Database - Support #4701

try to improve H2 transaction commit performance

06/29/2020 05:10 PM - Eric Faulhaber

Status:	Closed	Start date:	
Priority:	High	Due date:	
Assignee:	Alexandru Lungu	% Done:	100%
Category:		Estimated time:	0.00 hour
Target version:			
billable:	No	case_num:	
vendor_id:	GCD	version:	

Description

Related issues:

Related to Database - Support #4702: write temp-table performance test cases	**Closed**
Related to Database - Bug #4057: h2 performance degradation	**New**
Related to Database - Bug #4760: new keywords added in H2 1.4.200	**Closed**
Related to Database - Support #6679: H2 general performance tuning	**Internal Test**

History

#1 - 06/29/2020 05:12 PM - Eric Faulhaber

- Related to Support #4702: write temp-table performance test cases added

#2 - 06/29/2020 05:13 PM - Eric Faulhaber

- Subject changed from test performance of latest H2 release to try to improve H2 transaction commit performance

#3 - 06/29/2020 07:07 PM - Eric Faulhaber

H2's transaction commit code repeatedly shows up as a performance bottleneck when profiling FWD application code. We are doing what we can to invoke it less from FWD, but we also want to improve the performance of the logic within H2 to the degree possible. This will involve setting up the H2 build environment (see Building H2). Constantin will be able to give advice if something is unclear there.

Please coordinate with Stanislav on building test cases (see #4702) which will stress this area of H2 and see if something can be done to improve the internal performance of H2 in this area.

Start with version 1.4.197 and apply our existing patches before you build. Constantin has a relatively new patch in addition to the older ones. He can get you the correct set.

In #4057, we also are going to be looking at whether the latest build of H2 can help improve performance. If so, then we will want to apply any further performance fixes to that version and move to it.

#4 - 06/29/2020 07:07 PM - Eric Faulhaber

- Related to Bug #4057: h2 performance degradation added

#5 - 06/29/2020 08:24 PM - Constantin Asofiei

Eric Faulhaber wrote:

Start with version 1.4.197 and apply our existing patches before you build. Constantin has a relatively new patch in addition to the older ones. He can get you the correct set.

See #4011-415 for the patch.

#6 - 07/02/2020 11:26 AM - Adrian Lungu

- File simple-stress.png added

I have done some testing using the Apache JMeter in order to see some differences between H2 versions: 1.4.196, 1.4.197, 1.4.197 patched, 1.4.198 and 1.4.199 (I can do the testing on 1.4.200 as well if needed, but I need to change the sqls a bit - it seems that there is a syntax change in 1.4.200). I hoped for some results similar to #4057; however I got a rather different gradient. Anyways, I used only some simple generated sqls containing basic statements. The in-memory database had only one small table with 5 columns (1 integer and 4 varchar 200).

The image below shows the tests run over each version. The running system had 200 users/threads with 5/10 loops each. There was no auto-commit and the transaction isolation is the default one. The full tests have 200-2000 randomly generated statements, while the other have around 200 statements with 60% statements only of one type (insert, select etc.).

One should consider a 5-10% error threshold in time measurement. Also, note that the first run is not taken in consideration when computing the average. I consider that three cached runs are more relevant (the first one runs over a non-cached memory and seems to have a high variance).

The JMeter tests are saved, so I can test any future patched version in a reasonable time.

