InmemLookup - Using Lua Tables for Efficient Retrieval of Low Volume Lookup Data in QlikView Expressor

Rev 1.0, 11-Dec-13

Overview

For situations in which lookup table volumes are relatively small (e.g. < 100,000 records), using in-memory Lua tables for efficiently storing and retrieving values by keys can be as much as 5 to 10 times faster than using a Lookup Table and Lookup Rule in QlikView Expressor dataflows. This document describes the implementation of a Datascript Module that supports the creation, persisting, reloading and usage of lookup tables based on an in-memory Datascript table. Attached to this posting is a .zip file that includes a Datascript Module that implements an in-memory lookup table and a document that describes its usage.

Installation

To install the InmemLookup module into your QlikView Expressor installation, extract the contents of InmemLookup.zip to a temporary directory and add the file **InmemLookup.lua** into the 'external' subfolder of the Deployment Package folder that contains the dataflows that will use the module.

Testing Your Installation

The InmemLookup module includes a built-in test function that you can use to verify that you have properly installed the module. To test the installation, perform the following steps:

1. Use "Start->All Programs->QlikView->expressor3->QlikView Expressor command prompt" to launch a command line shell conditioned for QlikView Expressor and enter the command: ***datascript***

2. Enter the following commands at the '>' prompt in the datascript interpreter (the text in bold is what you type and the remaining text is what should be output by a successful execution of the test):

```
> require('inmemlookup')
> InmemLookup.test(131071)
Loaded 131071 key value pairs into lookup table saved to TestInmemLookup.out
Finished comparing all keys and values from deserialized lookup with original
(preserialized) lookup
```

Sample Project

In addition to **InmemLookup.lua** module itself, the archive file also includes a QlikView Expressor workspace export file named **InmemLookupProject.zip** that contains a sample project with 3 dataflows demonstrating the use of the module. These dataflows in this project are as follows:

- **TestInmemLookup_Create**: A dataflow that uses a Write Custom operator to populate and then save to file an in-memory lookup table.

- **TestInmemLookup_Usage**: A dataflow that uses a Transform operator to retrieve and recreate the in-memory lookup table created by "TestInmemLookup_Create," subsequently using it to lookup values by key.

- **TestInmemLookup_Usage_ToFile**: This dataflow is similar to TestInmemLookup_Usage except that instead of ending with a "Trash" operator it ends with a Write File operator that writes the expanded records to a file for visual verification purposes

Use the "DESKTOP->Import Projects" menu item to import the sample project from the export file. Note that all 3 of the dataflows in the project should run successfully once you have either changed the folder path for the "ExpressordataFiles" file connection to a valid folder on your computer or create the directory: C:\ExpressordataFiles. Be sure to execute the **TestInmemLookup_Create** dataflow first in order to generate the in-memory lookup table before executing either of the other two dataflows.

Data Volume Limits to InmemLookup Tables

Please note that the InmemLookup implementation has an absolute limit of 131,071 keys, so be sure you stay below that limit. If your project attempts to store a greater number of keys, the key insert operation will raise an error message and loading the lookup table will fail and abort the dataflow.

Volume Exceeded Warning Message

The "New" constructor function for creating a new lookup table accepts a single, optional 'size' parameter. The key insertion logic will issue a one-time warning message to the log file on the first key insertion operation that reaches that number of unique keys stored (and whatever values go with them).

Multi-Part Keys

The InmemLookup tables accept either numeric or string scalar values as keys. If your lookup has a multi-part key, you will need to use a form of string concatenation to create a single, scalar string value to use for each key [e.g. string.concatenate(Department_ID, '|', Location_ID)].

Storing Records for Values

Often an element in a Datascript table stores a single scalar value, as in:

```
ColorCodeLookup[input.colorCode] = input.colorName
```

But there will be situations when you will want to store multiple field values for the same key. You can do this by assigning a nested Datascript table to each key, as in:

```
PostOfficeLookup[input.postalCode] =
        { City = input.PO_City,  State = input.PO_State}
```

When accessing these multiple fields in the lookup rules in dataflow operators, you can use a local variable to store the associative table reference before accessing each of the field values, as in:

```
local t = PostOfficeLookup[input.postalCode]
output.City = t.City
output.State = t.State
```

Handling Duplicate Keys

There may be cases in which you will need to record multiple values for the same key. This is possible with the InmemLookup module with a small amount of additional logic. That logic entails detecting the circumstance in which there is already a value for the key and using a nested array to store both the old value(s) and the new value.

