LOOKUP - A Plumber's Swiss Army Knife

Rob van der Heij
rvdheij@nl.ibm.com

VM and VSE Technical Conference
Session 3D8
June 2000, La Hulpe, Belgium

Introduction

The ***lookup*** stage has already been available in *CMS Pipelines* for a long time. Looking for a short summary of its function I found the description by John Hartmann, the author of *CMS Pipelines*, most to the point:

> Lookup processes an input stream (detail records) against a reference (master records), comparing a key field. When a detail record has the same key as a reference record, one or both of the records are passed to the primary output stream. Unmatched detail records are passed to the secondary output stream. Unmatched reference records are passed to the tertiary output stream after end-of-file has been reflected on the primary input stream.

Although this description is still correct for the function currently available, there is a lot more to tell about ***lookup***. In this paper I will not just explain the basics of using ***lookup*** but I will also try to cover the enhancements to ***lookup*** in CMS14 (the CMS that comes with VM/ESA 2.3.0).

Novice plumbers should really make themselves familiar with ***lookup*** since it is a very useful part of their toolkit. I hope that even experienced plumbers will find here plenty of new ideas to use ***lookup*** for solving their plumbing problems, as happens to me almost every day while using *CMS Pipelines*.

The Runtime Library

The features of ***lookup*** that I will show in this paper are available in the latest level of *CMS Pipelines* that comes with VM/ESA 2.3.0 (and 2.4.0). Especially if you are still running an earlier level of VM/ESA you should consider to download the free-of-charge **Runtime Library** from the Pipes Web pages at Princeton.

`http://pucc.princeton.edu/%7Epipeline/`

You will not only find here the latest level of *CMS Pipelines*, you will also find several papers on *Pipes*, and useful utilities written by other plumbers.

The Original Use for Lookup

The most common use for ***lookup*** is when you need data from two input streams (e.g. tables) to be combined. Suppose you have a list of all userids defined on the system with the name of that user. The following pipeline will show the names of the users that link to the 19E disk.

```
'PIPE (end \ )',
   '\ cp query link 19E',
   '| split ,',                          /* Split at commas             */
   '| x: lookup w1 detail master',       /* Lookup with userid as key   */
   '| spec w2.2 1 read',                 /* cuu and link mode           */
         '10-* nw',                      /* .. and append the name      */
   '| cons',
   '\ < users list',                     /* Full list of users          */
   '| x:'
```

The ***lookup*** in this example will first read the **master** records from the secondary input (the file "users list"). After the entire stream has been read, ***lookup*** will read records from the primary input. The output of the CP command is split into one userid per record, and passed to the primary input of ***lookup***. The ***detail master*** keywords specify that both the **detail** and the **master** record should be written in that order to the primary output when a detail record is matched by a master record. The ***spec*** stage takes these two records (because of the ***read***) to build a single output record per userid.

The example above assumes that all userids will be in the master file. If this is not the case you will need to do something with the userids that have linked to your disk but are not in the list. The unmatched detail records are written to the the secondary output of lookup. We will just use the userid for those.

```
'PIPE (end \ )',
   '\ cp q link 19E',
   '| split ,',
   '| x: lookup w1 detail master',
   '| spec w2.2 1 read',                 /* cuu and link mode           */
         '10-* nw',                      /* .. and append the name      */
   '| f: faninany',
   '| cons',
   '\ < users list',                     /* Full list of users          */
   '| x:',
   '| spec w2.2 1 ,User, nw w1 nw',     /* no match, use the userid    */
   '| f:'
```

The stream we did not use in this example is the tertiary (or third) output stream where ***lookup*** writes the unused masters (the master records that were not referenced by a detail record). To display these unused masters on the console you would just add another reference (the third) to the label of the ***lookup*** in the example above.

```
   '\ x:',
   '| f:'
```

The naming and numbering of the streams may be confusing when you are new to multi-stream pipelines. The primary or first stream has number 0, the secondary stream is number 1 and the third or tertiary stream is number 2, and so on. You can find a good introduction to multi-stream pipelines in the papers by Melinda Varian and in "*CMS/TSO Pipelines* Author's Edition" by John Hartmann. You can find both at the WEB pages at Princeton.

The softcopy BOOK version of "*CMS/TSO Pipelines* Author's Edition" is also on the latest "VM Collection CD-ROM" (publication SK2T-2067-16)[1]

The single pipeline using ***lookup*** as shown above can replace common REXX programming like the section below.

---

[1] Be aware that the CD-ROM also contains other books that refer to *CMS Pipelines* (i.e. the documentation for VM/ESA 2.4.0). In those other books you will not find most of the ***lookup*** features discussed here.

```
'PIPE cp query link 19E | split , | stem resp.'
do i = 1 to resp.0
  parse var resp.i userid cuu mode .
  'PIPE < users list | locate 1.8 /'left(userid,8)'/ | var answer'
  if symbol('ANSWER') = 'LIT' then say cuu mode 'Who is' userid
  else say cuu mode substr(answer,11)
end
```

Even though both solutions use *CMS Pipelines* you will notice the difference in style. This difference is often referred to as "PipeThink." If you try both solutions with a serious number of logged on users and a non-trivial file, I am pretty sure you will also notice the difference in resource usage.

Specifying the Matched Records to be written

There are a number of keywords that specify what needs to be written to the primary output of ***lookup*** when the key field of a detail record matches the key of a reference record. You can use **detail** to have only the detail record (to do some kind of validation on the detail records) or **master** to only get the reference record from the table. The combination **detail master** or **master detail** will give you both, in the order specified.

The **allmasters** keyword specifies that master records with duplicate keys are also kept in the reference table (they are normally discarded). So the sequence **detail allmasters** will cause ***lookup*** to output the detail record followed by all master records that have the same key.

Depending on the type of data being processed it may be difficult to distinguish between detail record and master records when **allmasters** is used (because you may not know how many master records will match the detail record). In that case it may be helpful to use the **pairwise** option that will output one copy of the detail record for each reference record that was matched.

The railroad track syntax diagrams in the *CMS/TSO Pipelines* documentation show what options can be combined or not, but you will probably find it very intuitive.

The Input Ranges

The ***lookup*** stage optionally takes one or two input ranges to define the key field. The first range defines the key field in the detail record, the second one is for the key field in the master record. If the second input range is omitted it defaults to the input range as specified for the detail records. If both ranges are omitted the entire input record is used as a key field.

The key field used to be just a column range, but with *CMS Pipelines* 1.1.10 this was enhanced to allow the general input range that can be specified with many other stages. Not only can you now do a ***lookup*** against *words* but you can even use the constructs like *substring* to identify the key field. This can save you a lot of ***specs*** stages before and after the ***lookup***.

You may also need to use the ***pad*** option when one of the keys fields could be shorter than the other. There also is an ***anycase*** option when you want a case-insensitive match to be done.

