Database - Bug #7334

Reclaim used sessions to improve performance

05/08/2023 10:12 AM - Alexandru Lungu

Status:	Test	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Dănuț Filimon	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**case_num:**	
vendor_id:	GCD	**version:**	

Description

Related issues:

Related to Database - Bug #7167: Associating records from opened buffers to n...	**Closed**

History

#1 - 05/08/2023 10:25 AM - Alexandru Lungu

- Assignee set to Dănuț Filimon

- Status changed from New to WIP

This task is meant to solve #7167, #6820, and #4021 at once.

Please refer to #7167 for the initial problem. Here we will evaluate the performance of a solution of reclaiming sessions in a limited interval of time:

```
AFAIK, this doesn't happen for temp database - only for the persistent one. I thought of having some sort of e
xpire time for session after use. Basically, allow each session to be reclaimed, but only within a limited tim
e threshold. This way, we allow caches to be persisted across session uses and buffers shouldn't be "revalidat
ed".

Having this kind of time threshold sounds right. We leave a reclaim time of ~1s. If the session is reclaimed,
is due to a compound business logic operation (logic A is invoked and logic B is invoked right after - they ca
n share the same session as A and B compose a single logical operation). If it is not reclaimed, it means that
 there is some sort of user input waiting in between (logic A is invoked and user input is awaited. at some po
int logic B is invoked - they doesn't need to share the same session as they represent different logical opera
tions).

If this was for _temp database, the solution would have been straight-forward. Talking about multi-user persis
tent databases, I am not 100% sure we can reclaim sessions that easily without breaking something - I may be w
rong. Please advice on this approach.
```

For a quick test, please create a SessionFactory which is responsible of creating sessions. Creating a session means SessionFactory.create(). This would either create a new session or reclaim the last session used by the requiring context, if still in the allocated time frame. For this matter, we can make a Session.makeReclaimable(SessionFactory factory).

Closing a session means session.close(), but this won't completely close the session or connection if it is marked as reclaimable. It will be only marked as a "reclaimable" session for its SessionFactory. Note that we should trigger its closing after the timeframe expires at some point.

As there are lots of session instances created, we should allow bypassing SessionFactory. Use it only with Persistence$Context.getSession. Other session lifecycles can still use constructor / close.

I am not sure of the ideal time frame value (1 second maybe). I am also thinking of not thrashing a session, so maybe we can reclaim one session only 5 times?

#2 - 05/09/2023 03:53 AM - Alexandru Lungu

Alexandru Lungu wrote:

> Basically, allow each session to be reclaimed, but only within a limited time threshold.

I put some thought on this and it isn't that trivial. There is not obvious moment when we can check that this time threshold is passed so we close the session and connection. The best we can do here is to create a separate daemon thread which does the clean-up. Once per second (or more?) it can check if there is an orphan session hanging, so it closes it. However, I don't immediately understand the time overhead of such separate thread. I would like a single thread to manage this cross-session, so we don't have one daemon thread for each FWD user. Hopefully the synchronization won't hurt that much here.

For to start, I think we can bypass this time threshold and just let the sessions be reclaimed without limitation. But bear in mind that a time threshold mechanism should be easily added afterwards.

Danut, please go ahead with a basic implementation of a SessionFactory. It should reclaim the existing session if exists or create another one if none exists. Closing a session means only "markReclaimable", without actually closing it. Please take care about the multi-context environment - let one SessionFactory per Persistence.Context for now. **Ensure you do the pre-closing work (rollback orphaned transactions), without actually closing (connection close).**

Finally, please note that Eric tried this at some point (in #4021) and didn't got encouraging results. Please double check your implementation and do some debugging / VisualVM / memory dump kind of work after, to check if this solution is OK.

#3 - 05/11/2023 10:16 AM - Dănuț Filimon

I created a SessionFactory which allows a Session to be reclaimed. I did a round of testing which mimic the specifications mentioned in #7167-1.

A profiling test where 1.7k session are created got the following results:

	Old (ms)	New (ms)	Difference
Average of 5	25650.8	12081.2	-52.9012%

Another test using VisualVM shows the following difference between creating a new session and reclaiming a "closed" session:

		Invocation count (1)		Invocation count (2)	
	15	17	33	35	-9

	202	36886	.8	22	9.7776
	68.9	1707	0.241	2	-99.6502
	993	8532148	775	8532148	-21.9536
	951	1707	4.9	1707	-99.4847
	251	1705542	1378	1705542	+449.0039

If there are any other methods that should be mentioned, please tell me so that I can update it with the snapshot data. The performance of reclaimable cache is very good, currently only one session can be reclaimed since the context holds one session per database.

#4 - 05/12/2023 03:44 AM - Alexandru Lungu

- % Done changed from 0 to 50

I will test/profile this as it is now. However, I am further on interested in:

- a multi-user environment test. Do your same profiling test with 1.7k sessions in parallel with 10 users (or more if you can). I am curious if the database connection trashing had an impact before and now is resolved.
- a method to close this session at some point. Check if you can smoothly integrate a daemon thread which checks the life-span of a session and closes it if not already reclaimed.
    - you will need some good tests here. ensure you don't introduce overhead with too much synchronization
    - check the performance with different life-spans (1-5-10s) Also, you need to adapt your test to allow such time-out.
    - the daemon thread should be cross-session (I think), to avoid spawning one thread per context. You need to register each reclaimable session to this daemon manually. Avoid registering the same session multiple times. Make sure you do this only for certain databases (you should see that _temp database for instance is not subject to such open/close sessions).
- redo the multi-user environment (with some sort of sleep to allow some sessions to expire) after you implement the daemon thread

#5 - 05/12/2023 04:45 AM - Dănuț Filimon

- % Done changed from 50 to 0

Committed 7334a/rev.14567. Added the initial implementation for reclaimable sessions.

