative approach in which we envision the Clients data set as a tree structure which we will traverse in order to find matches with the Houses data set. If nothing else, we may learn something and add some additional tools to our SAS tool belt. Before we dive into the description of a tree-based algorithm for merging our data sets, let's step back for a moment and provide a quick overview of what a tree is from a computer science perspective.

A Brief Overview of Trees

In computer science, a tree is a data structure consisting of nodes (or vertices) connected by edges (or branches, to continue the metaphor). Unlike the more generic data structure known as a graph, a tree may not contain any cycles. That is, between any two nodes, there must be one and only one path along the edges. Furthermore, a tree is hierarchical: a single node is identified as the root, and all other nodes are considered descendants of the root. The nodes directly connected by a single edge to the root are the root's children; the root is the parent of those same nodes. Those nodes in turn may have their own child nodes (which naturally are termed the grandchildren of the root). In a generic tree, each node can have any number of children (including no children). Nodes that have no children are called leaves. The number of levels in a tree is equal to the longest path in the tree from the root to any leaf (the root is typically considered to occupy level 0; the root's children occupy level 1, and so on). Finally, a tree is a recursive data structure: every child node forms a sub-tree in which it itself is the root.

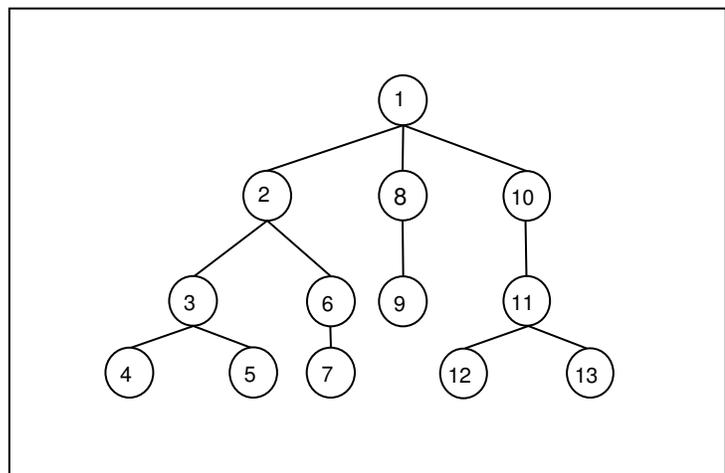
There are many varieties of tree data structures in computer science: for example, by restricting the number of children permitted to any single node to a maximum of two, we have a binary tree. By requiring that each node have exactly two children (and no less), we have a full binary tree. Other varieties of tree data structures may restrict how nodes are added to and removed from the tree (and how the other nodes in the tree may or may not be rearranged as a result).



Let's use our new knowledge of trees to identify the relevant portions of the example tree in the figure to the left. Node A is the root of the tree. Nodes B, C, and D are node A's children; node A is the parent of B, C, and D. Similarly, nodes E and F are the children of B, and B is the parent of E and F; E and F are siblings. Nodes I, J, K, G, L, and M are the leaves of the tree. The tree has 3 levels (depth of 3) because the longest path from the root to the leaves requires traversing 3 edges (recall that the root is level 0). Notice that there are shorter paths to the leaves, however: the leaf G is at the second level of the tree. Notice also that not all nodes have the same number of children: some have three children, some have two children, some have one, and of course the leaves have no children. Finally, notice that there are no cycles in the tree; that is, convince yourself

that between any two nodes, there is one and only one path that you can travel. Were we to add an edge between nodes F and G, for example, we would have a cycle and thus no longer have a tree; instead, we would have formed a graph, which has its own uses in computer science (such as modeling maps), but has no place in our present discussion.

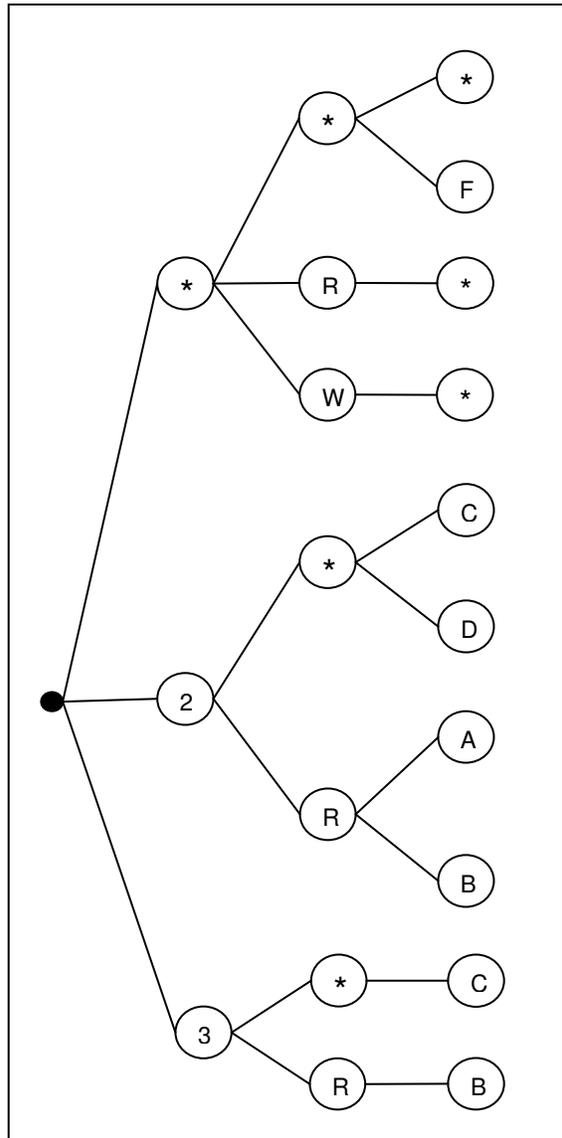
Now that we have a basic understanding of tree-based data structures and their associated terminology, let's discuss one of the most important aspects of working with trees: tree traversal. Once we've stored our data in a tree, how do we go about retrieving that data in a useful manner? Tree traversal is the process of visiting each node in a tree (and possibly performing some action on the data stored in each node as you visit it), by following the edges of the tree in some order. There are several approaches we could take to tree traversal. For example, a breadth-first traversal might visit each node in a level, from left-to-right, before visiting the nodes in the next level. Another common traversal strategy is a depth-first strategy: we might visit the left-most child, and then it's left-most child, all the way until we reach the left-most leaf; having reached the left most leaf, we would backtrack to its parent and then visit the next left-most child. In other words, we would move down and to the left whenever possible, backtracking once we reach a leaf, until we've visited all nodes in the tree. The example to the right illustrates a depth-first traversal of a tree using the left-most child strategy we just described (the numbering of the nodes indicates the order in which they would be visited using this particular depth-first strategy).