Replace a Cascade of Locates

The ***lookup*** stage is also very handy to select the records that match one or more of the search criteria you have. The classical solution for this is a cascade of ***locate*** or ***pick*** stages like this:

```
'PIPE (end \ name RMHLUP.DOC:124)',
   '\ < rscs logging ',
   '| x1: locate w2 ,DMTNHD144I,',    /* Is it the first key         */
   '| f: faninany ',
   '| process',
   '\ x1:',
   '| x2: locate w2 ,DMTNHD145I,',    /* Is it the second key        */
   '| f:',
   '\ x2:',
   '| x3: locate w2 ,DMTNHD146I,',    /* Is it the third key         */
   '| f:',
   '\ x3:',
   ...
```

If you use ***lookup*** you can replace such a cascade with the following pipeline:

```
'PIPE (end \ name RMHLUP.DOC:139)',
   '\ < rscs logging ',
   '| l: lookup w2 w1 detail ',
   '| process ',
   '\ literal DMTNHD144I DMTNHD145I DMTNHD146I',
   '| split',                         /* Keys as single records      */
   '| l:'                             /* .. and passed as reference  */
```

This solution makes adding a few more keys to search for very easy. If necessary you could even load the keys from a file or produce them with another pipeline. Rita, the Pipeline Profiler, showed that even in the case of just two keys to look for the ***lookup*** solution already uses less CPU time than the equivalent ***locate*** stages.

Plumbers who kept up with recent enhancements may know the ***all*** stage can be used to build a cascade of ***locate*** stages. Since ***all*** under the covers just is such a cascade, it will not run faster than one crafted by hand.

Combine Tasks in a Single Lookup

The following example comes from one of my CGI scripts (Common Gateway Interface - the way to program a web server). When forms data on a web page is sent to the web server the posted data is passed to the CGI in records like this:

```
name=Rob
email=rvdheij@nl.ibm.com
country=nl
```

The portion before the "=" is the field name as specified in the form definition. The value entered by the user is at the right hand side of the "=" sign. The browser only sends fields entered by the user so you need some programming to verify that all required fields are present and supply defaults for optional fields that were omitted.

The following subroutine pipeline reads a list of field names plus their default value through the secondary input stream and then verifies the records on the primary input stream against that list. When the ***lookup*** stage terminates the unreferenced master records (i.e. field names that are valid but were not posted) are written to the primary output of the subroutine as well. This means that the subroutine outputs the entire set of field names with either the posted value or the default (if specified).

```
'callpipe (end \ name ValidPrm)',
   '\ *:',                              /* Posted tags to verify       */
   '| l: lookup anycase',               /* Ignore case difference      */
              'fs = f1 detail',         /* Match on field name         */
   '| i: fanin',
   '| xlate fs = f1',                  /* Make name uppercase         */
   '| insert ,=,',                     /* Make it VARLOADable format  */
   '| *:',
   '\ *.input.1:',                     /* List of tags plus default   */
   '| l:',
   '\ l:',                             /* Unused master records with  */
   '| i:'                              /* their default value         */
```

A pipeline like this can do a lot of the screening work in just a single ***lookup*** stage. It is not difficult to enhance the pipeline to also check that all required fields have been posted.

After this screening the output of the subroutine could be passed to a ***varload*** stage so the remaining REXX program can use the REXX variables corresponding with the field names.

Note: It would not be safe to use ***varload*** without such a check because users could modify the data sent to your server to include field names that were not on the form and thus modify REXX variables that you don't want to be modified in your program (e.g. supply an additional “userid” field).

Counting the Number of References

There are several options that deal with counting the number of references made to a reference record (i.e. the number of hits). With each master key in the table a counter is maintained, originally intended to count the number of hits. The enhancements to ***lookup*** support manipulation of the counter in a few other ways too. I will be showing an interesting example of that later in this paper.

The COUNT option

When the option ***count*** is specified ***lookup*** will record the number of references made with each master key. When the stage terminates all reference records will be written to the tertiary output, prefixed by the value of the corresponding counter. Without the ***count*** option only the unreferenced master records are written to this output stream.

In many situations you can also use ***sort count*** to find the frequency of the keys but ***lookup*** is practical if you need to keep the detail records in the pipeline because it does not reorder the records as ***sort count*** would do. If you are using ***lookup*** already in the pipeline and need to count the hits the ***count*** option is cheaper than the alternative.

The TRACKCOUNT option

While ***count*** only outputs the number of matched records when the ***lookup*** stage terminates, ***trackcount*** prefixes each master record with the current count value when it is written to the primary output.

This option is very useful when you need to assign sequence numbers to all records with the same key. If the input stream was sorted you could do that with the arithmetic built in the ***specs*** stage, but ***trackcount*** does it even without sorting the input stream.

The SETCOUNT option

With the ***setcount*** option you indicate that each master record is prefixed with a 10-byte numeric value that will be used as the initial setting of the counter (the default setting for the counter is of course 0). The main use for this option is to maintain a kind of running total over multiple invocations of the ***lookup*** stage.

The INCREMENT option

When ***increment*** is specified ***lookup*** expects each detail record to be prefixed with a 10-byte numeric value that is added to the counter of the master record that it matches.

A possible way to use this option is to do some kind of score keeping where ***lookup*** sums all values with the same key. When this is combined with ***trackcount*** you can maintain a running total.

Combining the Counting Options

The different options for manipulation of the counter are independent of each other. In particular the fact that ***count*** is not implied by any of the others may be confusing at first but it does give you maximum control over the process.

The description of what records will be written to the tertiary output (the masters that were not matched) has been refined to "the reference records with a count of zero." Using the ***increment*** and ***setcount*** options you can make ***lookup*** write master records to the tertiary output that were actually matched precisely once or whatever your application might need.

Updating the Reference Table

One of the major restrictions of ***lookup*** used to be that the reference table was loaded at the start of the pipeline and could not be modified afterwards. When you are working with static data this is often not really a problem since you can ***buffer*** the detail records before passing them to ***lookup***. When you are working on real-time data (e.g. in some kind of server processing the commands when they come in) you just don't want to delay processing until all data is available. Some plumbers have found ways to make ***lookup*** terminate and reload its table again, but that process becomes pretty expensive when the table gets large.

In response to our pleas the Piper has enhanced ***lookup*** with several ways to change the reference table while ***lookup*** is running.

The AUTOADD option

When ***autoadd*** is specified, ***lookup*** automatically adds each detail record to the reference table when it is not found in the table.

The ***autoadd*** option is very neat for finding unique keys in a file, again without the need to reorder the file. The following pipeline lists the duplicate words in an an input file:

```
'PIPE < input file | lookup autoadd detail | > dup words a'
```

Each time a new key is encountered in a detail record, no match is found in the reference, so that record is not written to the primary output. It is then added to the reference, however, so if that key is encountered in a later detail record, that record will be written to the primary output, because it will match a master record.

If you would need to see the unique keys instead, you would use the secondary output rather than the primary (because the detail record does not match the first time). That pipeline looks like this:

```
'PIPE (end \)',
   '\ < input file ',
   '| l: lookup autoadd detail',
   '\ l:',
   '| > unique words a'
```

Some plumbers prefer to use the ***not*** stage in that case to swap the primary and secondary output:

```
'PIPE < input file | not lookup autoadd detail | > unique words a'
```

When you start to combine ***autoadd*** with the ***count*** option it may be annoying that the count is always one off (because the detail record is added after the first miss). The ***before*** option makes ***lookup*** add the record before matching it. This means that the count will be complete and that none of the detail records will be written to the secondary output.

