Kepler - Bug #4764

ProvenanceRecorder.changeExecuted slow after workflow run

02/05/2010 06:19 PM - Oliver Soong

Status:	New	**Start date:**	02/05/2010
Priority:	Immediate	**Due date:**	
Assignee:	Daniel Crawl	**% Done:**	0%
Category:	provenance	**Estimated time:**	0.00 hour
Target version:	Unspecified	**Spent time:**	0.00 hour
Bugzilla-Id:	4764		

Description

If I run any of the tpc workflows (e.g., tpc09), any subsequent changes to Kepler (say changing workflow parameters) cause Java to peg one of my CPU cores. This includes canceling changes to RExpression. I've seen this behavior on Windows XP and 7. While I haven't seen it under linux or OS X, I haven't tested those as extensively. I have tried small test workflows, and haven't seen a particularly noticeable slowdown, so it may be related to the size of the workflow run. I have to restart Kepler to get things back up to speed, and it's bad enough that I'm actually restarting Kepler after every run.

I'm not sure it's a memory thing. java.exe is about maxed out on memory (~0.5 GB) in the Task Manager, but the Check System Settings window says I have 46% free. I was watching jstat, and changes don't seem to trigger a flurry of garbage collection.

History

#1 - 02/11/2010 12:12 PM - Oliver Soong

I profiled Kepler comparing an execution and an execution followed by a parameter change. First, I loaded tpc09 into a local repository and pre-cached all the data. I then restarted Kepler with profiling on, opened tpc09 from the repository, ran it, then closed Kepler. The restsarted Kepler again with profiling on, opened tpc09 again, ran it, changed a parameter, and closed it again. A couple of things jump out.

First, >50% of all processing is driven by java.net.SocketInputStream.socketRead0:
java.net.SocketInputStream.socketRead0(SocketInputStream.java:Unknown line)
java.net.SocketInputStream.read(SocketInputStream.java:129)
java.net.SocketInputStream.read(SocketInputStream.java:182)
java.io.DataInputStream.readInt(DataInputStream.java:370)
org.hsqldb.Result.read(<Unknown Source>:Unknown line)
org.hsqldb.ServerConnection.run(<Unknown Source>:Unknown line)
java.lang.Thread.run(Thread.java:619)

Second, the count for this call is nearly doubled when changing the parameter (78137 to 139903).

Since a lot of stuff relies on hsqldb, the high percentage may not be too surprising. However, the doubled count is why the program runs slow.

So far, this is triggered by an execution, seems to scale with workflow size, and involves something that makes calls to the hsqldb.

I'll attach the hprof data.

#4 - 02/11/2010 07:21 PM - Oliver Soong

I've tracked this down to the ProvenanceRecorder listener on the workflow NamedObj. After running tpc09, I change the targetYear parameter to 2005. This takes about 1:30-2 minutes, entirely spent in the ProvenanceRecorder.changeExecuted function. Technically, it's spent in 3 separate calls to the ProvenanceRecorder.changeExecuted, as the parameter value spawns a derivedFrom change and an entityId change. Nonetheless, this performance is bad enough and I think I have enough specificity to bump the priority.

#5 - 02/11/2010 09:51 PM - Oliver Soong

The function recordContainerContents is the cause of the slow behavior. It is called from _recordWorkflowContents which is called from changeExecuted but conditional on _haveExecuted, which is why it only occurs after a run. From what I can tell, the slow behavior is not due to anything particularly obvious, but the function is called ~3200-3400 times at an average cost of about 10ms.

I mentioned in the previous comment that it's called 3 times. It's actually called 4, but one is in the preinitialization step of the run.

My best guess is to look into SQLRecorder.java, and particularly at the functions registering entries with provenance.

#6 - 02/12/2010 08:43 AM - Daniel Crawl

Could you provide the workflow?

#7 - 02/12/2010 08:54 AM - Oliver Soong

It's in the Kruger SVN:
https://code.ecoinformatics.org/code/kruger/trunk/workflows/tpc09-plant-dynamics/tpc09-plant-dynamics-woody.kar

The problem is data access. On the other hand, if it's really just the _haveExecuted, then maybe that won't matter. If it looks like it does, then I think we need to bug Regetz.

#8 - 02/12/2010 03:26 PM - Daniel Crawl

If I run the workflow without accessing the data, and then change the targetYear, I only get 2 or 3 calls to changeExecuted. Can you reproduce without the data?