Tables as Trees

Let's revisit the Clients data set from the introduction. Below is the same data set, but the client variable has been moved to the right and the data set has been sorted by the attribute variables. The borders of the data set have been drawn to clearly distinguish the by-groups within each variable. What if we interpreted each by-group as a node in a tree? Variables to the left would hold the parent nodes of the variables to the right, so that each variable essentially serves as a level in the tree. To the right of the data set, we have a tree-based representation of the same data set (the unlabelled black circle at the far left of the tree is the root, which does not have a corresponding entry in the data set).

num_beds	ext_color	price_pnt	client
*	*	*	Carl Clueless
*	*	F	Sam Saver
*	R	*	Monte Moneybags
*	W	*	Monte Moneybags
2	*	C	Fred Familyman
2	*	D	Sam Saver
2	R	A	H. I. Maintenance
2	R	B	Fred Familyman
2	R	B	H. I. Maintenance
3	*	C	Fred Familyman
3	R	B	Fred Familyman



Depth-first Traversal and Pruning to Perform a Merge

Our ultimate goal is to merge this data set with the Houses data set. There are two key observations that we need to make to accomplish this goal:

- (1) A path from the root to the leaves, in which the data in the nodes in the Clients tree matches the values of the corresponding variables in the Houses data set (or the node in the Clients tree is a wild card) represents a match between the data sets.

- (2) Each node has at most two child nodes that we might need to visit to find a path from the root to the leaves: first, a wild card node (if present), and second, a node whose data matches the value of the corresponding variable in the Houses data set for the current observation.

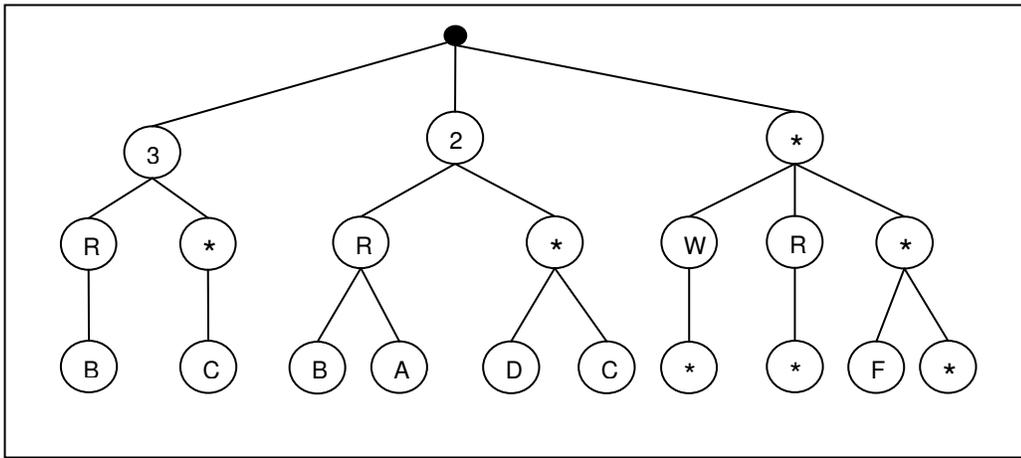
Observation (1) suggests that we are interested in a depth-first search strategy, because we are looking for paths from the root to the leaves to find matches between the data sets. Observation (2) suggests that we might want to employ some form of pruning strategy to eliminate the sub-trees that have no chance of providing a path to the leaves so that we don't waste time visiting nodes that we don't need to. In fact, if we employ such a pruning strategy, we will build a binary tree as we go which contains exactly those paths that represent matches between the two datasets.