Alexandru Lungu wrote:

> I will test/profile this as it is now. However, I am further on interested in:
>
> - a multi-user environment test. Do your same profiling test with 1.7k sessions in parallel with 10 users (or more if you can). I am curious if the database connection trashing had an impact before and now is resolved.
> - a method to close this session at some point. ...

I will continue testing this case in a multi-user environment and then look into the other points mentioned.

#6 - 05/12/2023 04:46 AM - Dănuț Filimon

- % Done changed from 0 to 50

#7 - 05/15/2023 09:05 AM - Dănuț Filimon

I remodeled the profiling test to fit a multi-user case. I never detailed what the contents of test are, only that it mimics #7176-1, so I am going to write them down now, including the modifications included in the remodeling.

The profiling test before:

- Creates 1k records in a persistent table.

- Creates 1k records in a temporary table.
- Using another temporary table, it stores 20 non-empty and 150 empty handles of the persistent table and 1350 empty handles of the temporary table with 1k records.
- Run an external procedure 1.7k times.
- Deletes the records from the persistent table.

The multi-user profiling test:

- The persistent table has an initial number of 1k records.
- Creates 1k records in a temporary table.
- Using another temporary table, it stores 10 non-empty and 20 empty handles of the persistent table and 100 empty handles of the temporary table with 1k records.
- Run an external procedure 200 times.
- No records are deleted from the persistent table.

For each number of clients, the test was executed 5 times. The average results for each client were used to calculate another average based on the number of clients. Here are the results:

Type	7334a.rev14566 (ms)	7334a.rev14567 (ms)	Difference
1 client	4903	3019.2	-38.4213%
2 clients	6058.5	5809	-4.1181%
5 clients	10940.68	10650.08	-2.6561%
8 clients	20668.325	19562.625	-5.3497%

I also tested a persistent table with an initial number of 10k records. At the same time, I raised the number of records created in the temporary table from 1k to 10k.

Type	7334a.rev14566 (ms)	7334a.rev14567 (ms)	Difference
1 client	34924.8	35275.8	+1.005%
2 clients	39203.5	38290.6	-2.3286%
5 clients	90447.6	92364.84	+2.1197%
8 clients	181751.325	188737.325	+3.8437%

The difference in both test sets is not impressive (**-5% - +5%**) and the only improvement is of **38%** when testing a single client (case that was already tested in #7334-3). I will retest after I improve the implementation.

#8 - 05/17/2023 07:31 AM - Dănuț Filimon

I added the Daemon thread for closing session based on a lifespan value. I also tested different lifespan values as suggested (1s, 5s and 10s). I used the same multi-user test with 1,2,5 and 8 clients. I couldn't test more than 8 clients because it would only open 8 and the rest would start after any of the other clients are closed. The test use the average of a total of 5 runs, but in cases where a run takes longer than 2-3 minutes, I only did 3 runs.

Note that the client executes a procedure 200 times, and every 20 times it is executed it also **pauses**.

Lifespan 1s, Pause 1.5s:

Type	Pause time (Total s)	7334a.rev14566 without Pause (ms)	7334a.rev14568 without Pause (ms)	Difference
1 client	30	4911	3257.4	-33.6713%
2 clients	30	5749.2	5793.3	+0.767%
5 clients	30	8440.92	8009.96	-5.1056%
8 clients	30	9773.825	10372.525	+6.1255%

Lifespan 5s, Pause 6s:

Type	Pause time (Total s)	7334a.rev14566 without Pause (ms)	7334a.rev14568 without Pause (ms)	Difference
1 client	120	4901.333	3111	-36.5274%
2 clients	120	5263.333	4920.5	-6.5136%
5 clients	120	8044.6	8036.93	-0.0953%
8 clients	120	9531.833	9196.041	-3.5228%

Lifespan 10s, Pause 11s. I tested 8 clients and decided not to test it further after comparing the results. If it's necessary, I will run the test using 1,2 and 5 clients and update this table.

Type	Pause time (Total s)	7334a.rev14566 without Pause (ms)	7334a.rev14568 without Pause (ms)	Difference
8 clients	220	8752.166	9761.25	+11.5295%

I think the most appropriate value for the lifespan of a session should be between 3-5 seconds. Please let me know if I should try testing using 3 as the lifespan of the session. I also did not prevent registering the same session multiple times because I didn't want to influence the results of the lifespan with a value that is too small or too large. Alexandru, what values should be considered in this case?

EDIT: Test specifications related to pause calls.

#9 - 05/17/2023 08:14 AM - Alexandru Lungu

Danut, please commit your latest changes to 7334a. I will do a quick review.

#10 - 05/17/2023 08:53 AM - Dănuț Filimon

Committed 7334a.rev14568. This update allows managing of expired session across all SessionFactory instances using a Daemon Thread.