Test	Version	Threads	Loop	Columns	AutoCommit	Isolation	Step1	Step2	Step3	Step4		First Run	Time1	Time2	Time3	TimeAvg
e.jmx	v196	5	5	4 / 200 char	FALSE	Default	full test – 2000					93	82	81	81	81.33333333
e.jmx	v197	5	5	4 / 200 char	FALSE	Default	full test – 2000					89	78	80	81	79.66666667
e.jmx	v197 patched	5	5	4 / 200 char	FALSE	Default	full test – 2000					86	81	80	79	80
e.jmx	v198	5	5	4 / 200 char	FALSE	Default	full test – 2000					67	63	64	63	63.33333333
e.jmx	v199	5	5	4 / 200 char	FALSE	Default	full test – 2000					68	64	69	62	65
f.jmx	v196	200	5	4 / 200 char	FALSE	Default	full test – 200					28	22	23	23	22.66666667
f.jmx	v197	200	5	4 / 200 char	FALSE	Default	full test – 200					31	23	24	23	23.33333333
f.jmx	v197 patched	200	5	4 / 200 char	FALSE	Default	full test – 200					28	23	23	23	23
f.jmx	v198	200	5	4 / 200 char	FALSE	Default	full test – 200					28	19	17	17	17.66666667
f.jmx	v199	200	5	4 / 200 char	FALSE	Default	full test – 200					24	18	18	17	17.66666667
g.jmx	v196	200	5	4 / 200 char	FALSE	Default	Heavy insert test – 200					43	41	40	36	39
g.jmx	v197	200	5	4 / 200 char	FALSE	Default	Heavy insert test – 200					42	35	35	35	35
g.jmx	v197 patched	200	5	4 / 200 char	FALSE	Default	Heavy insert test – 200					42	40	36	36	37.33333333
g.jmx	v198	200	5	4 / 200 char	FALSE	Default	Heavy insert test – 200					40	32	28	28	29.33333333
g.jmx	v199	200	5	4 / 200 char	FALSE	Default	Heavy insert test – 200					42	29	26	26	27
h.jmx	v196	200	10	4 / 200 char	FALSE	Default	Heavy commit test – 200					28	24	22	23	23
h.jmx	v197	200	10	4 / 200 char	FALSE	Default	Heavy commit test – 200					27	24	24	23	23.66666667
h.jmx	v197 patched	200	10	4 / 200 char	FALSE	Default	Heavy commit test – 200					28	24	22	22	22.66666667
h.jmx	v198	200	10	4 / 200 char	FALSE	Default	Heavy commit test – 200					27	19	18	18	18.33333333
h.jmx	v199	200	10	4 / 200 char	FALSE	Default	Heavy commit test – 200					26	19	18	18	18.33333333
l.jmx	v196	200	5	4 / 200 char	FALSE	Default	Heavy delete test – 200					24	23	24	23	23.33333333
l.jmx	v197	200	5	4 / 200 char	FALSE	Default	Heavy delete test – 200					27	23	22	23	22.66666667
l.jmx	v197 patched	200	5	4 / 200 char	FALSE	Default	Heavy delete test – 200					32	24	23	22	23
l.jmx	v198	200	5	4 / 200 char	FALSE	Default	Heavy delete test – 200					27	18	18	18	18
l.jmx	v199	200	5	4 / 200 char	FALSE	Default	Heavy delete test – 200					26	20	18	18	18.66666667
j.jmx	v196	200	5	4 / 200 char	FALSE	Default	Heavy update test – 200					43	38	38	38	38
j.jmx	v197	200	5	4 / 200 char	FALSE	Default	Heavy update test – 200					43	39	38	37	38
j.jmx	v197 patched	200	5	4 / 200 char	FALSE	Default	Heavy update test – 200					43	39	38	38	38.33333333
j.jmx	v198	200	5	4 / 200 char	FALSE	Default	Heavy update test – 200					35	30	30	30	30
j.jmx	v199	200	5	4 / 200 char	FALSE	Default	Heavy update test – 200					45	30	31	31	30.66666667
k.jmx	v196	200	5	4 / 200 char	FALSE	Default	Heavy select test – 200					23	19	17	17	17.66666667
k.jmx	v197	200	5	4 / 200 char	FALSE	Default	Heavy select test – 200					23	18	18	17	17.66666667
k.jmx	v197 patched	200	5	4 / 200 char	FALSE	Default	Heavy select test – 200					29	18	17	16	17
k.jmx	v198	200	5	4 / 200 char	FALSE	Default	Heavy select test – 200					18	11	10	10	10.33333333
k.jmx	v199	200	5	4 / 200 char	FALSE	Default	Heavy select test – 200					19	11	10	11	10.66666667

Surprisingly, 1.4.196, 1.4.197 and 1.4.197 patched almost share the same times, while 1.4.198 and 1.4.199 have noticeable lower times in almost all tests. Eventually, this does not reflect the FWD use case necessarily as no index is used, no join queries are used and the statement selection is random. However, this should be seen as a basic performance test in terms of memory management and access (in regard with transactions).

At this point, I'm rather curious what test plan was used in order to generate the graphic in #4057. I couldn't detect in any of my test plans a 5 times decrease in performance from 1.4.196 to 1.4.197. #4057-3 seems to show only a testcase regarding a drop and create table statements (are this type of stress tests relevant to FWD?).

By now, I guess #4057-8 might prove a good idea if 1.4.200 respects the gradient (following 1.4.198, 1.4.199 having better performance). I will go ahead with testing some Hotel GUI workflows to get a better grip over #4701-3.

#7 - 07/02/2020 03:29 PM - Eric Faulhaber

Adrian, this performance data presents an interesting counterpoint to the information posted previously to #4057. Please put a link in that task to these findings. Since you have JMeter already set up, please test version 1.4.200 as well (with and without our patches applied -- we will not be abandoning these with newer versions, unless they have made fixes which make ours unnecessary), then add that data to your results. It will be interesting to see how these findings compare with Stanislav's findings in #4057 using the FWD test cases he is writing.

Have you found anything about the transaction commit implementation of H2 (any recent version, including the one we are using) which looks like it can be improved for better throughput?

#8 - 07/02/2020 04:05 PM - Adrian Lungu

I shared the findings in #4057. I will go ahead with testing the 1.4.200 version. Need just some syntax changes to the sql - hope the result will be consistent with the other tests.

Eric Faulhaber wrote:

> Have you found anything about the transaction commit implementation of H2 (any recent version, including the one we are using) which looks like it can be improved for better throughput?

I've only done some surface testing by now and didn't fully get into the H2 code yet. After adding 1.4.200 to the testing plan, I will switch to the H2 code investigation.

#9 - 07/02/2020 04:58 PM - Eric Faulhaber

Just found this in the H2 v1.4.200 change log, as something that was fixed:

- Issue # 1820: Performance problem on commit

I haven't found their issue tracking system to be able to look up the context. H2 has a Google Group with an Issues section, but everything there looks out of date.

In any case, this is promising for 1.4.200.