When ***autoadd*** is used the input range for the detail record and the master record must be the same.

The ***keyonly*** keyword specifies that only the key of the record should be stored in the reference table (for example to save storage). Especially with ***autoadd*** this option is useful.

An elegant example from the Piper builds REXX stemmed variables from an input file like TCPIP DATA. The lines in that file look like this:

```
nsinteraddr   1.2.3.4
domainorigin  foo.bar
nsinteraddr   3.4.5.6
```

The first word in the record is a variable name, the remainder is the value of that variable. The same variable can occur more than once so we want the values to be stored in REXX stemmed variables. The output records are of the form "=name.i=value," where "i" is the numeric index of the stemmed variable. This format can be used as input for ***varload*** or ***varset***.

```
'callpipe (end \ name StemBuild)',
   '\ *: ',
   '| l: lookup count',              /* Dump masters with count     */
         'trackcount',              /* Master with current count   */
         'autoadd before',          /* Add master automatically    */
         'keyonly',                 /* Only need the key           */
         'w1',                      /* First word is the key       */
         'master detail',
   '| spec ,=, 1 11-* n ,., n',     /* Write =name.                */
         '1.10 strip n ,=, n',      /* .. and  sequence number     */
         'read w2-* n',             /* .. and the value            */
   '| f: faninany',
   '| *:',
   '\ l:,
   '\ l:',                          /* The dumped masters          */
   '| spec ,=, 1 11-* n ,.0=, n',   /* Write  =name.0=             */
         '1.10 strip n',            /* .. and the counter          */
   '| f:'
```

This pipeline uses the ***trackcount*** option to assign sequence numbers to each occurrence of the key (so we index the stemmed variable). The ***autoadd before*** causes each new key to be added before the matching the detail record so the sequence numbers will start at 1. The ***count*** option causes ***lookup*** to dump the contents of the reference table with their count value when it terminates. These records are used to set the ".0" variables that define the number of entries in the stemmed array.

Note: This subroutine pipeline is available as the ***stembuild*** built-in stage of *CMS Pipelines*.

The Tertiary Input Stream - Adding Masters

For more complicated dynamic updates of the reference table a tertiary input stream was added to ***lookup***. Records read on this input stream are added to the reference table while ***lookup*** is running.

Consider the case where you have a service machine that will process commands from authorised users. A ***lookup*** can be used to verify the originator against the list. To define additional authorised users I have used ***immcmd*** to set up an immediate command ADD. A userid specified on the immediate command will be passed to the tertiary input of ***lookup***.

```
'PIPE (end \ name RMHLUP.DOC:243)',
   '\ starmsg ',                      /* Read commands via SMSG      */
   '| not chop 8 ',                   /* Discard SMSG header         */
   '| x: lookup 1.8 detail ',         /* Check originator            */
   '| process ',                      /* Process the command         */
   '| cons ',
   '\ < auth users',                  /* Read authorised users       */
   '| x:',
   '| insert ,Rejected command: ,',   /* Identify unauthorised cmds  */
   '| cons',                          /* .. and display them         */
   '\ immcmd ADD',                    /* Set up immediate command    */
   '| spec w1 1.8 | xlate ',          /* Isolate userid & upper case */
   '| >> auth users a',               /* Update the file too         */
   '| x:'                            /* Add this user to table      */
```

You will notice that added authorised users are also appended to the disk file. Reading and writing the same file in the pipeline works here because ***lookup*** reads the entire file before it starts to process any detail records.

A service machine using ***lookup*** like this does not have to be stopped anymore to add authorised users, but typing the command from the console of the service machine is not always practical. A more logical solution would be to have an extra command that can be issued by authorised users.

```
'PIPE (end \ name RMHLUP.DOC:385)',
   '\ starmsg ',
   '| not chop 8 ',
   '| l: lookup 1.8 detail',
   '| add: pick anycase substr w1 of 9-*',
              '/== ,ADD,',           /* Divert ADD command          */
   '| process',                      /* Process others              */
   '| cons',
   '\ < auth users',                  /* List of authorised users    */
   '| l:',
   '\ add:',                          /* The ADD command             */
   '| spec substr w2 of 9-* 1.8',     /* Take userid as key          */
   '| xlate',                        /* .. in upper case            */
   '| copy',                         /* Consume one record          */
   '| >> auth users a',              /* Update file too             */
   '| l:'
```

What I have added to this pipeline is the ***pick*** stage to select the records that have the ADD command (after the userid that issued the command has been verified by ***lookup***). The input range ***substr w1 of 9-**** (i.e. the first word, counting from column 9) allows for a correct test without the need to modify the input record (since I want the others to pass to the processing stage unmodified).

The ADD commands that are identified by ***pick*** are passed through the ***spec*** stage to take the second word of the command.

The ***copy*** stage is needed here to prevent the pipeline from **stalling**. When your pipeline stalls it is important to understand why that happens, and fix it in the design of your pipeline. There is no general solution to prevent stalls in your pipeline, as some may want you to believe. In this particular case the stall is caused by the requirement that ***lookup*** wants the record on the primary output to be consumed before it can read one on the tertiary input. The ***copy*** stage provides a one-record buffer that can hold the record and allow ***lookup*** to complete writing the record and proceed to where it can read from the teriary input stream.

The Quarternary Input - Remove Masters

You will not be surprised to find that the Piper has also created a way to remove masters from the reference table. An additional fourth input stream is used for that. When a record is read on the fourth input stream all[2] reference records with the corresponding key are removed from the reference table.

Getting back to the example of the pipeline that processes commands from authorised users, you could add another ***pick*** stage to select DEL commands from authorised users and use that to remove authorised userids from the reference table. Instead of the two ***pick*** stages you can also use another ***lookup*** stage to separate the ADD and DEL commands from the rest.

Instead of working out such an example, I will show you one of my other favorite stages: ***deal***.

Combination with the ***deal*** stage makes for very interesting pipelines. In its basic form ***deal*** will pass the records to its output streams in a round-robin style like a card player deals cards. But ***deal*** can also pass the record to the stream identified by a field in the input record or by a record read from the secondary input stream.

I will now extend the example a bit by moving the command decoding outside the processing stage. Where the previous example used a ***pick*** stage to select the ADD command I now use a ***lookup*** to check the command.

```
'PIPE (end \ name RMHLUP.DOC:431)',
   '\ starmsg ',
   '| not chop 8 ',                    /* Remove prefix              */
   '| l1: lookup 1.8 detail'           /* Check authorised users     */
   '| l2: lookup anycase substr w1 of 9-* w1 master detail',
   '| d1: not deal',
   '| d2: deal secondary',
   '| process',
   '\ d1:',                            /* The master record          */
   '| spec w2 1',                      /* Just to stream number      */
   '| d2:',                            /* Secondary input for deal   */
```

The first ***deal*** in this example works in round robin fashion, but since I need it to start with the secondary output I use ***not deal*** that swaps the outputs of the stage.

The second ***deal*** has the ***secondary*** option specified so it will take the stream identifier from the secondary input. This secondary input is in fact derived from the records dealt

[2] The *all* applies to the situation where ***allmasters*** is specified. When ***allmasters*** is not specified there will be at most one reference record to be removed for a given key.

by the first ***deal*** stage. For each detail record that is read from the primary input, ***deal*** will write that record to the output stream indicated in the second word of these master records.