#11 - 05/17/2023 10:06 AM - Alexandru Lungu

Review of 7334a.rev14568

- Make initialize() method synchronized to avoid concurrent initalization of the thread.
- This two step hasReclaimable and factory.create is not safe from thread's perspective. You may incorrectly recieve reclaimed as true, even if the create will generate a new session instead. I think it is best to use an "initialization lambda" here. Basically, pass in a lambda (Consumer<Session>), which does the whole associate code after. This will be called by the factory if the session is not reclaimed and will do the whole associating stuff. Avoid synchronizing this lambda call as it is huge. Let the Context do only the bufferManager.registerPersistenceContext after create. Anyway, you should get rid of hasReclaimable.
- Use extends Thread on a separate line.
- You need to evict these factories at some point. When the context is closed, you need to remove the factory from the thread. In otherr words, when a user disconnects, you will need to remove its session facory from the thread. This can be done in Context.cleanup. Ensure you also close the reclaimable session (if any) on cleanup.
- Note that you iterate the factories with a for-each. If you add/remove factories in that time, you may face a concurrent modification exception. Do a for (SessionFactoty factory : new ArrayList(factories)) (or something similar) to iterate a clone of the array.

#12 - 05/18/2023 03:42 AM - Alexandru Lungu

Danut, can you retrieve the same statistics as in #7334-8 without any pause? I am curios how fast is using a single session in one single demanding run.

#13 - 05/18/2023 05:54 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Danut, can you retrieve the same statistics as in #7334-8 without any pause?

After retesting 7334 revisions 14566, 14567 and 14568 using a lifespan of 1 second for reclaimable sessions, I obtained the following:

	7334a.rev14566 (	7334a.rev14567 (	7334a.rev14568 (
Average o	29865	16349.8	16510.2

f 5			

The difference between rev.14566 and rev.14567 is **-45.25%**. And the one between rev.14567 and rev.14568 is ~**1%**. There is not much change.

#14 - 05/18/2023 07:03 AM - Alexandru Lungu

What about multiple client? I suspect your test was done with only one client?

#15 - 05/18/2023 10:02 AM - Dănuț Filimon

Committed 7334a.rev14569. I applied the changes mentioned in #7334-11.

I misunderstood your question and retested using rev.14569. I modified the test in #7334-8, now each client runs the external procedure 1.5k times instead of 200 times. Here are the new results:

Type	7334a.rev 14566 (ms)	7334a.rev 14569 Lifespan 1s (ms)	Difference (1)	7334a.rev 14569 Lifespan 5s (ms)	Difference (2)	7334a.rev 14569 Lifespan 10s (ms)	Difference (3)
1 client	27211	14847	-45.4375%	15159.66	-44.2884%	14670.33	-46.0867%
2 clients	30870	31541.166	+2.1741%	32199.33	+4.3062%	32154	+4.1593%
5 clients	68086	69101	+1.4907%	69382.533	+1.9042%	68330.4	+0.3589%
8 clients	136871	142547.708	+4.1474%	139342.416	+1.8056%	140138.083	+2.3869%

The implementation shows an improvement for 1 client, but for multiple clients it does not. Since the results are obtained from an average, I observed that in the case of a **5 seconds lifespan**, the runs have a minimum difference of **6s** and a maximum of **17s** between each other when running **8 clients** and an average of **3s** when running **5 clients**. This does not happen when using a **lifespan of 10s**.

#16 - 05/22/2023 02:37 AM - Alexandru Lungu

I think we should insist on the multi-user test-case. We should see why we don't have performance improvements there.
Danut, please reduce your test-case: only some active-buffers (2?), only 3 session reclaims per user and a large lifespan (10 min). Run with 2 users and debug.

- Double check that each user uses its own session (they have different sessions and connections) and they both reclaim their own session each time needed.
- Check if any client is ever waiting for another client.
- Check the behavior of the session closing thread; does it block any client?
- Other synchronization issues.

#17 - 05/22/2023 03:13 AM - Alexandru Lungu

Review of 7334a

- Place the opening brace of PersistenceConsumer on the next line.
- I think you can make the code more consistent if you mark expire as synchronized. If the factory is already synchronized, this will be a no-op. However, if somebody uses expire out of context, make sure it uses it properly. Set reclaimable and timestamp to null in finally clause.
- Do a safe check in markReclaimable: if reclaimable is not null, then log a severe and make sure you expire the current reclaimable. You can make the method synchronized, instead of syncrhonizing this. Run a large application and make sure the warning doesn't pop up.
- Delete hasReclaimable
- At lambda definition, wrap the parameter currentSession in parenthesis

#18 - 05/22/2023 06:11 AM - Alexandru Lungu

Rebased 7334 to latest trunk. 7334a is now at rev. 14578.

#19 - 05/23/2023 07:56 AM - Dănuț Filimon

Committed 7334a/rev.14579. Applied the changes mentioned in #7334-17.

#20 - 05/25/2023 04:52 AM - Alexandru Lungu

Danut, as discussed, your tests may be spoiled by the fact that each user is doing lots of small transactions on the same table. Even if NO-LOCK, this still hurts the concurrency of the test-case so that the reclaiming is not that visible in the testcases.

Please use two users and an extra database table named seq with one integer field. It should have a dump with only one row (0). The test starts by each user increasing the first row with EXCLUSIVE-LOCK. This way, the first user gets an even number and the second user gets an odd number. After, each user will make its own scenario. The even user will do transactions on the A table, while the second user will do transactions on the B table. This way, we shall se only the time difference of reclaiming sessions, no locking the same table.