#10 - 07/03/2020 07:33 AM - Adrian Lungu

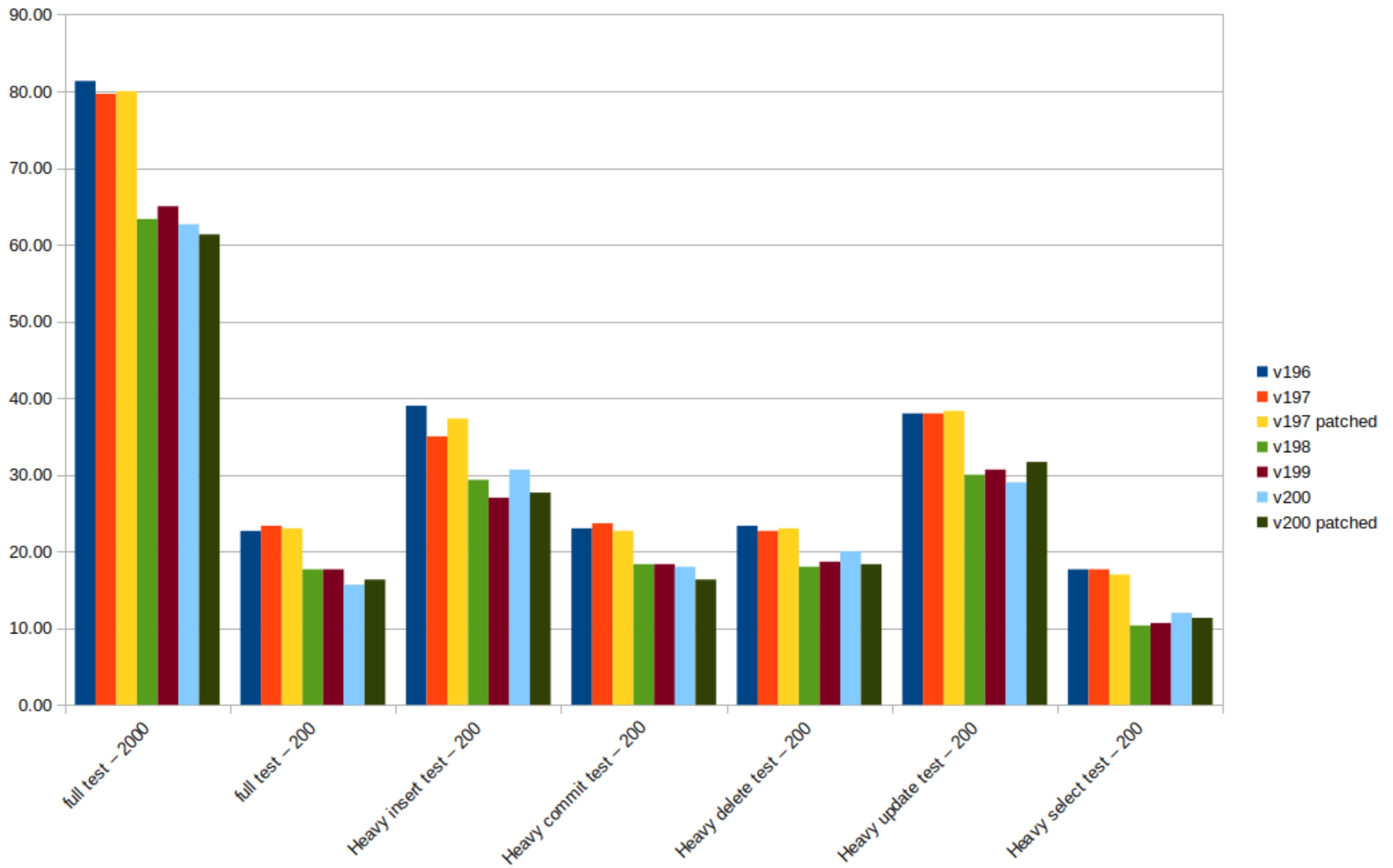
- File 200-stress.png added

- File chart-stress.png added

I have computed the testing results for 1.4.200 and 1.4.200 patched. The spreadsheet is attached and a representative chart is displayed below.

The 1.4.200 was not trivially patched with h2_synchronization_transactional_fix_20200529a.patch. The first 3 file patches succeeded with fuzz at most 3, while the 4th file (RegularTable) couldn't be patched. I could manually add the second hunk to RegularTable, while the other 3 hunks were patching non-existent methods in 1.4.200 (unlock, removeChildrenAndResources and doLock2). Actually, these methods were moved to

org.h2.pagestore.db.PageStoreTable.java. I manually applied the last 3 patches there. I can't say at this stage if these last patches are still required as they only apply synchronization in regard with this, which now refers to a PageStoreTable object, not to a RegularTable as in 1.4.197.



The results are encouraging. There is a visible increase in performance from the current 1.4.197 patched to the latest versions. 1.4.198, 1.4.199, 1.4.200 and 1.4.200 patched are in the same threshold, so I can't say that there is a considerable time reduction for 1.4.200 versions.

#11 - 07/03/2020 07:33 AM - Adrian Lungu

- % Done changed from 0 to 30

- Status changed from New to WIP

#12 - 07/03/2020 09:34 AM - Stanislav Lomany

Adrian, could you share 197 patched and 200 patched?

#13 - 07/03/2020 02:44 PM - Adrian Lungu

- File h2-200-patched.tar.gz added

- File h2-197-patched.tar.gz added

Stanislav,

I attached the h2 versions I used in testing. I can't attach the already built ones as they are too big.

Regarding my statement in #4701-10, the 4th file patch should target PageStoreTable instead of RegularTable as it still makes sense to have that modifications (synchronized blocks). If needed, I can provide the updated patch which removes the fuzziness and moves the modifications to PageStoreTable in order to have an 1.4.200 targeted patch.

#14 - 07/03/2020 02:56 PM - Eric Faulhaber

Adrian, please post the clean patches for the 1.4.200 source code base. Unless Stanislav's findings are vastly different than yours, we are likely to move to that version.

