Using HASH to find a sum over a transactional path

Richard A. DeVenezia, Independent Consultant, Remsen, NY

Abstract

The HASH component object has become well established in current DATA Step programming as a versatile tool for solving a wide variety of data processing and transforming problems. The principal strengths of a hash are automatic dynamic memory allocation and keyed access to data.

This paper will demonstrate how to use a hash to traverse linked records in a transactional data set. Code was run in SAS® Foundation version 9.2 but will also run in earlier version 9 releases.

Keywords: Hash, Hiter, Transaction, Linked List

Data

Transactional data can take on many forms. For this paper the following business rules apply:

- Each transaction record has an identifier (tid), an amount and a prior transaction identifier (pid)
- Each tid is a unique 15 digit character string
- A transaction may be the prior transaction of only one other transaction
- A set of related transactions comprise a path and are linked through the pid.
- When a transaction is not a successor in a path, its pid will be set to missing
- A path P of length N > 1 contains transactions t₁,...,tn such that ti.prid = ti-1.tid for i=2 to N
- Paths of length 1 are valid

The diagram shows a sample with two paths.

Task

You are tasked with computing the total amount for each transaction path.

The two paths in the diagram are:
start_tid: 12345 total: $305, and
start_tid: 35000 total: $875

Analysis

The data set and business rules focus on tid as the key and pid and amount as linkage and auxiliary data respectively. The rules are accurate but do have some redundant statements regarding prior, successor and path lengths.
Concentrating on the backward linkage may induce a predisposition for developing a last (tail) to first (head) solution. The main problem with this approach is that, in a single record, there is no way to determine if a given record is the tail. The biases introduced in the business rules obscure a sublime and compact solution.

Consider what happens when the roles of **pid** and **tid** are reversed.

When the **pid** is the key and the **tid** is the linkage data the path traversal will flow from head to tail. The head is easily determined because its **pid** is set to missing. The paper does not discuss issues that arise from broken data. For example, the case when a record's **pid** points to a non-existent **tid**.

**DATA Step**

```
data trail_sums;
  keep tid ntids total_amount;
```

For each path starting **tid** we want the number of nodes in the path and the SUM of the amounts.

**Hashes**

Consider a path as a series of nodes. Each path has a head node. Each node in a path has a pointer to the next node in the path – opposite that of the data which has a pointer to the prior node. The data for the role reversed mapping is available in a single record; given a prior id (**pid**) we know what transaction id (**tid**) will be next.

The DATA Step will utilize two hashes:

**heads** - key: the starting **tid** of each path, data: the transaction amount. Items are only added when **pid** is missing.

**trans** - key: the **pid** of a transaction, data: the **tid** of the transaction and its amount. Items are only added when **pid** is not missing. *Note: The key is distinct because a transaction can have only one prior transaction.*

**Hash preparation**

The DECLARE HASH statement and DEFINE methods are used prepare the hashes:

```
DECLARE HASH heads(hashexp:9);
heads.defineKey ('tid');
heads.defineData('amount');
heads.defineDone();

DECLARE HASH trans(hashexp:10);
trans.defineKey ('pid');
trans.defineData('tid','amount');
trans.defineDone();
```

**hashexp:** is a hash constructor argument that specifies the power of two exponent number of slots to use internally. SAS Documentation indicates that a hash having one million keys will perform better when the default setting of 8 is overridden with a setting of 9 or 10.
The `defineKey` and `defineData` methods specify the role that DATA Step Program Data Vector (PDV) variables will have as they are tacitly interfaced with the hash object during ADD and FIND methods calls.

**Hash population**

The hash objects are populated during an explicit loop over the input dataset.

```plaintext
do while (not done);
  set transactions end=done;
  if missing(pid)
    then heads.add();
    else trans.add();
  end;
```

Remember that `pid` is the key of `trans` and `tid` is the data, whereas `tid` is the key in the dataset and `pid` is the data. Reversing the key and data roles changes the backward linking structure of the dataset into a forward linking structure.

*Note: The HASH dataset: constructor argument is not used because we are also tracking the path heads.*

**Path iteration**

The data is processed using nested loops. The outer loop iterates the path heads and the inner loop traverses nodes forward from the head node.

The DECLARE HITER statement prepares an object for stepping through each item in a hash. The outer iteration will rely on this. The inner iteration will not use an HITER, but instead will iterate along the linkages until there is no next node.

```plaintext
DECLARE HITER headsIter('heads');
do while (headsIter.next() = 0);
  total_amount = amount;
  ntids = 1;
  do while (trans.find(key:tid) = 0);
    total_amount + amount;
    ntids + 1;
  end;
  output;
end;
STOP;
```

The source code is short and subtle. There are several side effects going on and they are elucidated in the following points:
### Point 1

```ruby
DECLARE HITER headsIter('head');
do while (headsIter.next() = 0);
```

The `.next()` method advances to the next item in a hash and populates the PDV host variables. A non-zero return means the last item in the hash has been reached. *Note: When an HITER is instantiated it will be in a 'ready-state' prior to the first item, and thus <hiter>.FIRST does not need to be explicitly called.*

### Point 2

```ruby
total_amount = amount;
ntids = 1;
```

These statements occur in the heads iteration and initialize the summary variables. Total amount is set to the amount in the head node -- the preceding `next()` had automatically updated the host variable named 'amount'. The number of nodes counter `ntids` is reset to 1.

### Point 3

```ruby
do while (trans.find(key:tid) = 0);
```

This statement is a traversal over the forward linked list that was created during the population phase. The `key:` argument overrides the automatic use of `pid` value as the key, and causes the current `tid` value to act in the role of the pid-key. The `find` method will either locate the next node in the path, or return non-zero when the tid is the last node in the path.

What happens when the next node is located?

The 'amount' variable will receive the transaction amount of the next node, and **most importantly** the `tid` will be updated to that of the next node and will be used in the while test-condition

### Point 4

```ruby
  total_amount + amount;
  ntids + 1;
```

This statement occurs within the path traversal and accumulates the node amounts in `total_amount`.

### Point 5

```ruby
  output;
```

This statement occurs after the path traversal but within the heads iteration, and thus writes to disk a single summarization record for each path starting `tid`.

### Conclusion

Business rules are an important part of any problem solving. Slavish adherence to rules *as stated* can lead to ineffective or suboptimal solutions; take time to examine rules with an analytic frame of mind. The HASH object is a great asset for DATA Step coders, especially those wanting to take runtime performance to the next level. As my nephew would say "HASH FTW".
Contact Information

Richard A. DeVenezia
9949 East Steuben Road
Remsen, NY  13438
(315) 831-8802
http://www.devenezia.com/contact.php

Richard is an independent consultant who has worked extensively with SAS products for over fifteen years. He has presented at previous SUGI, NESUG and SESUG conferences. Richard is interested in learning and applying new technologies. He is a SAS-L Hall of Famer and remains an active contributor to SAS-L.

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.

This document was produced using OpenOffice.org Writer.