You can do the same test for 5 users (use mod 5, not even/odd) or 8 users (use mod 8, not even/odd), bu t you need more tables.

From my point of view, this is a parallel issue in which multiple users with NO-LOCK on a single table are still slowing each other own. This should be investigated separately (InMemoryLock).

#21 - 05/31/2023 03:53 AM - Alexandru Lungu

Rebased 7334a to trunk / rev. 14592 7334a is now at rev. 14596

I am doing a test/profiling round with the current changes.

#22 - 05/31/2023 11:09 AM - Alexandru Lungu

- % Done changed from 50 to 80

The changes are really good, providing a -2.7% performance increase. I didn't found any issues while testing.
Danut, please focus on finishing the tests here. I will like to do some last checks and merge this in trunk.

#23 - 06/01/2023 04:03 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Danut, please focus on finishing the tests here. I will like to do some last checks and merge this in trunk.

I remade the multi-user test set after finding out that they locked the same table for a simple record reading. I used what you suggested in #7334-20, tested 2, 5 and 8 clients that access different tables and are routed using another persistent table with EXCLUSIVE-LOCK, here are the results:

Type	No changes (ms)	Reclaimable session - Lifespan 1s (ms)	Difference (%)
2 clients	5930.7	4407.4	-25.684%
5 clients	16070.24	14926.56	-7.116%
8 clients	32013.725	31074.7	-2.933%

Note that the new test is based on #7334-7 multi-user profiling test with the only change being the number of records created in the temporary table which is 10k. Should we consider increasing the lifespan value or leave it as it is?

#24 - 06/01/2023 04:35 AM - Alexandru Lungu

- Status changed from WIP to Review

- % Done changed from 80 to 100

I am quite pleased with your results and my performance tests (-2.7%).

However, as this is quite a change on the session managing layer in FWD, I would like Eric to have a second pair of eyes here. **I know he attempting something similar previously in #4021.**

Edit: It is about 7334a /rev. 14596.

#25 - 06/23/2023 03:19 AM - Alexandru Lungu

Eric, Constantin, Ovidiu: can you provide a feedback here. The performance tests show real good results and the implementation looks quite robust. I intend to merge 7334a asap. However, I am not sure of the session time-out policy. Right now, a session expires (and is closed) after 1s of inactivity:

- Should we change the policy to 1s of life-time instead of 1s of inactivity? I am 90% sure that 1s of inactivity is the right policy here.
- Should we make this configurable? This way we can increase time-out period - or even disable it.

#26 - 06/23/2023 03:55 AM - Constantin Asofiei

Alexandru, there are some concurrency issues, but my main concern is this:

```
      synchronized (this)
      {
         if (reclaimable != null)
         {
            session = reclaimable;
            reclaimable = null;
            timestamp = null;
            return session;
         }
      }
      session = new Session(database, dmoVersion);
      initialize.accept(session);
```

If there is no 'reclaimable' session, then the initialize.accept does all the work to associate DMOs with the session. But, otherwise, the session is reclaimed and the DMO state (at session and DmoVersioning) remains unchanged. I think the session cache is OK, but what's the impact on DmoVersioning?

#27 - 06/23/2023 04:35 AM - Alexandru Lungu

> Alexandru, there are some concurrency issues

I will do a second check then. Please let me know if you already spotted something of interest to fix asap. Anyway, this is actually good news; the time may be even better than -2.7% after fixing the issues :)

> But, otherwise, the session is reclaimed and the DMO state (at session and DmoVersioning) remains unchanged

From my POV, **unchanged** is good for this case. I find the #7334 feature similar to "the legacy code doesn't trigger persistence actions for a while, so the Session and DmoVersioning should be unchanged".

I (somehow) see your point. DmoVersioning will not have anything evicted unless the session **expires**. As long as the session reaches the reclaimable state (not in use, but not expired), all cached records will stay in place both in Session and DmoVersioning. The side-effect on DmoVersioning is that records will be de-registered way later than now, so they will be stored in the DmoVersioning for more time. However, the session will be still linked to the DmoVersioing under the hood, even if not in use.

However, this is quite dependent on the time-out value. The intent here is to use the same session on two scenarios that happen very fast one after another. If the use-case triggers more than 1 work-flow through the FWD persistent layer, we shall execute them using the same database session. This being said, **I think** we can lower the time threshold even to something like 10ms and still re-use the session between persistence actions.

#28 - 06/23/2023 05:05 AM - Constantin Asofiei

Alexandru, about the concurrency issues:

- This code can execute on the SessionCloseThread. This is not an AssociatedThread (and can't be as there is only one thread) and has no access to context-local data. Any problem during reclaimable.close(); will execute DBUtils.handleException in this thread, which is not OK:

```
   public synchronized void expire()
   {
      try
      {
         reclaimable.close();
      }
      catch (PersistenceException exc)
      {
         DBUtils.handleException(reclaimable.getDatabase(), exc);
      }
      finally
      {
         reclaimable = null;
         timestamp = null;
      }
   }
```

- in this code, you need to move initialized = true before the new SessionCloseThread().start();. There is no guarantee the thread will start **before** the initialize() finishes.

```
      public synchronized static void initialize()
      {
         if (!initialized)
         {
            new SessionCloseThread().start();
            initialized = true;
         }
      }
```

- otherwise, please remove the history numbers and leave only one (the username and date of the change can remain, but we want to keep only one history number per branch).

#29 - 06/23/2023 05:09 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Alexandru, about the concurrency issues:

Danut, please do the fixing.