An Example Traversal

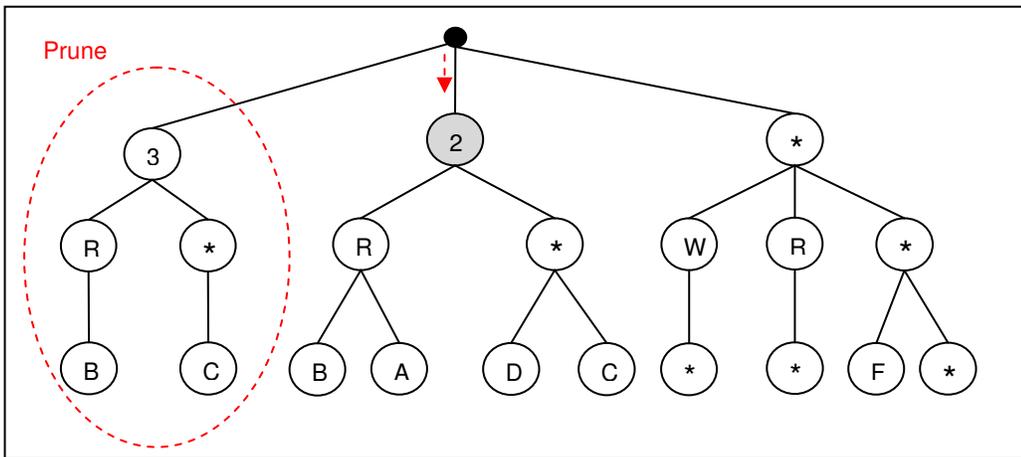
Using the second record in the Houses data set as an example, let's attempt a traversal of the Clients tree to illustrate our approach. Below is the observation we will use in this example:

House	num_beds	ext_color	price_pnt
2	2	R	B

Below is the Clients tree, simply rotated 90 degrees so that the root is at the top:

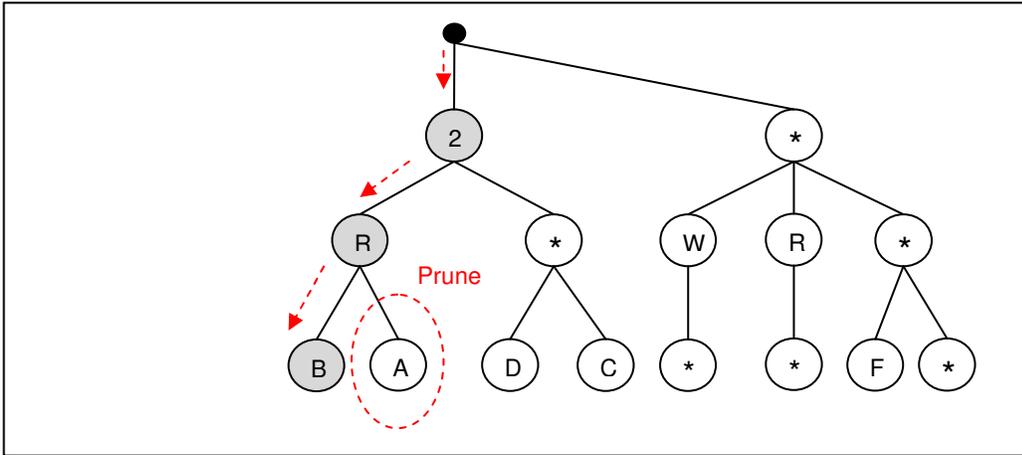


We begin the traversal at the first level in the tree. Recall Observation (2) from earlier: there are at most two sub-trees that we need to worry about traversing, the wild card sub-tree and the matching value sub-tree. Level one in the tree corresponds to the num_beds variable in the Houses data set, whose value for this observation is 2. Thus, the sub-tree whose root value is 3 (circled below) can be pruned. Thus, we begin our traversal with the node with value = 2:

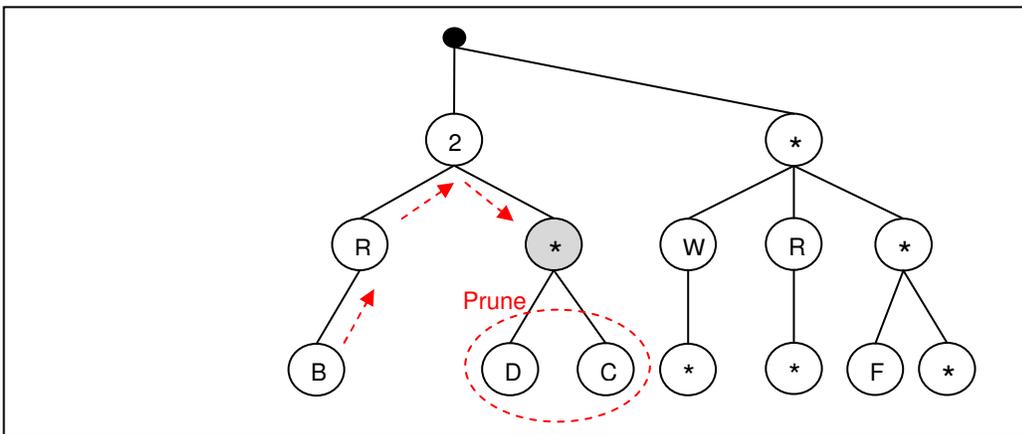


Please note that we are merely performing a logical prune of the tree – we are not going to physically delete the sub-trees that we prune, because we might need them for matching future observations from the Houses data set. Instead, pruning in this context just means that we will ignore those portions of the tree during our traversal.

