Kepler - Bug #3173

Improve data frame handling between RExpression actors

03/11/2008 08:36 PM - ben leinfelder

Status:	Resolved	**Start date:**	03/11/2008
Priority:	Normal	**Due date:**	
Assignee:	ben leinfelder	**% Done:**	0%
Category:	actors	**Estimated time:**	0.00 hour
Target version:	Unspecified	**Spent time:**	0.00 hour
Bugzilla-Id:	3173		

Description

The short version: true data types might not be preserved when using text file-based serialization of data frames between RExpression actors. Jim Regetz has provided some [detailed] suggestions....
...from Jim:

Hi all,

The sample workflow Kepler/R/demos/RExpression_Dataframe_Test.xml is broken for me. This demo is used in Example 10 in the Using R appendix in the current Kepler User Guide.

In the workflow, an actor named RExpression creates a dataframe and passes it to REpxression2. RExpression2 is supposed to extract a length-one vector named 'dframe' that contains the value 1.0. [Incidentally, the name 'dframe' is misleading here, insofar as this object is a vector rather than a dataframe.] The assigned value is then passed to a NonstrictTest actor, which in turn tests it against the value "1.0".

As expected, the workflow executes without reporting an error. However, this silent execution is misleading, as the test is not actually being performed. Note the following message in the terminal console from which I started Kepler:

Process :java.lang.UNIXProcess@180ba4b
Process complete: 0
Process :java.lang.UNIXProcess@813e58
Process complete: 0
Warning: '.RExpression_Dataframe_Test.Nonstrict Test' The test produced only 0 tokens, yet the correctValues parameter was expecting 1 tokens.
1396 ms. Memory: 89536K Free: 36252K (40%)

Moreover, the NonstrictTest actor also fails to report an error if the expression df2[1,1] is changed to df2[1,2] in RExpression2, even thought it clearly should.

The problem has to do with a subtle change in the storage mode of the data.frame columns resulting from the write.table/read.table cycle currently used to emulate the "passing" of data.frames between R actors (via files in the Kepler cache). The R statement:
df <- data.frame(c(1,2,3),c(4,5,6))
in the first RExpression actor produces a dataframe with columns of type 'double'. However, when this dataframe is written to the Kepler cache by RExpression using write.table, the ASCII representation in the file is integer-like (i.e., no decimal points). The read.table function in RExpression2 subsequently interprets and stores the columns as type 'integer'. The 'dframe' object is therefore an integer vector, which the dput function outputs in ASCII as "1L" rather than "1.0" (R uses this L notation to explicitly identify integers).

The simplest fix would be to explicitly coerce 'dframe' to a numeric vector in the R script, thus changing the underlying storage type back

to double. Adding the following line to the R script would work:
dframe <- as.numeric(dframe)

A slightly more general solution is to modify R statements embedded in the the RExpression java code, explicitly converting integer vectors to double before 'passing' them to an output port. Note that this would require testing the storage.mode, not just the mode (as is currently done in the actor). I believe the following modification would do the trick:

Replace this line:
r_out = r_out + "else {if (mode(x)=='numeric') dput(x)\n";

with these (apologies for line-wrapping):
r_out = r_out + "else {if (storage.mode(x)=='numeric') dput(x)\n";
r_out = r_out + "if (storage.mode(x)=='integer')
dput(as.numeric(x))\n";

This ensures that R integers and doubles will both appear the same on output ports (hmm, is that always desirable?).

However, this doesn't change the fact that the dataframe in the first actor is not actually identical to the dataframe in the second actor, even though the Kepler user might expect it to be. Often this won't be a big deal, but it may lead to some surprising and potentially nasty side-effects. Here are some examples, off the top of my head: * Integer-like columns of type 'double' end up as type 'integer' * Character columns become factor columns * Integer rownames become character rownames * Rounding error may occur if many significant digits * Any custom data.frame attributes will be lost

So, a bigger change would be to replace the write.table/read.table mechanism for passing dataframes with something else. I'm not sure to what extent Dan et al evaluated alternatives, but here are my thoughts:

```
Option A: Use dput/dget, which is already how Kepler deals with R
vector data. This will use an ASCII text representation specifically
designed for R objects. This is already what the actors use to output
vectors. Data types will be retained along with attribute information,
although rounding error can still occur. Downside is that it can be
slightly slower than read.table and write.table.

Option B: Use serialize/unserialize. This adds a tiny bit of
complexity in that file connections will need to be explicitly
opened/closed using R. However, saving an object in the Kepler cache as
a serialized binary will ensure that the object received by a subsequent
R actor is the same. In addition, this method is faster and has a
smaller storage footprint. On my machine, a dataframe of 50000 rows with
20 numeric columns is ~7.6MB in memory. The write.table function takes
over 2 seconds to complete, and produces an 18MB text file. The
serialize function completes in 1/10 sec, producing a file approx the
same size as the original R object itself. On the input side, read.table
is even slower, and can impose an additional memory hit of 3-4x the
object size during the read process; unserialization is still only 1/10
sec, and has no extra memory overhead.
```

I don't know how best to wrap this in the java code, but the pure R code for serializing an object 'x' in a file 'x.sav' is:

1. save 'x' as serialized object on disk
   conn <- file("x.sav", "wb") #open file for writing in binary mode
   serialize(x, conn)
   close(conn)

1. create 'x' from serialized object on disk
   conn <- file("x.sav", "rb") #open file for reading in binary mode
   x <- unserialize(conn)
   close(conn)

Obviously a downside to serializing in this way is that it is only

useful for passing data between R actors. But I would argue that a user who wants to pass data from an R actor to any other actor (whether R or not) using an generic tabular text representation should explicitly use write.table anyway, with full recognition of what that modality entails.
I should also point out that the R serialization functions are described as "experimental" in the documentation, but this primarily recommends against using the serialized objects for long-term storage; serving as an ephemeral bridge between one actor and the next, within a single workflow, strikes me as a totally different use case, and should not be a problem.

I have some other thoughts about the dput mechanism currently used to pass R vector data to ports, but I'll hold off on saying more about that for now.

Thanks,
Jim

_____
Kepler-dev mailing list
Kepler-dev@ecoinformatics.org
http://mercury.nceas.ucsb.edu/ecoinformatics/mailman/listinfo/kepler-dev

History

#1 - 03/12/2008 10:40 AM - ben leinfelder

tried with my system (running R v2.4.1) and it does work.
Jim pointed out that my version of R does not use "1L" to explicitly label values as integers.
Latest (2.6.2) uses this notation.

Kepler installer comes with R 2.4.0. Should think about updating that, and also making sure the Rexpression actor works with the latest R version.

#2 - 03/12/2008 11:57 PM - ben leinfelder

The R actor now uses the serialize/unserialize method described by jim for data frames and also for other complex R objects (the result of lm() for example).

The sample workflow mentioned in this bug, also works (due to previous changes to support floating point numbers)

#3 - 03/27/2013 02:22 PM - Redmine Admin

Original Bugzilla ID was 3173