> otherwise, please remove the history numbers and leave only one (the username and date of the change can remain, but we want to keep only one history number per branch).

Sorry for that. I think is the third time you mentioned it :/ Trying to be more rigorous.

#30 - 06/23/2023 05:12 AM - Constantin Asofiei

I'm not sure how we can solve this:

```
     catch (PersistenceException exc)
      {
         DBUtils.handleException(reclaimable.getDatabase(), exc);
      }
```

if it executes on the SessionCloseThread. We need tests and make sure the Conversation thread for that session gets notified and the exception (and messages) gets processed. Which can be either a DatabaseConnectionException or a StopConditionException, plus the ErrorManager.display calls.

> The intent here is to use the same session on two scenarios that happen very fast one after another.

Think about in reverse: what happens if the LIFETIME is disabled (and the session is 'forever')? Can we get in trouble with the DmoVersioning (I don't mean the cache or memory usage, I mean breaking the application in an abnormal way).

#31 - 06/23/2023 06:29 AM - Alexandru Lungu

Constantin Asofiei wrote:

> if it executes on the SessionCloseThread. We need tests and make sure the Conversation thread for that session gets notified and the exception (and messages) gets processed. Which can be either a DatabaseConnectionException or a StopConditionException, plus the ErrorManager.display calls.

The error management here is quite odd. What if the session closing fails at some point (>10 seconds of lifetime). The user operates the application and at some point the server decides to close the session due to time-out. Thus, the exception will be thrown "out-of-nowhere", uncorrelated to the user action basically.

An error here is not quite fatal. DmoVersioning did its job, so an exception here only means that the connection couldn't be closed. Ignoring it will lead to having un-closed connections to the database, right? What would be the alternative plan anyway?

- do some business logic for ON ERROR? I don't this this is the case, but if it is, it is not quite good, as the session closing is non-deterministic
- display errors to the user + stop the client
  - either the client won't exist at that point, so there is nothing to do
  - the client exists, but most probably it already dealt with some persistence exceptions in the mean-time, or it will face them very soon
- even with all this effort of reporting the problem, the connection will still hang on the database (worst case).

What I am trying to say is that we can't properly recover in case of an exception, so maybe we should just do some logging and let it be. I am looking forward to SessionCloseThread as a FWD internal mechanism, unrelated to 4GL and client layers, so it should just fail in the background.

> > The intent here is to use the same session on two scenarios that happen very fast one after another.
>
> Think about in reverse: what happens if the LIFETIME is disabled (and the session is 'forever')? Can we get in trouble with the DmoVersioning (I don't mean the cache or memory usage, I mean breaking the application in an abnormal way).

There is something that is not right: what happens when the client actually exits? I think this should be handled so that the session is forcefully closed when the client disconnects, right? So that 'forever' actually means client life-time, not server life-time. Right now, the DmoVersioning and Session will leak in a 'forever' configuration.

Is the 'forever' configuration something that we actually consider (eventually for production environments)? If yes, we will go ahead implementing it.

#32 - 06/23/2023 07:42 AM - Constantin Asofiei

The agents in an appserver live for the lifetime of the FWD Server - so this Session can exist forever, if this LIFETIME is zero/very large.

What I'm trying to say with this extreme scenario is to think what happens with the DmoVersioning when there are multiple sessions in this 'pending to be reclaimed' state (as they 'live forever'). Don't worry about mem leaks for now: we need to know if this can have an actual impact in the application execution or not (i.e. if we can end up with DMOs in incorrect state, records not saved, etc).

#33 - 06/26/2023 07:25 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Constantin Asofiei wrote:
>
> > Alexandru, about the concurrency issues:
>
> Danut, please do the fixing.

I made a few changes here:

- I made the lifespan configurable and is initialized in SessionCloseThread.initialize(). Of course, the initialized variable was moved before starting the thread.
- In #7334-31 you mentioned that we could use logging since it may not be possible to recover if an exception is thrown, so I logged it.
- I modified the history entry numbers, only leaving the first one. Sorry for the trouble.

I also have a question, if we consider that a lifespan with the value zero will make the session to exist forever, is the thread even necessary? No session will expire, unless the factory is deregistered and a reclaimable session exists. I understand that we could use a large value for the lifespan, in this case using a thread makes much more sense, but running the following:

```
while (true)
{
    Thread.sleep(lifespan);
    removeIdleSessions();
}
```

will just cause unnecessary locking on the SessionFactory queue in removeIdleSessions.

> what happens when the client actually exits? I think this should be handled so that the session is forcefully closed when the client disconnects, right? So that 'forever' actually means client life-time, not server life-time.

The SessionFactory instance is deregistered in Persistence.Context.cleanup() which I assume is called when the client disconnects. Any reclaimable session available is closed is this situation.

I tested these changes and everything works well, should I commit them?

#34 - 06/30/2023 04:51 AM - Alexandru Lungu

- Status changed from Review to WIP

- % Done changed from 100 to 80

Danut, please commit them for further review.

> I also have a question, if we consider that a lifespan with the value zero will make the session to exist forever, is the thread even necessary? No session will expire, unless the factory is deregistered and a reclaimable session exists. I understand that we could use a large value for the lifespan, in this case using a thread makes much more sense, but running the following:

Let me reiterate the protocol here. **Constantin, I would like a feedback on this.**

- not setting a lifespan means eager session closing (like before).
- setting lifespan on 0 means infinite lifespan (client lifetime). This means we don't need the thread at all.
- setting lifespan on a positive value, means that we need a limited lifespan. This is what we intend to have it done right as it seems the best production use-case.