Assume the list of valid commands is in a separate file. All we have to do is add our two additional commands to modify the authorisation list.

```
'\ < command list',               /* List of supported commands  */
'| insert , 0, after',            /* Tag these with a 0          */
'| literal ADD 1',                /* Plus the ADD command        */
'| literal DEL 2',                /* .. and the DEL command      */
'| l2:',
```

The numbers with each command specify the output stream to be used by the second ***deal*** stage in the pipeline. The primary output of ***deal*** is already defined above (passing the records to the process stage).

```
'\ < auth users',
'| l1:',
'\ d2:',                          /* Secondary of deal:  ADDs    */
'| copy',
'| spec w2 of substr 9-* 1',      /* Take userid to be added     */
'| xlate',
'| l1:',                          /* Add this user to table      */
'\ d2:',                          /* Tertiary of deal: DELs      */
'| copy',
'| spec w2 of substr 9-* 1',      /* Take userid to be deleted   */
'| xlate',
'| l1:',                          /* Delete from table           */
```

In this example you may have missed the part that maintains the disk file AUTH USERS. Adding users to that file can be done by just appending lines to the file, but deleting userids does not work that way. A simple solution for this is to add the ***count*** option that will output the entire reference table with the number of hits when the ***lookup*** stage terminates (we don't need the number of hits, but otherwise ***lookup*** would only output the unmatched reference records). When the userid is forced off the system ***lookup*** would not be able to write those records. If this is a concern you could maintain a seperate file with users to be removed from the table and process that next time the pipeline is started.

Note: Unfortunately ***lookup*** does not release the storage for a reference record when that record is deleted from the table by passing its key to the quarternary input. This makes the simple approach outlined here a bit unpractical for large amounts of data (although I have built reference tables of several Megabytes). I will show possible solutions for that in the remainder of this paper.

The Other Output Streams

We have now seen the four input streams for ***lookup***, but in order to fully support updates of the reference table three extra output streams were added to ***lookup***.

Quarternary Output - Deleted Master Records

The master records that are removed from the reference table (by passing their key to the quarternary input) are written to this output stream according to the same rules that cause reference records to be written to the tertiary output when ***lookup*** terminates.

So when ***count*** is specified each deleted master record is written to this output stream, prefixed by its count value. When ***count*** is not specified the record will only be written if it was not referenced by a detail record.

These records may originate from the secondary or tertiary input of ***lookup***.

Note that the reference record is written to the quarternary output immediately when it is being removed from the table. Only the records that are still left in the reference table when ***lookup*** terminates will appear at the tertiary output.

The Quinary Output - Duplicate Masters

When ***allmasters*** is not specified, any duplicate master records (i.e. master records with the same key value) are not stored in the table. When the fifth output stream is defined and connected these rejected duplicate master records are written to this fifth output stream.

The Senary Output - Unmatched Deletes

When there is no reference record to delete for the key passed to the fourth input of ***lookup*** this delete is rejected, and the record is written to the sixth output stream, when connected.

What do you do with it?

Once you have mastered ***lookup*** you will find lots of places in your pipelines where ***lookup*** can simplify the plumbing. Often it can replace a pipeline that required the entire stream to be buffered (or sorted before use). This makes ***lookup*** very useful in servers that perform real-time processing of their input.

Feedback loop (autoadd)

In many cases you will need to use some kind of feedback in the pipeline that uses ***lookup***. The first example of such a feedback is when you write your own ***autoadd*** implementation. To do that you connect the secondary output (the unmatched detail records) with the tertiary input (reference to be added).

```
parse arg parm
'callpipe (end \ name RMHLUP.DOC:772)',
   '\ *:',
   '| x: lookup' parm ,
   '| *:',
   '\ x:',                              /* Take unmatch details        */
   '| f: fanout stop anyeof',
   '| copy ',
   '| x:',                              /* and pass a new masters      */
   '\ f:',
   '| *.output.1:'
```

Feedback loop (once)

When you know that you will only have a single detail record for each master record, it can be wise to delete the reference record from the table once it has matched a detail record (e.g. when the key is not fully unique). To do this you need a feedback from the primary output (the matched records) to the quarternary input (reference to be deleted).

```
parse arg rng .
'addpipe (end \ name RMHLUP.DOC:641)',
   '\ *: ',
   '| x: lookup' rng 'detail master', /* Both detail & master        */
   '| d: deal stop anyeof',           /* Separate detail & master    */
   '| *:',                           /* Matched detail records      */
   '\ x:',
   '| *.output.1:',                  /* Unmatched details           */
   '\ *.input.1:',                   /* Masters to be added         */
   '| x:',
   '\ d:',                           /* Matched master records      */
   '| copy',
   '| x:'                            /* To quarternary input        */
```

The advantage of such a feedback loop over the ***autoadd*** option is that it allows you to modify the record before passing it to ***lookup*** as a new master record (e.g. to add a sequence number). You probably should take care not to modify the key field (if you make it a duplicate key the master would be written to the quarternary output again).

Feedback loop (replace)

Another feedback might be to just replace the reference record when a second master record with the same key is received. Such a duplicate master record is normally rejected by ***lookup*** so we can use the records that appear on the fifth output to delete the old reference record and add the new one.

```
'callpipe (end \ name RMHLUP.DOC:661)',
   '\ *: ',
   '| o: not fanout stop anyeof',
   '| take last',                      /* Find EoF on input          */
   '| g: gate',                        /* .. to close the gate       */
   '\ o:',
   '| x: lookup' arg(1),
   '| *:',
   '\ x:',                            /* Secondary not connected    */
   '\ *.input.1:',                    /* New masters to be added    */
   '| a: fanintwo',                   /* Combine with feedback      */
   '| g:',
   '| x:',                            /* To tertiary input          */
   '\ d: fanin',
   '| x:',                            /* Quarternary stream         */
   '\ x:',                            /* Duplicate masters          */
   '| copy',                          /* Consume the record         */
   '| f: fanout',
   '| d:',                            /* Delete the old master      */
   '\ f:',
   '| a:'                             /* Add the new master         */
```

You will notice the ***fanout*** that provides two copies of the rejected master record. One copy is passed to the ***fanin*** to delete the old master, the other is passed to ***fanintwo*** to add it to the table. Because ***fanout*** strictly writes to its output streams in ascending order, we can be sure the old record is deleted before the new one is added. The ***fanintwo*** is needed to make sure the new key from the secondary input (that was rejected before) is preferred over one that might be waiting on the primary input.

Determine the Length of a Run

The following problem was suggested in the PIPELINE CFORUM on IBMLINK. The input stream contains (fixed length) records where part of the input record (say column 1.4) serves as a key. These records should be sorted such that the records with the most common key appear first in the output stream. The input stream was not sorted in any specific way.

The “classical” way to do this is to first sort the records on the key and then join all records with a common key. The more common a key is, the longer the record now will be. By prefixing these records with their length (***addrdw cms***) and sorting them again on that length field (in descending order) the most popular keys appear first. When the original records are re-built the stream is sorted as requested.

Using ***lookup*** you can do this in a very elegant[3] way.