#15 - 07/03/2020 04:34 PM - Adrian Lungu

- File h2_1.4.200_synchronization_transactional_fix_20200703a.patch added

I attached the patch based on h2_synchronization_transactional_fix_20200703a.patch, but compatible with the 1.4.200 h2 database version

#16 - 07/06/2020 11:49 AM - Adrian Lungu

I investigated the H2 sources and eventually tried out some optimizations, but without any considerable performance boost. I can't yet reproduce an improvement close to the one depicted in #4057 (close to 80%).

The major points to be considered:

1. The FWD H2 connection is using mv_store=false (so it exploits the older Page Store engine). 1.4.200 has the MV store as default engine. In fact, the 1.4.200 version notes include Scalability and stability of MVStore engine are improved. I couldn't find major differences in the stress tests between these two. However, MV store may prove useful in certain scenarios and FWD can exploit this (should be tested).
2. multi_threaded and mvcc were removed in 1.4.200, so the FWD H2 connections should be updated in this direction.

Right now, I can't find a reliable direction for the upcoming investigation. Eric, are there some tools which can be used in profiling the FWD performance in regard to the transaction commit issue. On a general purpose, H2 seems to handle the basic testcases efficiently. I am thinking that maybe there is pattern in FWD which somehow exploits very specific weaknesses of H2 (indexes, joins, sub-queries etc.)?

#17 - 07/06/2020 12:03 PM - Constantin Asofiei

Adrian, in my profiling, I saw a significant time being spent in orh.h2.Session.cleanTempTables. Please create a test with 1000s of temp-tables and a loop like this:

```
def var i as int.
def var j as int.
repeat transaction j = 1 to 10000:
   create tt1.
   tt1.f1 = j.
end.
```

Do idea here: AFAIK the inner block will commit the tx on each iteration, and orh.h2.Session.cleanTempTables will be called.

#18 - 07/06/2020 12:43 PM - Constantin Asofiei

Constantin Asofiei wrote:

> Adrian, in my profiling, I saw a significant time being spent in orh.h2.Session.cleanTempTables. Please create a test with 1000s of temp-tables and a loop like this:

BTW, for FWD to generate different physical temp-tables for 4GL tables, these need to differ in terms of their schema, and not field names.
For example:

```
def temp-table tt1 field f1 as int field f2 as int.
def temp-table tt2 field f3 as int field f4 as int.
```

will be backed by the same H2 database.

So, these need to differ by their list of field data types, and not their names.

#19 - 07/07/2020 11:19 AM - Adrian Lungu

- File parsed_log.zip added

I created a test with 1000 temp-tables and the suggested loop. The transaction is in fact committed after each iteration (also created different tables according to the #4701-18 hint). I could find a way to extract the sequence of SQLs executed on the H2 engine. I made use of the TRACE_LEVEL_FILE=4 command, INFO logging and a parsing script.

I attached here the parsed log which contains all the statements executed on H2 when using the testcase in #4701-17. I will continue by using the H2 profiler and my debugger to detect any performance issue.

#20 - 07/10/2020 09:49 AM - Adrian Lungu

- File h2_1.4.200_local_temp_table_fix_20200710.patch added

The org.h2.Session.cleanTempTables direction was a good starting point. H2 was sequentially checking all local temp tables at each commit (in order to detect if they are flagged as onCommitDrop or onCommitTruncate). In the #4701-17 testcase, the total run-time was of around 7 seconds. org.h2.Session.cleanTempTables was holding the database synchronized for a total of 2/3 seconds without finding any table marked with onCommitDrop or onCommitTruncate.

I changed org.h2.Session.cleanTempTables such that the flagged tables are stored separately and can be accessed directly (without any sequential searching). Based on H2 profiling, the time reduction is around 40%-50%. However, this is because the testcase is very specific (it exploits this vulnerability explicitly). This improvement refers to testcases where:

1. A lot of local temp-tables are used concurrently.
2. A lot of users are accessing the database. org.h2.Session.cleanTempTables is locking the database each time it needs to search for onCommitDrop and onCommitTruncate tables.

#21 - 07/10/2020 12:49 PM - Eric Faulhaber

Adrian, we will start using this version of H2 (i.e., 1.4.200, patched) as the default with 4011b. Please update the gradle build accordingly.

#22 - 07/10/2020 03:03 PM - Constantin Asofiei

Adrian Lungu wrote:

> 1. A lot of local temp-tables are used concurrently.

This is something in use by customer applications - lots of temp-tables, either static or dynamic, can be used by a single FWD client.

> 1. A lot of users are accessing the database. org.h2.Session.cleanTempTables is locking the database each time it needs to search for onCommitDrop and onCommitTruncate tables.

Can you please upload the patched src/main/org/h2/engine/Session.java file? I'd like to see how the locking is being done now.

#23 - 07/11/2020 09:09 AM - Adrian Lungu

- File Session.java added

I attached the src/main/org/h2/engine/Session.java file.

1. A lot of users are accessing the database. org.h2.Session.cleanTempTables is locking the database each time it needs to search for onCommitDrop and onCommitTruncate tables.

By this, I mean that the synchronization in Session.cleanTempTables is done way more rare (there is no new locking method). This is because, localTempTablesManager.canClean(closeSession) now prevents redundant calls to Session._cleanTempTables. This is an improvement in the concurrency matter, as the problematic iterative search was locking the database for too long without proper action.

#24 - 07/13/2020 02:26 AM - Eric Faulhaber

We have upgraded to H2 v1.4.200 (patched), starting with 4011b/11555.