  What I'm trying to say with this extreme scenario is to think what happens with the DmoVersioning when there are multiple sessions in this 'pending to be reclaimed' state (as they 'live forever'). Don't worry about mem leaks for now: we need to know if this can have an actual impact in the application execution or not (i.e. if we can end up with DMOs in incorrect state, records not saved, etc).

From my understanding, DmoVersioning is an atomic mechanism of increasing the DMO versions. If any session is updating the state of the DMO, its version is increased. Other session will have to refetch the DMO as it may be stale according to DMOVersioning. Therefore, this is just a mechanism to avoid using stale data from persistent database. I don't really see why the session life-time may influence this. Of course there will be inconsistent data in long-living sessions, but as long as the DmoVersioning is doing a good job, the session should identify stale data even after hours/days of running.

I don't know exactly how shall we test if DmoVersioning is practically under-performing with these changes. Conceptually, it shouldn't.

Danut, the best we can do here IMO is to do a multi-session test with clients that read/write to a database. I know you've already tested #7334-20 for performance. Please consider refactoring it with some asserts:

- check that the user retrieves the correct data each time. This may be a bit hard as the multi-user environment makes the expectations inconsistent. Therefore, try to identify some deterministic cases here.
  - Keep a separate "last-update" table with exclusive lock.
  - If a session reads, it can check last-update table to identify who updated it last time. For example, if user 3 updated it with value 3, than we can expect to have value 3 in the DMO.
  - Follow this "artificial" versioning to check if FWD does the same job in the background right.
- consider using INSERT, UPDATE, READ and DELETE to have both read and write operations.
- mind testing with different kind of transactions (with one / more statements in it)
- mind testing with different kind of LOCKS
- I expect slow execution of such test, so I feel that we can have a 1/2 hour kind of script, as long as there are thousands of operations executed by each user in the end.
- Make the session last forever for best test environment.
  The point here is to ensure that we get consistent results on a dummy test here. Thank you!

#35 - 06/30/2023 04:55 AM - Constantin Asofiei

Alexandru Lungu wrote:

> Let me reiterate the protocol here. **Constantin, I would like a feedback on this.**
>
> - not setting a lifespan means eager session closing (like before).
> - setting lifespan on 0 means infinite lifespan (client lifetime). This means we don't need the thread at all.
> - setting lifespan on a positive value, means that we need a limited lifespan. This is what we intend to have it done right as it seems the best production use-case.

Yes, this is what I meant. But I didn't mean to actually implement the 'infinite lifespan', if it would not be used in production cases. My goal was to think of possible ways to break the session reclaim support - and, my theory was that if a break point can be found in a 'infinite lifespan' session (never reclaimed), than that bug will exist also in reclaimed sessions.

So, after analyzing DMO versioning the conclusion is that it works OK with the 'infinite lifespan', then we should be good.

#36 - 07/03/2023 03:31 AM - Alexandru Lungu

Danut, please focus on this task by testing DMO versioning. I am keen on merging 7334a asap.

In addition to #7334-34, make some kind of "infinite testcase". Let it run for 1/2 hours and then use a break-point to analyze the state of Session and DmoVersioning. Keep VisualVM open with the monitor to keep track of any heap spikes of GC overhead. Make your best to create a test with millions of small records that are used by the application (not all in the same time OFC).

#37 - 07/03/2023 09:48 AM - Alexandru Lungu

I rebased 7334a to trunk. 7334a is now at rev. 14640.

#38 - 07/04/2023 03:17 AM - Dănuț Filimon

Alexandru Lungu:

> Danut, please commit them for further review.

Committed 7334a/rev.14641. This will make the lifespan value configurable and the cases that were mentioned in #7334-34 are handled in the following way:

- a negative value: Logs a warning and sets the lifespan to **1s**;
- zero: The SessionCloseThread won't run at all (will log it as info), and the factories queue will be initialized in SessionCloseThread.initialize() instead of the constructor;
- positive value: The usual behavior that was targeted in the development until now.

> Danut, the best we can do here IMO is to do a multi-session test with clients that read/write to a database. I know you've already tested #7334-20 for performance. Please consider refactoring it with some asserts: ...

I am currently working on creating this test, which is not far from being complete.

#39 - 07/05/2023 07:28 AM - Dănuț Filimon

I managed to finish and test Session and DmoVersioning, I especially looked into Session.cache, Session.cacheExtension and DmoVersioning.version. The test works by running insert/read/update/delete or a mix of operations on 8 persistent tables, 1 client uses a routing table to run a list of operations on a persistent table in an infinite loop.

I wanted to use the main procedure to run 1 other persistent procedure that kept data about the executed operations but didn't manage to make use of the reclaimable sessions since the logic couldn't be accessed. Since I complicated it a little too much, I ended up discarding the validation and resume to a smaller test.

After 2 hours of using VisualVM and monitoring the activity of the script, I don't see any overhead in the GC. The heap space used follows a pattern of rising for ~5 minutes and then being reduced when the GC takes action. The cache sizes for Session.cache and DmoVersioning.versions are **10k** and **80k** respectively when running 8 clients.

As for the validation, I plan to use global shared variables and maintain the same logic as much as possible.