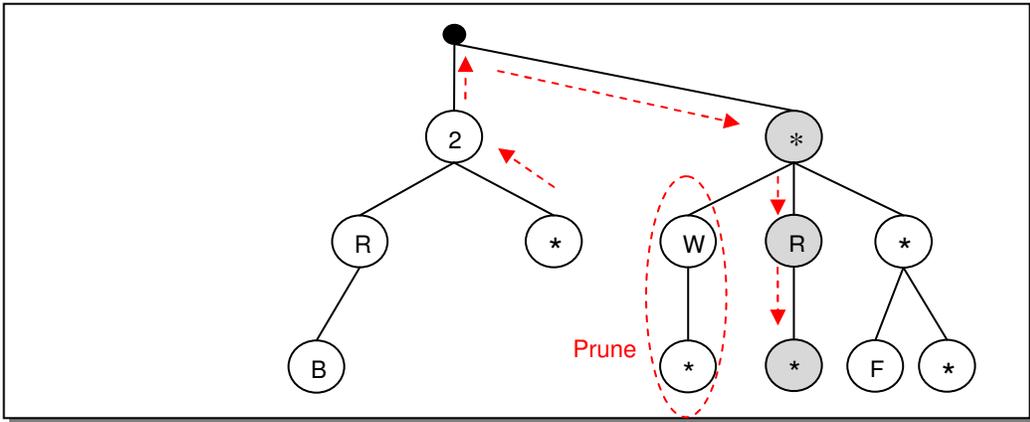
We are performing a depth-first traversal, so we will now visit the children of the shaded node. Thus, we are visiting nodes at level two in the tree, which correspond to the `ext_color` variable in the Houses data set. In this case, because `ext_color = "R"` in the Houses data set, there is nothing to prune at this level in this sub-tree, and we continue our traversal with the leaves under the shaded node. The leaves correspond to the `price_pnt` variable, whose value in our current record is "B". We prune the leaves under the current node, keeping only a wild card node (there is none in this case) and a node whose value is "B". The current state of our traversal is illustrated below:



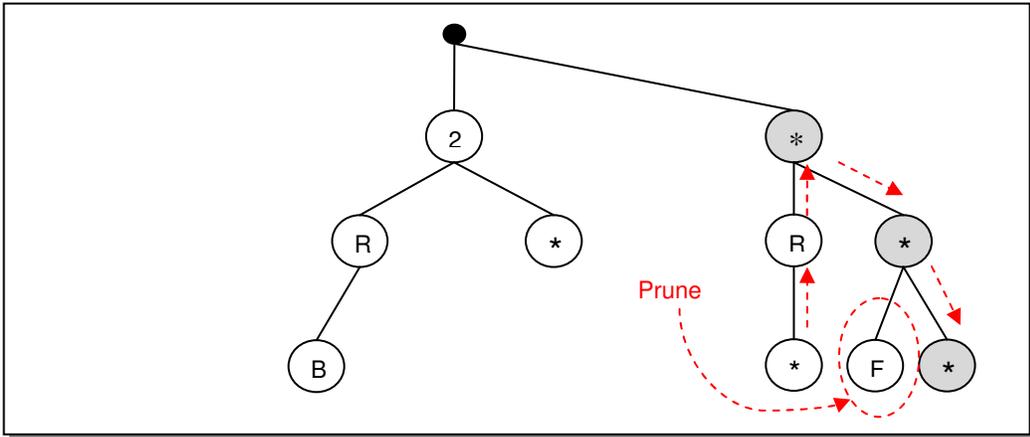
Because we've successfully reached a leaf, we have our first match! The match corresponds to the record(s) in the Clients data set in which `num_beds = 2`, `ext_color = "R"`, and `price_pnt = "B"` – there are actually two such records in the Clients data set, one for Fred Familyman and one for H.I. Maintenance. Both of those clients therefore have a match with house #2. We now continue our traversal, which, because we've reached a leaf, requires some back-tracking and then a continuation of the depth-first strategy. The path that we follow is illustrated below:



Once we reach the wild card node (shaded above), which is an automatic match, we prune the children as before and find that there are no remaining nodes to visit. There is no match and thus we back-track to the root and continue a depth-first traversal of the from the root's wild card child. We examine level two nodes under the level one wild card node and prune as before:



We have found another match, represented by the shaded nodes above, which corresponds to a record in the Clients data set for Monte Moneybags. We continue our algorithm as illustrated below to find the fourth, and final, match for house #2 (a match for Carl Clueless):



To convince ourselves that this worked properly, let's examine the subset of the result data set created by the PROC SQL solution for house #2:

client	house	num_beds	ext_color	price_pnt
Carl Clueless	2	2	R	B
Fred Familyman	2	2	R	B
H. I. Maintenance	2	2	R	B
Monte Moneybags	2	2	R	B

PROC SQL also found four matches for house #2, one each for Carl Clueless, Fred Familyman, H.I. Maintenance, and Monte Moneybags. Our tree-based approach produced the same result.