#25 - 07/13/2020 02:28 AM - Eric Faulhaber

Eric Faulhaber wrote:

> Adrian, we will start using this version of H2 (i.e., 1.4.200, patched) as the default with 4011b. Please update the gradle build accordingly.

Sorry if my previous note was not clear. I made the gradle changes myself, so you do not need to do this anymore.

#26 - 07/13/2020 03:20 AM - Constantin Asofiei

Adrian Lungu wrote:

> By this, I mean that the synchronization in Session.cleanTempTables is done way more rare (there is no new locking method). This is because, localTempTablesManager.canClean(closeSession) now prevents redundant calls to Session._cleanTempTables. This is an improvement in the concurrency matter, as the problematic iterative search was locking the database for too long without proper action.

OK, so in our case cleanTempTables will never execute the _cleanTempTables because localTempTablesManager.canClean will return false.

#27 - 07/13/2020 03:45 AM - Eric Faulhaber

Adrian, I should have confirmed this at the beginning, but was your performance comparison work done with the MVStore engine or the PageStore engine? I assume your optimization patch was done with PageStore, since that is what FWD uses, but I just want to be sure.

#28 - 07/13/2020 04:32 AM - Constantin Asofiei

Constantin Asofiei wrote:

> Adrian Lungu wrote:
>
>> By this, I mean that the synchronization in Session.cleanTempTables is done way more rare (there is no new locking method). This is because, localTempTablesManager.canClean(closeSession) now prevents redundant calls to Session._cleanTempTables. This is an improvement in the concurrency matter, as the problematic iterative search was locking the database for too long without proper action.
>
> OK, so in our case cleanTempTables will never execute the _cleanTempTables because localTempTablesManager.canClean will return false.

Adrian, please double-check that on commit drop or on commit delete rows tables are never used in FWD (these are the kind of options at CREATE TABLE which would require to lock on the database in cleanTempTables). What FWD uses are ON DELETE CASCADE at the foreign-key for the child extent tables.

#29 - 07/13/2020 11:21 AM - Adrian Lungu

> Adrian, I should have confirmed this at the beginning, but was your performance comparison work done with the MVStore engine or the PageStore engine? I assume your optimization patch was done with PageStore, since that is what FWD uses, but I just want to be sure.

Eric, the tests were done using the same connection string used in FWD (removing mvcc and multi_threaded where needed). This means that the tests were done using PageStore.

> Adrian, please double-check that on commit drop or on commit delete rows tables are never used in FWD (these are the kind of options at CREATE TABLE which would require to lock on the database in cleanTempTables). What FWD uses are ON DELETE CASCADE at the foreign-key for the child extent tables.

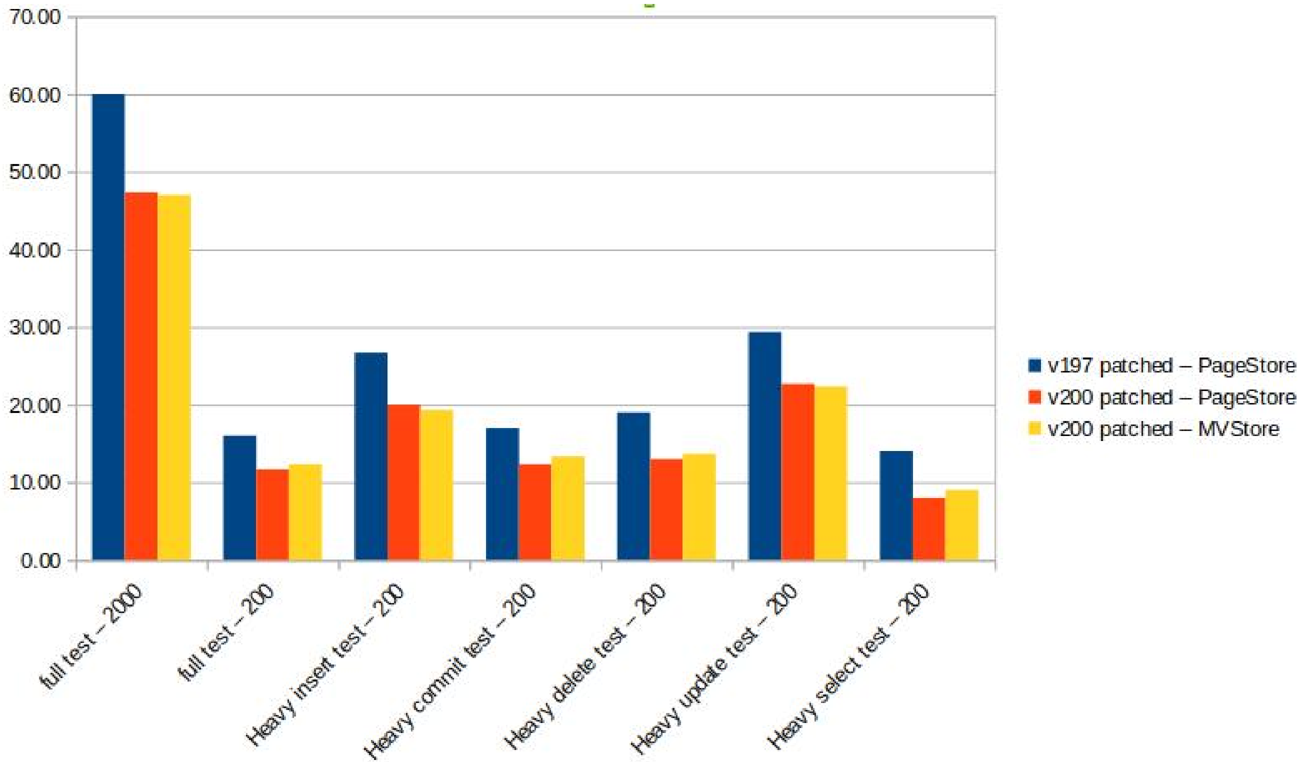
Constantin, I can't find any usage of H2 on commit drop or on commit delete rows in FWD. This means that, in fact, _cleanTempTables will never be executed in FWD's case. However, if future modifications are done in order to exploit these features, the 1.4.200 H2 patch enhances the performance of those anyway. localTempTablesManager.canClean may return true if there are on commit drop or on commit delete rows tables and those are processed efficiently. The only worry regarding the use of on commit drop or on commit delete rows is the database synchronization - unless mv_store is used.