#40 - 07/06/2023 06:31 AM - Alexandru Lungu

- Status changed from WIP to Review

- % Done changed from 80 to 100

I will redo the testing and performance with this latest version (7334a / rev.14641). I intend to rebase and merge 7334a today EOD / first thing tomorrow.

Constantin, if there are other topics to be discussed here, please let me know. I will always check for obvious memory leaks in my profiling tests with 7334a (infinite session lifetime).
Danut, just to confirm:

- If I don't set any option, there will be no reclaim.
- If I use 0, the lifetime will be infinite.
- If I use a positive value, this will mean the ms (milliseconds) of maximum time in which a session can be reclaimed.

#41 - 07/06/2023 07:01 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Danut, just to confirm:
>
> - If I don't set any option, there will be no reclaim.
> - If I use 0, the lifetime will be infinite.
> - If I use a positive value, this will mean the ms (milliseconds) of maximum time in which a session can be reclaimed.

- If no option is set, the lifespan will be set to a default value (**1s**).
- If 0 is used as an option, the session will go through the reclaim process indefinitely.
- If a negative value is provided, the lifespan will be set to the default value.
- If a positive value is provided, it will represent the maximum time in which the session can be reclaimed.

In any case, the session will go through the reclaim process.

#42 - 07/06/2023 07:24 AM - Constantin Asofiei

Alexandru Lungu wrote:

> **Constantin, if there are other topics to be discussed here, please let me know. I will always check for obvious memory leaks in my profiling tests with 7334a (infinite session lifetime).**

I don't have anything else on my mind. Regarding the memory leaks: some of them may be slowly accumulating, and they will not be reported by Eclipse MAT. So please manually check the heap dump, look into the various caches and see if they are abnormal.

#43 - 07/07/2023 04:09 AM - Alexandru Lungu

Rebased 7334a with trunk rev. 14642. 7334a is now at rev. 14647.

Danut, please update 7334a and try to do a full testing round with Hotel GUI / personal test-cases. Please double check the interaction of your recent bootstrap changes and SessionFactory.SessionCloseThread.initialize();. Ensure that this thread is properly initialized before any session is created / registered / marked as reclaimable.

#44 - 07/07/2023 06:41 AM - Alexandru Lungu

The time improvement is -2.5%.

I've done two heap dumps and compared them. I have the same number of instances with very similar sizes when comparing, so there is no obvious leak (at least for DmoVersioning, Session, SessionCloseThread and SessionFactory). I had the server running for ~2h with 2 batches of 100 iterations.

Quite interestingly, I have a different number of UnclosablePreparedStatement between dumps (855 and 849, so less in the second dump). In fact, in persistence.orm I have 219.670 instances before and 219.664 after (the difference is related just to UnclosablePreparedStatement).

As it is now, I am planning to merge 7334a into trunk as there are no obvious problems. Danut, please address #7334-34 asap, so we can merge this in the next hour.

#45 - 07/07/2023 07:31 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Rebased 7334a with trunk rev. 14642. 7334a is now at rev. 14647.
>
> Danut, please update 7334a and try to do a full testing round with Hotel GUI / personal test-cases. Please double check the interaction of your recent bootstrap changes and SessionFactory.SessionCloseThread.initialize();. Ensure that this thread is properly initialized before any session is created / registered / marked as reclaimable.

I tested Hotel GUI and my own test-cases. SessionCloseThread is initialized correctly and the process involving creating/registering/marking/reclaiming sessions is proceeding as expected.

#46 - 07/07/2023 08:31 AM - Alexandru Lungu

Merged 7334a to trunk as rev. 14645. Archived 7334a.

#47 - 07/10/2023 03:52 AM - Dănuț Filimon

I updated Database Configuration with the configuration and explanation for session lifespan.

#48 - 07/10/2023 05:21 AM - Alexandru Lungu

- moved to #7425 -

#49 - 07/13/2023 08:59 AM - Alexandru Lungu

- Status changed from Review to Test

This reached trunk and can be closed.

#50 - 07/31/2023 11:32 AM - Eric Faulhaber

- File 7490_1.patch added

(In a different task) Alexandru Lungu wrote:

> > > Otherwise, there are 20 org.postgresql.jdbc.PgConnection, some are hooked on the cache for org.postgresql.jdbc.PgPreparedStatement.
> >
> > This is interesting. We must not have prepared statements cached which live beyond the time a connection would normally be checked back into the pool. Alexandru, we need your input on this cache implementation.
>
> As it is about PG, c3p0 is the one doing the caching, so I don't have any input on the "cache implementation".

OK, makes sense. I misremembered that there was an additional layer of prepared statement caching for persistence databases as well. IIRC, the c3p0 prepared statement cache is tied to the connection, so the cached statements should go away when the connection is closed (i.e., returned to the pool). That being said, I have not looked at the internal implementation of c3p0.

> AFAIK, as we don't actually close the connection, but just mark it as reclaimable, the prepared statements (from c3p0) are still cached. I don't feel like this is a problem, we are still hanging on some cached PreparedStatement instances as long as we have a reclaimable session. This was relevant for the performance improvement. Of course, after the session expires, the session is closed, so the PreparedStatement are freed - or at least this should happen behind c3p0.

With the default settings, this is just a matter of timing, correct? Meaning, with Constantin's patch (attached for reference), we should not be in a situation where connections are held checked-out indefinitely, such that the pool is exhausted. Sessions not reclaimed in a short period of time are expired, releasing their connections back to the pool.