---

[3] Unfortunately this solution turns out to be more expensive than the one using ***sort*** and ***join***, but I still think the example is pretty neat.

```
 ...
'| f: not fanout',
'| x: lookup count 1.4 allmasters',
'\ x:',                              /* No secondary input / output */
'\ f:',                              /* 3rd input: new masters      */
'| x:',                              /* 3rd output: masters & count */
'| sort 1.10 d',                     /* Sort on count, descending   */
'| not chop 10',                     /* And remove the count again  */
```

The solution uses the feature of ***lookup*** to add records to the reference table dynamically. The ***allmaster*** makes ***lookup*** also retain the masters with duplicate keys. Because of ***not fanout*** the record is added to the table before it is passed to the primary of ***lookup*** (to match against the table). The ***count*** option records the hits per key, so all masters with the same key carry the same count. When ***lookup*** terminates at EoF on the primary input it will output the records from the reference table on the tertiary output (the only one I have connected). They can be sorted on the count field to get the most common keys in front. The ***not chop*** then removes this count field again. The example above can also be very useful without the last two stages. In that case it produces a sorted list of the records with the count of the key prefixed to each record with that key. This is like a ***sort count*** without loosing the individual records. This means you can use the count field to determine what to do with the individual records (e.g. report the individual records or just report the count).

Compute Minidisk Starting Cylinder

You can use ***increment*** for more than just counting. When I had to compute a new allocation for my mini disks I found the following pipeline very useful. The objective was to compute the new starting cylinder for mini disks but keep them on the same volumes (i.e. remove the gaps between them).

The MDISK statements in the CP directory are as follows:

```
MDISK cuu devtype start size volser mode rpass wpass mpass
```

The following pipeline takes the entire CP directory as its primary input.

```
'callpipe (end \ name RMHLUP.DOC:590)',
   '\ *: ',
   '| m: zone w1 abbrev mdisk 2 anycase',      /* Select MDISK stmts */
   '| spec w5 1.10 r 1-* nw ',        /* Prefix with minidisk size   */
   '| l: lookup autoadd',
         'increment',                /* Increment value on details  */
         'trackcount',               /* Provide running total       */
         'w6',                       /* Match on volser             */
         'master detail',
   '| spec 1.10 1 read 1-* n',        /* Count from master & detail  */
   '| spec a: w1 - b: w6 -',          /* Running total & size        */
         'w2.3 1',                   /* Copy MDISK cuu & devtype    */
         'print a-b+1 nw.6 ri',      /* Computed start cyl          */
         'w6-* nw',                  /* Copy size, volser, etc      */
   '| f: faninany',                  /* Combine with unmodified     */
   '| *:',
   '\ m: | f:'                       /* The non-MDISK statements    */
```

There are a few things in this example that may need some additional explanation. When a detail record matches a reference record its increment value will already be added to the count value for that reference record. In the context of this example: the running total produced by TRACKCOUNT is the number of the last cylinder of this minidisk rather than the first cylinder. By substracting the size from the running total (the ***a-b*** in ***spec***) we get the starting cylinder. The ***+1*** is the correction to start allocations on cylinder 1 rather than 0. If allocation should not be from cylinder 1 but something different for each volser you could use SETCOUNT and pass the list of volsers to the secondary input of ***lookup*** and prefix each of those records with the address of the first free cylinder on that volume.

Such a list of volsers would also be used when you want to move the disks on some of the volumes only.

The fact that this pipeline does not delay the input records is a big advantage. We can now process the entire directory in a single pass without having to worry to get each MDISK statement back in the right USER entry again.

If needed for further processing you could also specify the COUNT option to have ***lookup*** output the address of the first free cylinder on its tertiary output (to be connected). This would show you how far the volumes were filled by the new allocations.

Writing a Cache Stage

I have wanted to write a cache or look-aside buffer for *CMS Pipelines* for a long time. Such a cache should provide a transparent way to store and retrieve the result of earlier processing. The ability to dynamically add records to the reference table allowed me to use ***lookup*** to do this.

Suppose you need to associate userids in your data with the name of the user, and that the name of the user is retrieved using the CMS command **namefind**. To avoid using **namefind** when the userid appears a second time you could first retrieve all names and store them somewhere (e.g. in a ***lookup*** stage) but this is not attractive when your names file contains a lot of users. What this following example does is retrieve the name when it is first needed and then remember it such that it can be used for subsequent queries.

The first thing in the pipeline is the ***lookup*** that searches for the userid in its reference table. If the userid is found the stored reference record can be written to the output of the pipeline and we are done.

```
/* NAMEFIND REXX      A Namefind with Memory                          */
'callpipe (end \ name NAMEFIND.REXX:4)',
   '\ *: ',
   '| pad 8',                          /* Make it 8 chars            */
   '| x: lookup 1.8 master ',          /* Find in table              */
   '| f: faninany ',
   '| *:',
```

If the userid is not in the reference table ***lookup*** will write the record to its secondary output. This record can be used to build the **namefind** command. The output of this command is used to create a new master record. Because the master records have both the userid and the name, we use a ***fanout*** and a ***spec*** stage to build such a record from the output of the **namefind** command.

```
   '\ x:',                            /* If no match found          */
   '| copy',                          /* Consume the record         */
   '| u: fanout',
   '| copy',                          /* Let the fanout run         */
   '| s: spec 1.8 1',                 /* Combine the userid         */
         'select 1 1-* n',            /* .. with the name           */
   '| b: fanout',                      /* One copy to                */
   '| x:',                            /* .. be added to the table   */
   '\ u:',
   '| spec ,NAMEFIND :USERID, 1',     /* Make NAMEFIND command      */
         '1-* nw ,:NAME, nw',         /* .. to get the name         */
   '| command',
   '| s:',                            /* Combine with userid again  */
```

Although the record was added to the reference table by the pipeline above, we still need to write the record to the primary output of this pipeline. This way the pipeline using this stage will not notice any difference whether the answer was obtained from the ***lookup*** or

by means of the **namefind** command (except of course in reduced consumption of CPU cycles).

```
   '\ b:',                              /* Another copy of user & name */
   '| f:'                               /* .. to the output            */
error: return rc * ( rc ¬= 12 )
```

Although this stage has two ***copy*** stages in it to consume the record, it is important to realize that the subroutine pipeline does not delay the record.

Note: A correct implementation should also deal with the situation where the userid is not found by **namefind**. In this example it will use the CMS error message instead of the name or cause a stall (when the EMSG setting is OFF). To fix this you would probably use the secondary output of ***command*** to check the return code of namefind.

A pipeline like this can be very useful when the process involved (in this case invoking the **namefind** command) is more expensive than maintaining the cache. However, the length of the key and the command for obtaining the result are contained in this pipeline stage, so you would need to design such a pipeline again for every situation where you need such a cache.

A generalized cache stage should make the actual resolving pipeline external to the cache and remove the need for specifying key length in the cache.

Generalized Cache

The resolving pipeline could be made external by specifying the processing pipeline as an argument to the cache (as for example the ***totarget*** and ***append***stages do) but it is more flexible to add an additional input and output stream as shown below.

```
   ...                                  /* Input records               */
  'p: pipcache',
   ...                                  /* Resolve the query           */
  'p:',
   ...                                  /* Use the result              */
```

In this topology the actual processing (invoking the **namefind** command in the previous example) is encapsulated between the primary output and secondary input of the cache. The records enter this pipeline at the top and either pass through the processing stage, or bypass that section, depending on whether the answer was in the cache or not.