This being said, the patch fits the need in FWD as no on commit drop or on commit delete rows tables are defined in H2.

#30 - 07/14/2020 06:50 AM - Adrian Lungu

- File chart-mv-vs-page.png added

Below there is a comparison between 1.4.197 patched using PageStore, 1.4.200 patched using PageStore and 1.4.200 using MVStore. Note that the same pure performance tests (randomly generated) as in #4701-10 were used. There is no visible difference between PageStore and MVStore in 1.4.200. I strongly believe that this is due to the fact that the tests are somehow general. There is a high change that PageStore is naturally better fitting FWD (as it was when using 1.4.197).



I will also make use of my tests which exploit a FWD real case.

#31 - 07/14/2020 08:01 AM - Adrian Lungu

- File profiling_mv_vs_page.png added

After Java profiling the #4701-17 testcase, I got the following results for 1.4.200 patched:

JAVA PROFILING	PAGESTORE 1.4.197	PAGESTORE 1.4.200	MVSTORE 1.4.200
PRIVATE – ONE USER	1.5	0.23	1.37
PUBLIC _temp – ONE USER	1.46	0.23	1.41
PRIVATE – 10 USERS	4.05	0.86	5.63
PUBLIC _temp – 10 USERS	18.23	1.35	JdbcSQLTimeoutException
SQL PROFILING	PAGESTORE 1.4.197	PAGESTORE 1.4.200	MVSTORE 1.4.200
PRIVATE – ONE USER	1.53	0.52	1.81
PUBLIC _temp – ONE USER	1.45	0.49	1.87
PRIVATE – 10 USERS	4.14	1.53	7.01
PUBLIC _temp – 10 USERS	14.14	2.3	JdbcSQLTimeoutException

The Session.cleanTempTables patch was targeting both MVSTORE and PAGESTORE. It seems that MVSTORE still has some flaws when dealing with multiple temp-tables (expecting the Session.cleanTempTables). Also not that the 1.4.197 version used for testing is still using the old Session.cleanTempTables.

The JDBC exception displays Timeout trying to lock table "SYS";. This is weird, as this is clearly caused by the engine difference (as long the PageStore engine does not show such exception). I also tried to set the LOCK_TIMEOUT to 30 seconds, but the exception still occurs. This is an issue to be taken in consideration if moving to MVSTORE.

#32 - 07/14/2020 09:42 AM - Constantin Asofiei

Eric/Adrian/Stanislav: H2 v 1.4.200 introduced new keywords, for example array. This statement works with 1.4.197 but not with 1.4.200:

```
CREATE LOCAL TEMPORARY TABLE tt1 (
   ARRAY INTEGER
) ;
```

Is there a place in the H2 source-code where all keywords are defined?

We need to update NameConverter.reservedSQL and P2JH2Dialect.getReservedKeywords().

#33 - 07/14/2020 01:43 PM - Eric Faulhaber

Constantin Asofiei wrote:

> Is there a place in the H2 source-code where all keywords are defined?

I am not familiar enough with the H2 source code base to answer, sorry.

#34 - 07/14/2020 01:45 PM - Eric Faulhaber

Adrian, have you found any other potential areas of improvement in H2, based on any of the testing/profiling you have done? Are you still looking?

#35 - 07/14/2020 02:42 PM - Constantin Asofiei

- Related to Bug #4760: new keywords added in H2 1.4.200 added

#36 - 07/25/2020 07:44 AM - Adrian Lungu

Moved the Timeout trying to lock table "SYS" topic from #4753 here.

1. The exception can appear with different stack traces: #4753-66 and #4753-81
2. The issue is due to a deadlock caused by the SYS table and the database common instance: #4753-111
3. The lock on the SYS table persists over multiple API calls due to the transactional keyword introduced with h2_synchronization_transactional_fix_20200529a.patch. In between the H2 API calls, another thread locks the database instance and causes deadlock: #4753-137

Just tested the 1.4.197 patched (with and without multi_threaded flag) and I receive the same exception. Right now, I need to find a way to integrate the changes from the patch (related to the transactional keyword) such that no deadlock appears.