Let's take as an example a lookup table that stores the prerequisite courses for each course offered by a university where the key is the course id and the value is either a single string value for the single prerequisite course or, in the case of multiple prerequisite courses, the value is an array of strings containing the prerequisite course ids. In this case, the lookup table creation logic might look as follows:

```
local coursePrereqs = {
  { Course = "Bio 101",  Prereq = "Bio 1" },
  { Course = "Chem 101",  Prereq = "Chem 1"},
  { Course = "Chem 101",  Prereq = "Calc 1"},                    -- multiple prerequisites for Chem 101
 }
 -- create new lookup table
 lkup = InmemLookup.New()
 -- populate the table.  Note that in a dataflow, each "coursePrereq" record would arrive in the form
 -- of the 'input' argument to either a "transform" function in a Transform operator or via the "write"
 -- function in a Write Custom operator.  In those cases the "for" loop wrapping would not be necessary.
 for i,v in ipairs(coursePrereqs) do
   local existingPrereq = lkup[v.Course]
   if existingPrereq then  -- already one or more existing prerequisites?
     if type(existingPrereq ) == 'table' then
       table.insert(existingPrereq, v.Prereq)  -- already 2 or more prereqs, so append this one to the table
     else
       lkup[v.Course] = { existingPrereq, v.Prereq}  -- must be the second prerequisite, make nested table
     end
```

```
  else
    lkup[v.Course] = v.Prereq    -- must be the first prerequisite for this course
  end
end
```

Similar logic would be needed in the rule using the “lkup” table to perform a lookup operation to detect whether a value returned from a lookup is either a single string or a nested table containing multiple values.

For example:

```
local course = "Chem 101"
prereq = lkup[course]
local description = string.format("Prereqs for %s:", course)
if type(prereq) == 'table' then
  -- handle multiple occurrences in the nest table here
  for i,v in ipairs(prereq) do
    description = string.concatenate(description, " ", v)
  end
else
  description = string.concatenate(description, " ", prereq)
end
print(description)
```

The print statement at the last statement would print the following message:

```
Prereqs for Chem 101: Chem 1 Calc 1
```

Many Readers, Exactly One Writer

The InmemLookup module supports multiple, simultaneous readers either from multiple concurrent operators doing lookup reads or multiple partitions on the same operator. Note that each operator or partition will recreate a full copy of the in-memory Datascript table on the call to “deserialize” since every partition thread maintains a separate state, each of which executes a separate “initialize” call when initializing the operator containing the lookup read logic.

There can only be one operator and one partition that loads and serializes the in-memory Expressor table to a file (not only would multiple writing partitions and multiple operators have their own different copy of the Expressor Datascript table, they would collide attempting to write the same lookup table serialization file with unpredictable results).

No Cycles in Tabular Values

For performance optimization reasons, the “serialize” method of the InmemLookup module does not contain any logic to protect against the possibility that a table stored as a value for any given key might contain values that are themselves tables that could potentially form a cyclic reference. Care must be

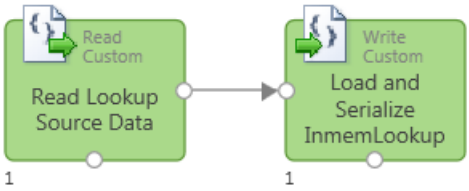
taken to ensure that tabular values stored do not contain cycles (they can contain arbitrarily deep hierarchies so long as they do not contain cycles).

Usage Details

This section provides more details on how to make use of the InmemLookup module.

Dataflow for Lookup Creation

The following dataflow provides an example of how to load a lookup table using the InmemLookup Datascript module.



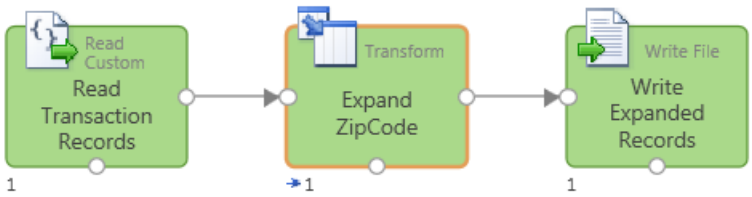
…and the Write Custom operator rule:

```
require('InmemLookup')

local poLookup
function initialize()
  poLookup = InmemLookup.New(99999)    -- Permit up to 99999 key value pairs before a
warning
end

function write(input)
  poLookup[input.ZipCode] = { City = input.PO_City, State = input.PO_State}
end

function finalize()
  poLookup:serialize('C:\\PostalCodeLookup.lua')  -- save the table for later usage
end
```

The dataflow above constructs a lookup table of records containing Post Office City and State names keyed by ZipCode and saves the lookup table to the file at C:\PostalCodeLookup.

Dataflow for Lookup Usage

The following dataflow provides an example dataflow that makes use of a previously created lookup table based on the InmemLoolkup module:



…and the transform operator rule:

```
require('InmemLookup')

local poLookUp

function initialize()
  poLookup = InmemLookup.New()
  -- restore the lookup data
  assert(poLookup:deserialize('C:\\PostalCodeLookup.Expressor Datascript'))
end


function transform(input)
  local t = poLookup[input.ZipCode]
  output.ShipToCity = t.PO_City
  output.ShipToState = t.PO_State
  return output
end
```