Implementation

Now that we understand the basic idea behind the algorithm, albeit at a high level, let's discuss how we might implement this solution in SAS. We'll begin by determining how to store the tree structure as a whole, which, due to some limitations in the SAS language related to dynamic memory allocation and user-defined data types, we may have to be creative with. We'll then present pseudo-code for the two key aspects of the algorithm: loading the look-up data set into the tree and performing the actual tree traversal. Finally, we'll look at the actual SAS code, breaking the code into small chunks and discussing each section of code in detail. The full, final SAS macro (sans error checking that should be present in production code) is presented in Appendix A at the end of the paper.

Simplifying Assumptions

While production code may not have the luxury to make the following simplifying assumptions, we do take a certain luxury in this paper in order to keep the SAS code as simple as possible. Specifically, we assume:

1. That the key variables are all character
2. That the key variables are all the same length (even if that length exceeds the required storage for shorter variables)

These assumptions could be removed by modifying how the code is implemented, requiring further complexity in the solution.

Storing the Tree

Because a tree structure is typically dynamic and can grow and shrink over time, programmers traditionally use dynamic memory allocation to store trees. The structure of a single node is determined by the programmer, and then nodes are created and destroyed on an as-needed basis during program run-time, using only as much memory as is needed to store the tree. In our case, we don't plan to grow and shrink the tree except during its initial creation in which we load the Clients data set into the tree data structure. Still, we won't necessarily know how many nodes we'll need ahead of time, and we'd rather not waste an entire pass through the Clients data set to compute the required size: we'd like to build the tree in one pass, dynamically allocating additional storage as needed.

While SAS does not permit dynamic memory allocation in the generic sense permitted by languages such as C, it does (as of version 9) include a dynamically allocated data structure: the hash object. To store our tree, we will create a couple of hash objects and utilize them to store the nodes and leaves of our tree, treating the hash objects as associative arrays.

The basic idea behind our storage and retrieval strategy is that we will assign each node a unique numeric identifier as we create the tree, starting with 0 for the root. The first hash table that we create will take as keys the parent's node number and the child's data value and return the child's node number. This hash table will prove extremely useful in pruning our tree: as we visit each level, we know the parent node number that we came from (because we were just there) and we know the two child data values that we're looking for (asterisk for wild card and the current data value in the Houses data set). Performing a hash find() operation will return the node numbers for the child nodes (if any) matching those requirements. We can think of this table as storing the edge information from each parent to each of its children.

The second hash table that we create will store information for each leaf of the tree. The hash object will take the node number of the leaf and a numeric count (leaf count) as keys and return the corresponding satellite date from the look-up table. The leaf count variable is needed in the key in the event that there is more than one record in the look-up data set for a given set of key values. For example, there are two records in the Clients data set in which the key variables are set as num_beds = 2, ext_color = "R", and price_pnt = "B". In this case, the leaf count allows us to return both matching records when the tree merge succeeds.

In addition to the main hash table and leaf hash table described above, we will need to keep track of some bookkeeping information during the traversal. Basically, we need to remember which nodes we've visited along the current path, so that we can backtrack when needed, and we need to remember, for each node along the current path, the status of our visit (not yet visited, visited left child, visited both children) when we return to that node. For these requirements, we only need to store data for the current path; because the current path has a maximum length equal to the number of levels in the tree, and because the number of levels in the tree is simply the number of key variables used in the merge, we can simply use a couple arrays to store this data.

Tree Creation Pseudo-code

There are two parts to the SAS code that might be considered tricky. The first is the method employed to load the look-up data set into the main hash table and the leaf hash table. The pseudo-code presented below will be executed exactly once in the data step (i.e. executed inside an "if _n_ = 1 then do" block). Omitted from the pseudo-code are the data declarations necessary to set-up the hash tables, which are described above. Note that when assigning node numbers to the tree, node 0 refers to the root.

```

Set node-count to 0

For each observation in look-up data set
  Set parent-node-number to 0

  For each key variable

    If hash does not contain parent-node-number and current key value Then
      Increment node-count
      Set node-number to node-count
      Add parent-node-number and key value to hash with node-number as data
    End If

    Set parent-node-number to node-number
  End For

  Find highest leaf-count for node-number in leaves hash
  Add satellite data to leaves hash for node-number and leaf-count + 1
End For

```

Tree Traversal Pseudo-code

Most computer science text books present tree traversal as a recursive operation. Because trees are a recursive data structure (a sub-tree of a tree is itself a tree), trees easily lend themselves to recursive operations. However, recursion is not well supported in SAS. Thus, we will need to write an iterative version of a depth-first tree traversal. The below pseudo-code assume that the tree has already been stored in the hash tables as illustrated above. Also, to keep the below pseudo-code short, I use the short-hand "visit that child" to refer to three operations: (1) obtain the node number of the child from the main hash table, (2) increment the level index, (3) update the path history arrays at the new level index with the new node number and with a status that indicates that the new node number has never been visited.

```

Set level to 1 (Count root as level one since SAS arrays are 1-based)
Set current node to root
Mark that current node has never been visited

Do until root attempts to backtrack to its parent
  If we reach a leaf
    Output all satellite data from leaves hash
    Backtrack to parent
  End If

  Else if current node has never been visited then
    Mark node as having wildcard child checked
    If node has a wildcard child then visit that child
  End If

  Else if current node has checked wildcard child then
    Mark node as having checked both children
    If node has a matching value child then visit that child
  End If

  Else if current node has checked both children then
    Clear path history arrays at this level
    Backtrack to parent
  End If
End Do