#38 - 07/27/2020 09:14 AM - Adrian Lungu

- File h2_1.4.200_meta_lock_fix_20200727.patch added

I found out that the transactional keyword was exploiting a vulnerability of constraints inside create table statements. The constraints (through AlterTableAddContraint) are locking the "SYS" table. If these constraints are used inside a transactional create statement (like in FWD), the "SYS" table won't be unlocked at the end of the command. I have seen multiple times in the flow of a table creation (CreateTable.update and PageStoreTable.addIndex) the following structure:

```
boolean isSessionTemporary = data.temporary && !data.globalTemporary;
if (!isSessionTemporary) {
    db.lockMeta(session);
}
```

This means that the meta locking is guarded by the temporary and globalTemporary flags. The meta locking in AlterTableAddConstraint.tryUpdate is done anyways, although the only meta dependent method used is db.addSchemaObject(session, constraint); inside:

```
if (table.isTemporary() && !table.isGlobalTemporary()) {
    session.addLocalTempTableConstraint(constraint);
} else {
```

```
    db.addSchemaObject(session, constraint);
}
```

It means that the meta lock should also be guarded by the flags, as in the other cases. I've attached the h2_1.4.200_meta_lock_fix_20200727.patch patch which includes this modification: adding a conditional for meta locking in AlterTableAddContraint.tryUpdate(). With this patch, I can't recreate the Timeout trying to lock table "SYS" bug anymore.

#39 - 07/27/2020 09:21 AM - Adrian Lungu

I uploaded fwd-h2-1.4.200-20200727.jar to devsrv01:/tmp/, which includes the patch mentioned in #4701-38.

#40 - 08/01/2020 05:54 PM - Eric Faulhaber

Adrian, as I understand it, your JMeter test environment can simulate heavy multi-user use, correct?

I would like you to please try an experiment, both for a single user, and for multiple users. Namely, how does performance and resource use compare between using a single H2 database instance (as we do today) to support all user sessions for legacy temp-table support vs. using a separate H2 database instance for each user session?

My expectation would be that using one database instance per session would use more resource (primarily heap), but it might be faster, as we can avoid any contention which might occur due to H2's "synchronize on the database" approach. I would not expect any benefit in the single user case, since that is essentially how it works today for a single user. But perhaps in the multi-user case, it could be faster by avoiding any lock contention?

I think the change would just require that the database name in the JDBC connection URL is made unique per user session. The test should be for in-memory, embedded database mode.

#41 - 08/03/2020 08:00 AM - Adrian Lungu

Eric Faulhaber wrote:

> Adrian, as I understand it, your JMeter test environment can simulate heavy multi-user use, correct?

Yet, it does.

> I would like you to please try an experiment, both for a single user, and for multiple users. Namely, how does performance and resource use compare between using a single H2 database instance (as we do today) to support all user sessions for legacy temp-table support vs. using a separate H2 database instance for each user session?

This is a feature I was looking at recently. The first time I noticed such improvement was in #4701-31. PUBLIC _temp refers to a single database instance, while PRIVATE refers to unique database instances for each user (anonymous databases). Barely have I focused on this matter by now, but I also feel that this is an area of improvement. I will start the experiment right away.

> My expectation would be that using one database instance per session would use more resource (primarily heap), but it might be faster, as we can avoid any contention which might occur due to H2's "synchronize on the database" approach. I would not expect any benefit in the single user case, since that is essentially how it works today for a single user. But perhaps in the multi-user case, it could be faster by avoiding any lock contention?

The results in #4701-31 do show an improvement in terms of time when dealing with multiple users. Indeed, for only one user, the time performance doesn't seem to change. However, the test in #4701-31 is slim and does not provide a memory insight. I will create more expressive results.

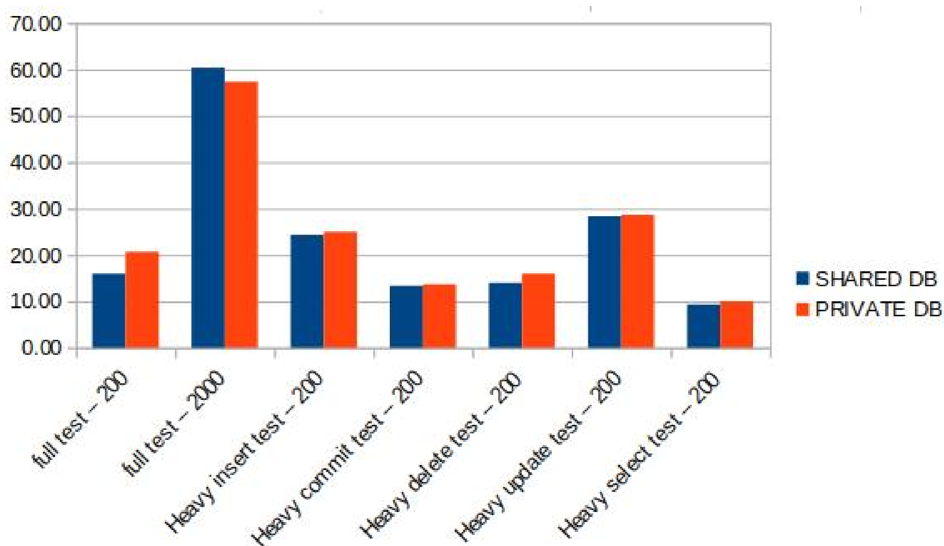
> I think the change would just require that the database name in the JDBC connection URL is made unique per user session. The test should be for in-memory, embedded database mode.

As mentioned earlier, private in-memory databases are suitable (jdbc:h2:mem:;db_close_delay=-1;mv_store=false;query_cache_size=1024;).

#42 - 08/05/2020 10:02 AM - Adrian Lungu

- File shared_vs_private.png added

The preliminary results reported by Apache JMeter are not very promising. However, note that the used tests are pure performance tests, which do not specifically reflect any FWD scenario (they are randomly generated). The diagram below shows a comparison between the use of a shared database (reflects the current FWD approach) and the use of private databases. All tests were run against 200 users (except for full_test_2000 which was run on only 5 users).