#9 - 02/12/2010 03:41 PM - Oliver Soong

That's correct and expected. recordContainerContents is initiated from changeExecuted, and it's recordContainerContents that is called thousands of times per call to changeExecuted.

I'm not convinced there's anything obviously wrong with the way this works, but perhaps we can think of ways to make this less of a hog. First, it's apparently alright for provenance recording to be delayed until the first run, so there doesn't seem to be an obvious reason why we have to make a full record for those 3 ChangeRequests that changing targetYear triggers. There are probably not-so-obvious reasons that I'm not aware of.

Second, I think a single provenance recording is checking and updating both the RegEntity cache and the provenance HSQLDB once per NamedObj (director, actor, port, parameter, relation, annotation, etc.) in the workflow, which in the case of tpc09 is apparently over 3000. Given the profiling results, I don't think the RegEntity cache/hash map is a problem, but I suspect we could make updating the provenance HSQLDB more efficient. My just-enough-to-be-dangerous understanding of DB stuff says they're usually optimized to operate in bulk, so perhaps it would be faster (but not necessarily easier) to build a list of entries in the workflow, make a single duplicates check query, then make a single update. There are probably reasons why this is a horrible idea, but I haven't thought of them yet.

#10 - 02/12/2010 04:04 PM - Oliver Soong

Forgot to answer the question. Sort of. I can reproduce the 3000+ function calls, but they seem to resolve faster, between 1-3 sec per set of ~3000 calls. It tried this with and without local repositories, and they're both the same, at about 1-3 sec. I doubt it has anything to do with the .kepler, since I've done clean-cache a bunch of times. Could it have to do with the KeplerData folder?

#11 - 02/12/2010 04:28 PM - Oliver Soong

So I wiped my .kepler and KeplerData, started Kepler (to initialize these folders), and things ran "fast" (1-3 sec per 3000+). I then copied the provenance folder from the old KeplerData over the fresh copy and ran, and things ran slow (30+ sec per 3000+). The "fast" provenance DB was about 8MB and the slow one was about 65MB. Scaling seems to be non-linear, although this isn't the best way to check timing. This also explains why the slowness seemed to be getting worse over time (but not fast enough for me to be sure of).

So it looks to me like the provenance DB bloats pretty quickly, slowing everything down. This is a little frustrating as an end user, but it seems like the code operates correctly. I'm willing to let this get triaged, as I personally don't have any qualms with blowing away KeplerData and .kepler frequently, but I believe this isn't how the design was envisioned.

I'm not entirely certain what provenance gets used for, but maybe certain entries can be culled? Alternatively, maybe we can allow the end user to flush selected content, a la wrm's ability to delete rows?

Since it no longer seems to be a Windows problem, I'm changing the platform for the bug.

#12 - 02/12/2010 05:22 PM - Daniel Crawl

Could I get a copy of the large db along with a list of steps performed to create it?

#13 - 02/12/2010 06:01 PM - Oliver Soong

I think it's too big to attach, so I've put it here:

http://www.nceas.ucsb.edu/~soong/kepler/provenance%20HSQLDB%20problem%20-%20Kepler%20folders.zip

Both KeplerData & .kepler are in there, but I've had problems running with a straight drop-and-replace (I think something in the ontologies configuration gets cranky). You might be able to swap enough parts, though.

Basically, I just used Kepler a lot. I can't recall exactly what I did or when this iteration of KeplerData was first created, but the general approach tends to be (Run->Modify->Save) xN. There are 23 executions in my wrm. There were probably a couple of clean-cache steps in there, and I might not have wiped KeplerData between updates.

#14 - 02/17/2010 10:03 AM - Oliver Soong

I finally remembered that I can look directly at the provenance db while Kepler's running. I just deleted all the runs, and some content got deleted, but not all. The ones that still have content are actor, data, director, entity, parameter, port, workflow, and workflow_change. I'm not sure if they're supposed to stick around or not.

The stuff that sticks around appears to be what causes the problem, though, as I still get the slowness after clearing the wrm and running once.

#15 - 03/27/2013 02:28 PM - Redmine Admin

Original Bugzilla ID was 4764

Files

java.hprof, no change.txt	152 KB	02/11/2010	Oliver Soong
java.hprof, changes.txt	171 KB	02/11/2010	Oliver Soong