I originally intended Session and in particular, holding a connection open, to be as short lived as possible. Checking in and checking out a connection from the pool is supposed to be a very fast operation, so the idea was that a Session instance was meant to be used only as long as a database connection was needed for some series of operations. However, the cache attached to the Session instance is not necessarily a fast resource, and cache operations can cause a performance bottleneck when new caches are constantly created, populated, emptied, etc. So, I understand the rationale behind this optimization.

However, I don't know that it's a good thing to hold open the connection (i.e., keep it checked out from the pool) while a session is in a "reclaimable" state. If this is only for a very short period of time (especially considering the relatively long life span of some sessions, due to the long lifespan of some application transactions), this may be ok. Holding a database connection longer than necessary for any period adds to the resource load of the database, but let's consider for a moment that this is acceptable for short reclaim settings. But as the reclaim period increases, we will have more connections in an idle, checked-out state for longer. This is not great, as it adds resource load to the database and to the application. Nevertheless, one option is simply to increase the connection count and pool size.

Even though Session originally was intended to be used for a single connection check-out, should we be checking in the connection when we enter the reclaim state, and checking-out a connection (not necessarily the same one) if and when the session is reclaimed? By doing this, we would not be able to rely on any cached, prepared statements to be available across session reclaims. Is this ok from a functional perspective? Also, is it ok from a performance perspective, or is skipping the check-in/check-out steps material to the performance improvement from this feature?

#51 - 08/01/2023 04:55 AM - Alexandru Lungu

- % Done changed from 100 to 80

- Status changed from Test to WIP

> Even though Session originally was intended to be used for a single connection check-out, should we be checking in the connection when we enter the reclaim state, and checking-out a connection (not necessarily the same one) if and when the session is reclaimed?

I don't think there is any problem if we actually "close" (send back in c3p0 pool) the underlying database connection and pick up another one on reclaim. Our persistence layer should be resilient with such "low-level" connection change. However, I didn't test such hypothesis.

From my POV, this reclaimable feature should be used only to "glue" two or more transactions to make use of the existing cache. Thus, it should be set to the lowest value possible, such that it still has time to "glue" use-cases that happen one after another in a really low time threshold. I am mostly thinking of appserver actions: a request is sent, but multiple transaction, thus sessions, could have been used to full-fill that appserver request. IMHO, the whole appserver request should have happened within the same database session, so that is my main motivation of supporting reclaiming sessions.

Reaching to a conclusion: if we feel like the c3p0 connection pool is exhausted by the reclaiming process, it means that we've set a reclaiming time that is too large. It should have "glued" some heavy processes with **finesse**, not dump all operations in a single connection for several seconds/minutes (or worse).

> By doing this, we would not be able to rely on any cached, prepared statements to be available across session reclaims.

I think that the session cache can be reused without problems. Indeed, the prepared statements may be spoiled. **However**, I've seen quite non-deterministic behavior from c3p0:

- when setting a connection option and connection is returned to the pool, any of the pool consumers have a non-deterministic change to pick up the connection with that option set.

I am thinking that prepared statements are supported similarly, but I need further investigation. Maybe c3p0 can return the same connection **with** the same cached prepared statements. Otherwise, it can deliver the prepared statement cache to
another consumer.

Eric, by **rely** I am rather thinking on performance rather than correctness. The prepared statements are recompiled if needed, so I don't ever think of "prepared statements cache" as a corrupt cache, but a cache that may have irrelevant statements cached.

> Is this ok from a functional perspective? Also, is it ok from a performance perspective, or is skipping the check-in/check-out steps material to the performance improvement from this feature?

The main drive for this task was #7167. The issue was that hanging records from opened buffers were not associated with the session. Lot of time was spent in iterating **all** record buffers from **all** scopes, filtering them by schema (and multiplex) and attaching (Session.associate) them back to the session (BufferManager.activeBuffers). As this was about the session cache association, I didn't actually thought of prepared statements caching in the first place, so I wouldn't mark this as "material to the performance improvement".

I moved this to 80% to allow us to test the connection check-in/check-out for reclaimable scenarios. Also, I agree that the patch sent by Constantin is a regression fix - good catch.

#52 - 08/07/2023 05:23 AM - Dănuț Filimon

Session reclaiming couldn't be disabled since most values either default to a set value or only disable the managing thread (see #7334-41). In #7490-59, Alexandru requested a way to disable the reclaiming process which was done in **7490a/rev.14683** by using the negative value (when one was used, it would change it to a default value). Database Configuration has been updated with the new configuration settings.

#53 - 01/15/2024 04:03 AM - Dănuț Filimon

- Related to Bug #7167: Associating records from opened buffers to new sessions is slow added

#54 - 02/12/2024 02:26 AM - Alexandru Lungu

This reached trunk a while ago. Is there anything left on this task? If not, please move it to test.

#55 - 02/12/2024 02:29 AM - Dănuț Filimon

- % Done changed from 80 to 100

- Status changed from WIP to Internal Test

Alexandru Lungu wrote:

> This reached trunk a while ago. Is there anything left on this task? If not, please move it to test.

There is nothing left to do here, last change involving session reclaiming was done for #7490 which already reached trunk.

#56 - 02/12/2024 02:29 AM - Dănuț Filimon

- Status changed from Internal Test to Test

Files

7490_1.patch	913 Bytes	07/31/2023	Eric Faulhaber