To get the key length out of the cache stage I have stored the query (the userid in this example) and the answer (the name of the user) in separate ***lookup*** stages. A fixed length sequence number is now used to link the two ***lookup*** tables. This means we can connect those two ***lookup*** stages as follows.

```
'addpipe (end \ name PIPCACHE.REXX)',
   '\ *.input.0:',
   '| x1: lookup 1-* 11-* master',     /* Find seqnr for this query   */
   '| x2: lookup 1.10 master',         /* Find the answer for seqnr   */
   '| i: faninany',
   '| not chop 10 ',                   /* Remove seqnr                */
   '| *.output.1:',                    /* .. and to secondary output  */
```

When the query is not in the first ***lookup*** stage we need to obtain the answer (by running the query through the resolving process) and then add the proper record to both tables.

The unmatched detail record comes out of the secondary output of the first ***lookup*** stage (we start that with an empty table). The query is prefixed with the sequence number and a ***fanout*** passes one copy of the record to the tertiary input of ***lookup*** to add it to the reference table.

```
'\ x1:',                          /* Query not in table          */
'| spec number 1.10 ri 1-* n',    /* Assign seqnr                */
'| o1: fanout',
'| copy',                         /* Consume the record          */
'| x1:',                          /* Insert in table             */
'\ x2:',
```

The other copy of the query with sequence number goes here. The sequence number is separated from the query to let the pipeline between the primary output and the secondary input obtain the answer. That answer is then passed back to a ***spec*** stage to combine it with the sequence number again to add it to the reference table of the second ***lookup*** stage.

```
'\ o1:',
'| ch: chop 10',                   /* Separate seqnr & query      */
'| copy',
'| s: spec 1-* 1 select 1 1-* n',
'| o2: fanout',                    /* Seqnr & answer              */
'| x2:',                          /* To second lookup stage      */
'\ ch:',                          /* Just the query              */
'| *.output:',                    /* .. primary output           */
'\ *.input.1:',                   /* From secondary input        */
'| s:',                           /* Combine with seqnr          */
```

What remains now is to copy the answer (that was read from the secondary input) to the secondary output. This makes every answer appear at the secondary output, whether it was found in the table or not.

```
   '\ o2: | i:'
error: return rc * ( rc /= 12 )
```

Note: The pipeline above uses ***addpipe*** to assure that End-of-File will propagate through the pipeline.

The stage we now have is really general. The cache can be added to an existing pipeline without affecting its function. When the process for obtaining the answer is expensive and the same query is passed more than once, it might be useful to use the cache.

The cache presented here retains **all** query and answer combinations, there is no cleanup process involved that will remove entries from the tables when they are not needed anymore. This may be acceptable when the number of different records is limited by some other factor, but for many production type pipelines this may be a problem.

Stopping and Restarting Lookup

As explained before the ***lookup*** stage does not release the storage when reference records are deleted.

In this section I will show how to write a shell around ***lookup*** that allows ***lookup*** to be be stopped and restarted in a transparent[4] way to reload the reference table.

If the pipeline containing ***lookup*** needs to be restarted several times, the obvious solution is to have a sipping pipeline in a loop.

[4] Transparent here means that the process outside this stage does not have to know about the stopping and restarting. Records should flow just like when the ***lookup*** was not stopped.

```
signal on error
do forever
  'peekto'                              /* Already End-of-File?       */
  'callpipe (end \ name RMHLUP.DOC:829)',
     '\ fg: fanin ',
     '| g: gate ',
     '\ *: ',
     '| g: ',
     '| x: lookup ...'
     ...
end
error: return rc * ( rc /= 12 )
```

The pipeline above contains a ***gate*** that will break the primary input of ***lookup*** when it is triggered. When ***lookup*** receives End-of-File on the primary input it will terminate. Since the trigger for ***gate*** will need to come out of the pipeline, an additional ***fanin*** stage is used to allow for a connection upwards to the primary input of ***gate***.

When ***lookup*** terminates it will dump all remaining reference records to the tertiary output (provided we have used the ***count*** option). When the ***callpipe*** is invoked again in the next pass of the REXX loop we need to pass these reference records (or some of them) to the secondary input of ***lookup***. This means that we need a mechanism to hold the records between the two invocations of ***callpipe***. It would be possible to use a REXX array for that (using the ***stem*** stage) but the overhead of that is pretty much. A very elegant solution offered by the Piper is to use ***buffer 1*** in an affixed pipeline.

An affixed pipeline is one that takes its input from an output of the stage that issues the ***addpipe*** and that feeds its output into an input of that stage.

```
'addstream both buff'
'addpipe (end \ name RMHLUP.DOC:860)',
   '\ *.output.buff: ',
   '| buffer 1 //',
   '| *.input.buff:'
```

In its basic form ***buffer*** will read all records until End-of-File and then write them to its output. The optional numeric parameter for ***buffer*** is used to specify that ***buffer*** should read up to a null record and then output that number of copies (in this case 1) to the output.

By using ***buffer 1*** in an affixed pipeline we have a way to store the records between two ***callpipe*** calls. The null string specified on ***buffer*** causes a null record to be written between two buffers, so we know when ***lookup*** should stop reading reference records.

So writing the records into the buffer goes like this:

```
   ...
   '| append strliteral ',            /* Append a null record       */
   '| *.output.buff:'                 /* .. and write to buffer     */
```

Reading the records from the buffer (in the next cycle) is done like this:

```
   '\ *.input.buff:',                 /* Read from buffer           */
   '| t: totarget nlocate 1',         /* .. up to null record       */
   ...
   '\ t:',                            /* Take the null record       */
   '| drop'                           /* .. and read that too       */
```

The null record that was written between two buffers must be read as well, since that will free the ***buffer*** again for reading another buffer of data.

We now need to run a pipeline that is slightly different during the first pass of the loop. An easy way to do that is to use a REXX variable to hold part of the pipeline specification.

```
refs = '*.input.1:'
pass = 0
do forever
  'peekto'
  pass = pass + 1
  'callpipe (end \ name RMHLUP.DOC:968)',
     '\ *: ',
     '| l: lookup ',
     ...
     '\' refs,                        /* Initial reference records   */
     '| l:',
     ...
  if pass = 1 then refs = ,           /* Reference records for       */
     '  *.input.buff:',               /* all remaining passes        */
     '| uptoand nlocate 1',          /* Including null record       */
     '| drop last'                    /* .. and remove the null      */
end
```

Normal REXX processing will substitute the variable with its current value when the host command is composed. Thus, in the first pass the master records are read from the secondary input (***.input.1:***), while in later passes they are read from the output of the affixed pipeline (***.input.buff:***).