```

At Long Last: the SAS Code

This section will look at the SAS code in small sections and explain them; the full, uninterrupted SAS code is presented in Appendix A of this paper. The first section of the macro is the %macro statement itself and the parameter declarations:

```
%macro wildcard_merge(  
  master=,      /* Master table */  
  lookup=,      /* Look-up table containing wildcards */  
  key=,         /* Key variables */  
  keySize=1,    /* Size of largest key variable (1 by default) */  
  satellite=,   /* Satellite var(s) on &lookup, quoted and comma-sep */  
  out=,         /* Output dataset */  
  wildchar=*,   /* Wildcard character (asterisk by default) */  
  hashexp=16,  /* Value for hashexp parameter (16 by default) */  
);
```

Next we create a local constant to make checking the result of hash operations more self-documenting; hash methods such as find() and check() return a 0 on success:

```
%local found;  
%let found = 0; /* Return value from hash find() on success */
```

Next we determine the number of key variables passed to this macro. Because the caller of this macro may use a variable range (like key1-key10), it's nice to let SAS do the work in determining the list of variables that make up the key:

```
/* Determine number of key variables */  
%local numLevels;  
  
data _null_;  
  set &master;  
  array keys[*] &key;  
  call symput("numLevels", put(dim(keys) + 1, 8.)); * +1 for root;  
  stop;  
run;
```

We can now begin the main algorithm, which takes place within a single DATA step. The first portion of the step occurs within an "if _n_ = 1 then do" block which builds the tree from the look-up data set. The below code includes just the portion of that first block which declares the hash objects as described in the earlier implementation sections of this paper.

```
/* Main algorithm */  
data &out;  
  
  /* Build Tree */  
  if _n_ = 1 then do;  
    length nodeValue $&keySize parentNodeNbr nodeNbr leafCount 8;  
    drop nodeValue parentNodeNbr nodeNbr leafCount;  
  
    declare hash tree(hasexp: &hashexp);  
    tree.defineKey('parentNodeNbr', 'nodeValue');  
    tree.defineData('nodeNbr');  
    tree.defineDone();
```

Continuing the hash definition with the leaves hash:

```
declare hash leaves(hashexp: &hashexp);
leaves.defineKey('nodeNbr', 'leafCount');
leaves.defineData(&satellite);
leaves.defineDone();
```

Below is the remainder of the tree-building code, which closely follows the pseudo-code in the Tree Creation Pseudo-Code section earlier in the paper. Notice that we permit multiple look-up records to correspond to the same leaf node via the leafCount key in the leaves hash. We determine how many records already map to each leaf before adding a new record by checking each leafCount value (starting from 1) until we find no match in the leaves hash.

```
nodeCount = 0; /* Max node number in tree */
drop nodeCount;

/* Load lookup data set into tree */
do until(end);
  set &lookup end=end;
  array keys[*] &key;

  parentNodeNbr = 0; /* Zero represents the root */

  do index=1 to dim(keys);
    nodeValue = keys[index];
    nodeNbr = nodeCount + 1;
    if tree.find() ne &found then do;
      tree.add();
      nodeCount = nodeCount + 1;
    end;
    parentNodeNbr = nodeNbr;
  end;
drop index;

/* Build leaf lookup - last nodeNbr is a leaf */
do leafCount = 1 by 1;
  if leaves.check() ne &found then leave;
end;
leaves.add();
end;
end;
```

At this point, the tree has been stored (between the two hash tables), and the “if _n_ = 1 then do” block is complete. From here on, normal processing of the DATA step (i.e. the implicit DATA step loop) takes over. First, we create a couple of arrays to store information on the path that are currently following during our traversal:

```
/* Define history arrays to keep track of our traversal */
array nodeNumber[&numLevels] _temporary_;
array nodeStatus[&numLevels] _temporary_;
```

We then read one observation from the master data set, which will we attempt to match to the tree using tree traversal and pruning:

```
/* Read the current master record */
set &master;
```

Finally, we are ready to begin the traversal of the tree for the current master record. Any paths that we find from the root to the leaves represent a match that we will output. We begin with some set-up code. We are going to track where we are in the tree using a level variable, which will serve as an index to the history arrays defined above. We begin with level = 1, which represents the root (despite the discussion in the Brief Overview of Trees that the root is at level 0, we are using level 1 for the root because SAS arrays start with index 1). We set our path to begin at the root and mark that the root is not visited. Each variable in the nodeStatus array will take on one of three possible values over the course of the traversal: 0 = no children visited; 1 = wildcard child visited (or does not exist); 2 = both children visited (or do not exist). Backtracking in this algorithm basically consists of moving back up a level in the path history and then continuing the traversal; if we backtrack from the root because it is fully visited, we will set level to 0 which will indicate that our traversal is complete.

```

/* Traverse the tree (iteratively) to find all matches */
level = 1; /* Level 1 is the root */
drop level;

nodeNumber[level] = 0; /* Start at the root */
nodeStatus[level] = 0; /* Mark that no children are visited */

do until (level = 0);

```

At each stage of the traversal, there are four things that might happen:

1. We reach a leaf
2. The current node has never been visited
3. We visited (or attempted to visit) the wildcard child of the current node
4. We visited (or attempted to visit) both children of the current node

Each of these cases is handled one at a time within the do loop. We begin with the base case of reaching a leaf, where we output all of the satellite records associated with the leaf and then backtrack to the parent.

```

/* Base Case: Reached a leaf */
if level = &numLevels then do;
  do leafCount = 1 by 1;
    if leaves.find() ne &found then leave;
    output;
  end;
  level = level - 1; /* Back-up to parent */
end;

```

We continue with the case in which no children have been visited. In this case, we check whether a wildcard child exists, and if so, we set variables appropriately so that we visit it on the next iteration of the loop.

```

/* Case 0: No children visited */
else if nodeStatus[level] = 0 then do;
  nodeStatus[level] = 1;

  /* Determine whether this node has a wild char child */
  parentNodeNbr = nodeNumber[level];
  nodeValue = "&WILDCHAR";
  if tree.find() = &found then do;
    level = level + 1;
    nodeNumber[level] = parentNodeNbr;
    nodeStatus[level] = 0;
  end;
end;

```

We continue with the case in which the wildcard child has already been visited, or, if there is no wildcard child, that fact has already been checked.

```
/* Case 1: Already visited (or attempted) wild card child */
else if nodeStatus[level] = 1 then do;
    nodeStatus[level] = 2;

    /* Determine whether this node has a value child */
    parentNodeNbr = nodeNumber[level];
    nodeValue = keys[level];
    if tree.find() = &found then do;
        level = level + 1;
        nodeNumber[level] = nodeNbr;
        nodeStatus[level] = 0;
    end;
end;
```

We finish with the case in which both children have been visited. If one or both children do not exist, that fact has been checked already. In this case, the only thing to do is to clear the history arrays for the current level and back-track to the parent:

```
/* Case 2: Already visited (or attempted) both children */
else if nodeStatus[level] = 2 then do;
    nodeNumber[level] = .;
    nodeStatus[level] = .;
    level = level - 1; /* Back-up to parent */
end
```

Before we exit the loop, we code a block to catch the impossible case in which the node status is set to some other value not already discussed. Then we close all open conditionals, loops, the DATA step itself, and the macro itself:

```
/* Unexpected case - Error condition */
else do;
    put "ERROR: Unexpected condition";
end;
end;
run;
%mend;
```

Benchmarking: Tree-based Merge vs. SQL

If we stop to look back at the PROC SQL solution presented at the beginning of this macro, we might ask ourselves why we bothered to write such a complicated solution when a few lines of PROC SQL would do. Beyond learning about trees and playing with SAS's hash functionality, it does turn out that in most cases, unless the look-up table is very small relative to the size of the master table, the tree-based approach is faster – much faster.

To illustrate this, I performed the following benchmarking procedure. Using a SAS program, I randomly generated master and lookup datasets, where the key variables were of length one and could take on any character from A to J with equal probability. In the wildcard dataset, there was a 50% chance that the key variables would take on a wildcard value instead. The program merged the master and lookup data sets using both the PROC SQL approach and the tree-based approach and the resulting data sets were compared to confirm that both approaches produced same result.

This experiment was executed several times, varying three parameters: (1) the number of keys used in the merge, (2) the number of observations in the master data set, and (3) the number of observations in the lookup dataset. For each combination of parameters, I ran 3 trials and reported the average real execution time in the table below (Note: for the one case where $\log_{10}(M*L) = 10$, I only ran one trial due to the long SQL execution time). The benchmarking environment is enumerated below:

Execution environment:

- SAS 9.1.3 SP4
- AIX 5.3 (64-bit)
- SPDS(AIX) 4.40 server

Run locally from:

- Enterprise Guide 4.1
- Windows XP SP3

The results of the benchmarking are summarized in the following table:

Num Keys	Master Obs	Lookup Obs	$\log_{10}(M*L)$	SQL Algorithm	Tree Algorithm	Speed-up
10	10,000	1,000	7	0:02.4	0:00.6	4.05 times
10	1,000	10,000	7	0:02.5	0:00.3	8.62 times
10	10,000	10,000	8	0:23.8	0:01.8	13.27 times
20	10,000	10,000	8	0:23.5	0:02.2	10.90 times
10	10,000	100,000	9	3:52.1	0:07.1	32.88 times
10	100,000	10,000	9	3:53.2	0:17.0	13.74 times
20	10,000	100,000	9	3:51.6	0:12.1	19.17 times
20	1,000,000	1,000	9	3:45.7	1:01.2	3.69 times
20	10,000,000	100	9	3:40.5	3:22.4	1.09 times
20	1,000,000	10,000	10	37:51.5	3:11.7	11.85 times

The column $\log_{10}(M*L)$ is the base-10 logarithm of the number of master data set observations times lookup data set observations. Notice that for a given value in this column, the SQL approach has a fairly consistent execution time. However, it is very interesting to note that as the number of observations in the lookup data set shrinks while the $\log_{10}(M*L)$ value is held constant, the tree algorithm's execution time worsens and approaches the SQL algorithm. It seems, then, that the tree-based algorithm is most useful when the size of the lookup data set is large – in fact, it appears that the larger the lookup dataset is relative to the master data set, the better the tree algorithm performs. Varying the number of keys does not appear to have a substantial impact for either algorithm.