Note: The ***uptoand*** stage above is a little pipeline that selects the records up to and including the specified target.

```
'addpipe (end \ name UpToAnd.REXX)',
   '\ *: ',
   '| t: totarget' arg(1),           /* Select up to the target     */
   '| f: faninany ',
   '| *: ',
   '\ t: ',
   '| take 1 ',                      /* And take the target too     */
   '| f:'
```

You may want to have the ***lookup*** dump the contents of the reference table to the tertiary output during the last pass of the loop, to provide a fully transparent behaviour. Since we normally cannot predict which will be the last pass the simplest solution is to run an additional ***callpipe*** after the loop has ended to output the contents of the reference table.

```
do forever
  'peekto'
  if rc /= 0 then leave
  ... one cycle
end
'callpipe (end \ name RMHLUP.DOC:1015)',
   '\ *.input.buff:',               /* Read all from buffer        */
   '| *.output.2:'                  /* To my tertiary output       */
```

There is one thing still missing in the infrastructure of this pipeline. We need something to trigger ***gate*** that will cause the pipeline to terminate so we can start another pass. Since we want to restart the ***lookup*** to free storage occupied by deleted reference records, starting a new cycle is only useful when we have deleted a number of reference records.

```
   '\ *.input.3:',                  /* Quarternary input: deletes  */
   '| o3: fanout',
   '| drop 100',                    /* Allow for 100 deletes       */
   '| take',                        /* .. then take the next       */
   '| fg:',                         /* .. and trigger the gate     */
   '\ o3:',
   '| l:'                           /* To quarternary of lookup    */
```

The number 100 in the pipeline above is in fact the tuning of the whole mechanism. This is where storage (occupied by reference records that were deleted) is being traded against CPU cycles (to reload the remaining reference records). You will probably need to have additional knowledge about your data to determine when to flush the ***lookup*** stage. If the size of the master records varies a lot you might consider to specify the ***count*** option and use the quarternary output to actually count the number of bytes that would be released when the ***lookup*** stage is recycled.

You could probably write a generic "Self Cleaning" shell for ***lookup*** using the techniques introduced above (this was the reason I asked John Hartmann for the ***setcount*** option so I could carry the count value forward from one cycle to the other and provide an overall counter).

A Cache with LRU cleanup

To perform a cleanup of the cache shown before we would need to pass the key of the record to the quarternary input stream of both ***lookup*** stages. The big problem here is of course to determine which keys should be deleted from the table while ***lookup*** is running. Because ***lookup*** currently does not free the storage for reference records when they are deleted, I felt it was rather useless to do such a cleanup. Instead, I have chosen to just restart the pipeline containing the ***lookup*** now and then and reload only part of the reference records in the next invocation of ***lookup***.

In order to keep track of what reference records were used, I will use the counter in both ***lookup*** stages (since the pair of ***lookup*** stages needs to contain the same reference records I can use two counters). The counter in the first ***lookup*** stage will be used just to record whether the reference record was used in the last cycle; the other one will count the number of cycles that have passed since this record was last used (so we add 1 to it for each cycle in which the first ***lookup*** has a count of 0).

You will find the full cache with Least Recently Used implementation in the appendix of this paper.

Conclusion

I hope this paper shows you why I feel ***lookup*** really is the Plumber's Swiss Army Knife. Often the ***lookup*** can replace a pipeline that would normally require the records to be all available before processing can begin; this makes ***lookup*** very useful for writing servers that process the records in real time.

For "normal" pipelines ***lookup*** will in many cases reduce the amount of coding you have to do. In most situations you will find such a simple pipeline run faster than without using ***lookup***.

Appendix: Cache with LRU Cleanup

The following stage implements a cache similar to the other one in this paper, but with an additional process to remove entries from the cache to limit the amount of storage being consumed.

The normal way to remove unwanted reference records from a ***lookup*** stage is to present the key of the record at the quarternary input of the stage. Unfortunately *CMS Pipelines* does not free up the storage when a master record is being deleted, so this would not limit the storage requirements for the cache. This restriction makes it necessary to restart the ***lookup*** stages now and then to reload their reference tables and thus prevent fragmentation of free storage.

Note: A future version of *CMS Pipelines* is likely to have this problem fixed. When such a version is available it would be possible to implement a LRU algorithm by deleting individual reference records from the table. It is however not unlikely that such a version would include the LRU algorithm as built-in option of ***lookup***.

To stop and restart the ***lookup*** stages means that the ***callpipe*** containing these stages should be in a REXX loop. The overhead of restarting the ***lookup*** for each reference record to be deleted would be rather heavy, so we will need some extra parameters to tune the stage.

```
arg buffkeep buffdel .
if buffkeep = '' then buffkeep = 100
if buffdel = '' then buffdel = buffkeep%10
```

Since we would not get savings from deleting the reference records until the ***lookup*** is restarted, we can postpone removing these reference records until the ***lookup*** is restarted. When the reference records are dumped by ***lookup*** we will need to filter some least recently used records out (by default 10% of the cache size) before loading them in the next invocation of ***lookup***.

To show the statistics of the cache we define an extra output where one line per cycle will be written, when defined.

```
'maxstream output'
if rc >= 2 then stats = '*.output.2:'
else stats = 'hole'
```

As before, we will unload the reference (the master records) into a ***buffer*** in an affixed pipeline at the end of each cycle. In the next cycle, the reference will be reloaded from that affixed pipeline, with the least recently used records being discarded during the reloading.

```
'addstream both buff'
'addpipe (end \ name PIPRMH09.DOC:211)',
   '\ *.output.buff: ',
   '| buffer 1 //',
   '| *.input.buff:'
```

Two REXX variables ***refs1*** and ***refs2*** will hold the pipeline segment that feeds the reference records into each of the ***lookup*** stages. At the start of the REXX loop we set them to ***hole*** so we begin with an empty reference table.

```
signal on error
refs1 = 'hole'; refs2 = 'hole'
buffadd = buffkeep                    /* Let cache fill first        */
cntr = 0
do forever
  'peekto inp'
  call cycle                          /* Sip until we need to flush  */
```

After the first cycle we need to define the two REXX variables to hold the pipeline segment that reads the current reference records from the ***buffer*** in the affixed pipeline. The affixed pipeline will also output a null record after each set of reference records, so we can use a ***totarget nlocate 1*** to read up to the null record. The ***buffer*** will contain the reference records of both tables pairwise, as will be shown later. Therefore a ***deal*** can be used to guide the reference records to the correct ***lookup***.

Note: The first ***lookup*** uses the ***count*** option to keep track of the reference records used during that cycle. Since these counters need to start at zero we remove the count from the previous cycle with a ***not chop 10***.

```
  if refs1 = 'hole' then              /* Is this the first pass?     */
    do
      refs1 = ,
        '   *.input.buff:',           /* From affixed pipeline       */
        '| t: totarget nlocate 1',    /* Stop at first null record   */
        '| dx: not deal',             /* Spread over two streams     */
        '| not chop 10'               /* Remove count from detail    */
```

The null record needs to be consumed to make sure ***buffer*** will be ready to read the next block of records when the sipping ***callpipe*** is being flushed.

```
      refs2 = ,
        '  t:',                      /* The null record             */
        '| take',                    /* .. remove it                */
        '| hole',
        '\ dx:'                      /* Master copy for 2nd lookup  */
```

The REXX variable ***buffadd*** holds the number of records that can be added to the cache before we need to flush it. For any cylcle other than the first this value should be identical to the number of records we removed from the cache.

```
      buffadd = buffdel             /* Add as much as we delete    */
    end
end
error: return rc * ( rc ¬= 12 )
```

Following is the ***callpipe*** that processes a number of detail records until *buffadd* new records have been added to the cache. When that happens the ***gate*** cuts the primary input and makes the ***callpipe*** terminate so we can flush the cache.

The second ***gate*** is to detect End-of-File on the primary input (either because it was cut by the first ***gate*** or because we ran out of detail records). This ***gate*** will be used to close the secondary input of the cache.

```
Cycle:
  'callpipe (end \ name PIPRMH09.DOC:205)',
     '\ fgat: fanin ',              /* Feed trigger to the gate    */
     '| g: gate',                   /* Quit after buffadd records  */
     '\ *.input.0:',                /* Detail records from caller  */
     '| g:',                        /* Sever after buffadd records */
     '| ox: not fanout',            /* A copy to 2nd, then to 1st  */
     '| take last',                 /* Get the last record         */
     '| gx: gate',                 /* .. and sever secondary input */
     '\ ox:',                       /* Input from caller again     */
```

Next is the pair of ***lookup*** stages again. The first ***lookup*** uses the ***count*** option to see whether the reference record was matched during this cycle.

```
     '| l1: lookup count 1-* 11-* master ',
```

The second ***lookup*** will use the ***count*** field to keep track of the number of cycles this reference record was not referenced. The ***increment*** option defines the first 10 columns of the detail record to have the increment for the counter. This is set to 0 so we don't modify the count of the masters during the cycle. The ***setcount*** causes the initial value

for the counter to be taken from the first 10 columns of the reference record when it is fed into the secondary or tertiary input stream.

Note: Since we don't really count, it would have been possible to use part of the record itself for the counter. I have not done that because it complicates the ranges a bit.

```
'| spec ,0, 10 1-* n',           /* An increment of 0          */
'| l2: lookup setcount increment count 1.10 master',
'| not chop 10',                 /* Remove the seq nr          */
'| f2: faninany',                /* Resolved & answered        */
'| *.output.1:',                 /* Output the answers         */
```

The REXX variable *refs1* is used to connect the pipeline for the initial reference records.

```
'\' refs1,
'| l1:',                         /* Detail not found           */
```

The records that come out of the secondary output of ***lookup*** are the unmatched detail records. Using the ***number from*** option we assign a sequence number that is used as the key to link the records of both ***lookup*** stages.

By using the REXX variable *cntr* we make sure the number is unique. We also will use this number to determine the age of a record. This is important for fine tuning of the flushing algorithm.

```
'| spec number from' cntr+1 '1.10 ri 1-* n',
'| f1: fanout stop anyeof',      /* Divert copy for calculation */
'| copy',                        /* Unblock the fanout         */
'| l1:',                         /* Add to first lookup        */
```

The tertiary output is where the master records from the first ***lookup*** are dumped when the pipeline terminates. The ***count*** option gives the number of hits for that master in the first 10 colums. Since we only care about whether the master was referenced or not, we can reduce the count to either 0 or -1 (the -1 for when the record was not referenced). That number will then be used to update the counter for the records in the second ***lookup***.

The section below uses another ***lookup*** during termination of the pipeline[5]. The reference tables of the two ***lookup*** stages above are fed into this third ***lookup*** to be able to update the counters. There will be one detail for each master, we need the ***lookup*** since the reference tables are sorted differently in both.

The reference records of the second ***lookup*** have the time-to-live value in column 1.10. This is used as the initial value because of the ***setcount*** option. The ***specs*** stage prefixes the detail record with either 0 or -1; so we decrement the time-to-live value by 1 when the record was not referenced during this pass.

```
'| spec c: 1.10 - 11-* 11',
      'print c>0?0:-1 1.10 ri',
'| l3: lookup trackcount setcount increment 1.10 master detail',
```

The ***trackcount*** causes ***lookup*** to prefix the master record with the current count value. The ***specs*** below copies this value to the detail record as well so we can sort the whole stream on that modified time-to-live value.

```
'| spec 1-* 1 write 1.10 1 read 1-* n',
'| sort 1.20',                   /* Least used ones first      */
```

This means that we can now drop the required number of records to make some space in the cache. The null string is appended to signal the end of the buffer.

---

[5] This could have been done as well by sorting both streams and use a ***specs*** stage to compute the modified time-to-live. Since this is a paper about ***lookup*** I felt this solution was appropriate.

```
'| drop' buffdel*2,              /* And discard some           */
'| append strliteral //',       /* A null record              */
'| *.output.buff:',             /* To affixed buffer          */
```

The initial reference table of the second ***lookup*** is filled as defined in the REXX variable *refs2*.

```
'\' refs2,
'| l2:',                        /* 2nd input: reference       */
```

Another copy of the new record is passed to here. It is already prefixed by a sequence number. An initial time-to-live value is computed. By using *buffkeep/buffdel* it will move down in the LRU table by the same percentage as what we flush out of the cache at each cycle.

```
'\ f1:',                        /* Copy record not in tables  */
'| nr: chop 10',                /* Keep the number here       */
'| copy',
'| s: spec ,'buffkeep%buffdel', 1.10 r 1.10 n',
      'select 1 1-* n',         /* .. and add detail record   */
'| l2:',                        /* 3rd input: new masters     */
 ,                              /* 3rd output: masters & count */
'| l3:',                        /* 2nd input: initial masters */
```

Note: The pipeline segment above connects to the secondary input of the third ***lookup*** stage. No records flow into that ***lookup*** until the other two terminate and start dumping their master records.

The cache miss record is output to the primary output of the cache where it will be read by the external pipeline that computes the value that was not found in the cache. We need the ***gate*** to propage End-of-File correctly.

```
'\ nr:',                        /* The cache miss record      */
'| oo: fanout',                 /* Divert one copy to gate    */
'| *.output:',                  /* To primary output where the */
,                               /* other pipe resolves it     */
'\ oo:',
'| take last',                  /* Wait until all processed   */
'| og: gate',                   /* Sever the secondary input  */
'\ *.input.1:',                 /* Read the resolved value    */
'| og:',                       /* Allow to be severed        */
'| gx:',                       /* Allow to be severed        */
'| f3: not fanout',            /* One copy of the record to  */
'| s:',                        /* feed into the 2nd lookup   */
```

An extra copy of the output record is written to the secondary output of the cache (where all output records appear, whether they were found in the cache or not).

```
'\ f3:',                        /* Another copy to be written */
'| f2:',                        /* to the secondary output    */
```

A final copy of the newly added records is used to keep track of the cache filling up. Dropping the first *buffadd* records makes the gate trigger when the cache is full. The REXX variable *cntr* is then updated with the last sequence number.

```
    '\ f1:',
    '| drop' buffadd,
    '| take 1',
    '| chop 10',                          /* Only keep the sequence nr   */
    '| var cntr',                         /* .. and save it              */
    '| copy',
    '| fgat:'

return

error: return rc * ( rc ¬= 12 )
```

Note: Depending on the chosen tuning parameters the actual cache algorithm (the reloading of the reference table) can become pretty expensive. One factor that affects the overhead of the stage is the *buffdel* parameter. It should probably not be set below 10% of the buffer size since LRU is a rough assumption anyway about what should remain in the cache.