Conclusion

Merging two data sets in which the key values of the look-up data set contain wildcards can be a challenge. This paper presented two techniques to perform such a merge: a simple approach using PROC SQL and a more complex tree-based algorithm. Having thoroughly discussed the background for the algorithm and its implementation, the paper showed that the tree-based approach, while more complicated to implement, provides an impressive performance gain if the size of the data set containing wild cards in the tree is large relative to the other data set.

References

1. SAS OnlineDoc 9.1.3. <http://support.sas.com/onlinedoc/913/docMainpage.jsp>
2. Harel, David. *Algorithmics: The Spirit of Computing*. 2nd Edition.
3. Cormen, Thomas H., et al. *Introduction to Algorithms*. 2nd Edition.

Contact Information

Your comments and questions are valued and encouraged. Contact the author at:

Matthew Nizol
Clinical Programmer
United Biosource Corporation
2200 Commonwealth Blvd, Suite 100
Ann Arbor, MI 48105
734-994-8940 ext 1605
matt.nizol@unitedbiosource.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies.

Appendix A: Full SAS Macro

```
%macro wildcard_merge(  
  master=,      /* Master table */  
  lookup=,      /* Look-up table containing wildcards */  
  key=,         /* Key variables */  
  keySize=1,    /* Size of largest key variable (1 by default) */  
  satellite=,   /* Satellite var(s) on &lookup, quoted and comma-sep */  
  out=,         /* Output dataset */  
  wildchar=*,  /* Wildcard character (asterisk by default) */  
  hashexp=16,  /* Value for hashexp parameter (16 by default) */  
);  
%local found;  
%let found = 0; /* Return value from hash find() on success */  
  
/* Determine number of key variables */  
%local numLevels;  
  
data _null_;  
  set &master;  
  array keys[*] &key;  
  call symput("numLevels", put(dim(keys) + 1, 8.)); * +1 for root;  
  stop;  
run;  
  
/* Main algorithm */  
data &out;  
  
  /* Build Tree */  
  if _n_ = 1 then do;  
    length nodeValue $&keySize parentNodeNbr nodeNbr leafCount 8;  
    drop nodeValue parentNodeNbr nodeNbr leafCount;  
  
    declare hash tree(hashexp: &hashexp);  
    tree.defineKey('parentNodeNbr', 'nodeValue');  
    tree.defineData('nodeNbr');  
    tree.defineDone();  
  
    declare hash leaves(hashexp: &hashexp);  
    leaves.defineKey('nodeNbr', 'leafCount');  
    leaves.defineData(&satellite);  
    leaves.defineDone();  
  
    nodeCount = 0; /* Max node number in tree */  
    drop nodeCount;  
  
    /* Load lookup data set into tree */  
    do until(end);  
      set &lookup end=end;  
      array keys[*] &key;
```

```

parentNodeNbr = 0; /* Zero represents the root */

do index=1 to dim(keys);
    nodeValue = keys[index];
    nodeNbr = nodeCount + 1;
    if tree.find() ne &found then do;
        tree.add();
        nodeCount = nodeCount + 1;
    end;
    parentNodeNbr = nodeNbr;
end;
drop index;

/* Build leaf lookup - last nodeNbr is a leaf */
do leafCount = 1 by 1;
    if leaves.check() ne &found then leave;
end;
leaves.add();
end;
end;

/* Define history arrays to keep track of our traversal */
array nodeNumber[&numLevels] _temporary_;
array nodeStatus[&numLevels] _temporary_;

/* Read the current master record */
set &master;

/* Traverse the tree (iteratively) to find all matches */
level = 1; /* Level 1 is the root */
drop level;

nodeNumber[level] = 0; /* Start at the root */
nodeStatus[level] = 0; /* Mark that no children are visited */

do until (level = 0);
    /* Base Case: Reached a leaf */
    if level = &numLevels then do;
        do leafCount = 1 by 1;
            if leaves.find() ne &found then leave;
            output;
        end;
        level = level - 1; /* Back-up to parent */
    end;

    /* Case 0: No children visited */
    else if nodeStatus[level] = 0 then do;
        nodeStatus[level] = 1;

        /* Determine whether this node has a wild char child */
        parentNodeNbr = nodeNumber[level];
    end;
end;

```

```

nodeValue = "&WILDCHAR";
    if tree.find() = &found then do;
        level = level + 1;
        nodeNumber[level] = nodeNbr;
        nodeStatus[level] = 0;
    end;
end;

/* Case 1: Already visited (or attempted) wild card child */
else if nodeStatus[level] = 1 then do;
    nodeStatus[level] = 2;

    /* Determine whether this node has a value child */
    parentNodeNbr = nodeNumber[level];
    nodeValue = keys[level];
    if tree.find() = &found then do;
        level = level + 1;
        nodeNumber[level] = nodeNbr;
        nodeStatus[level] = 0;
    end;
end;

/* Case 2: Already visited (or attempted) both children */
else if nodeStatus[level] = 2 then do;
    nodeNumber[level] = .;
    nodeStatus[level] = .;
    level = level - 1; /* Back-up to parent */
end;

/* Unexpected case - Error condition */
else do;
    put "ERROR: Unexpected condition";
end;
end;
run;
%mend;

```