This direction may not be expressive enough as it doesn't contain table specific operations (create, drop, indexing) or complex statements. Right now, I am extracting some SQL batches based on Stanislav's "Running FWD-based performance tests for H2" in https://proj.goldencode.com/projects/p2j/wiki/Performance_Testing_the_H2_Database. This can greatly improve the SQL testcases pool and eventually provide more trusty results. By now, the #4701-17 testcase was already profiled in #4701-31; hopefully, other FWD based testcases will also be more closer to the time differences there. The FWD based testcases can also be memory profiled due to the "Custom Java Profiler" from the wiki provided.

#44 - 08/06/2020 10:45 AM - Greg Shah

Your results in #4843 (per-user "private" _temp H2 instances vs the current "global" instance approach) are very encouraging. An improvement of 30% to 50% in multi-user systems is significant. It highlights the cost of contention in such cases.

What is the development effort needed to make this work? Is the meta database a problem?

#45 - 08/06/2020 11:46 AM - Adrian Lungu

Greg Shah wrote:

> What is the development effort needed to make this work? Is the meta database a problem?

At this stage, I think that a shared "meta database" should be made separately from the per-user private database. This "meta database" will hold all the meta tables defined in p2j.cfg.xml. This will imply two database connections for each user and all requests will require filtering/redirecting to the right database. Another matter is the sequence used for primary key generation (recid). This should be also made private for each user.

#46 - 08/07/2020 09:49 AM - Greg Shah

Please build the necessary changes. It would be best to implement this such that we can set a flag and choose "per-user vs global temp database", but if that is not reasonable then do it the simple way.

#47 - 08/09/2020 09:22 AM - Adrian Lungu

- File read_scenario.png added

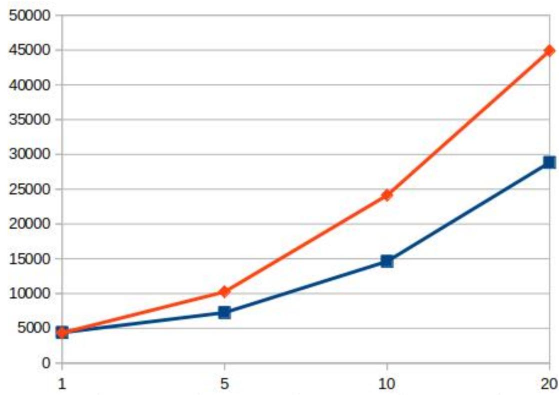
- File copy_scenario.png added

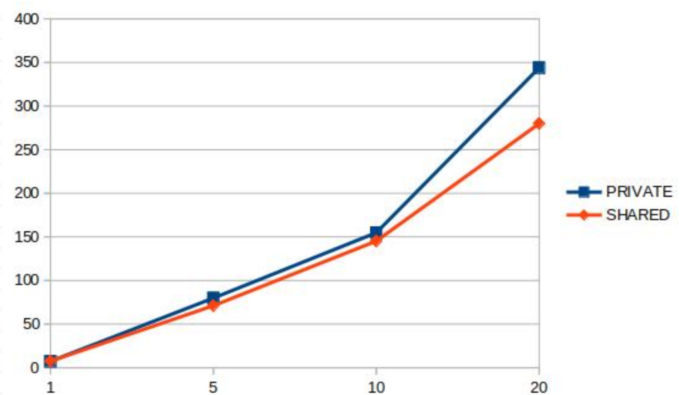
- File create_scenario.png added

- File delete_scenario.png added

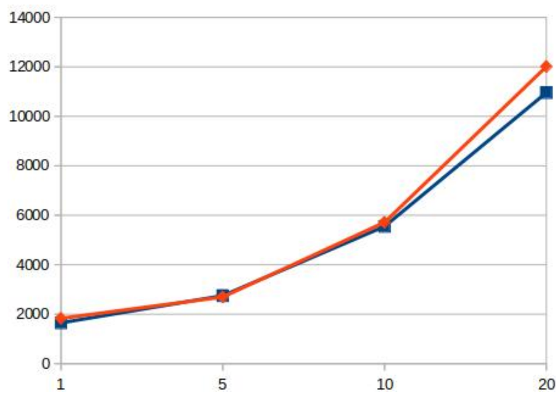
Recently I have finished building new FWD testcases based on [#4701-42](#4701-42) (extracting SQL batches from h2_performance testcases). Below there are diagrams similar to the one in #4843 describing the time and memory performance of per-user databases and global database. The x-axis represents the number of users, while the y-axis represents miliseconds (for the time diagram - on the left side) or MB (for memory diagram - on the right side).

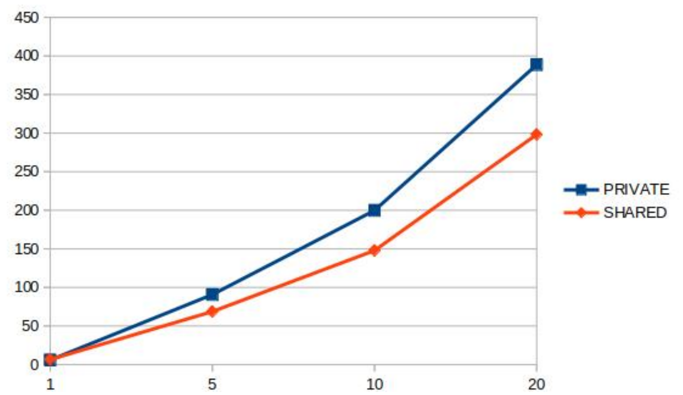
- Create scenario



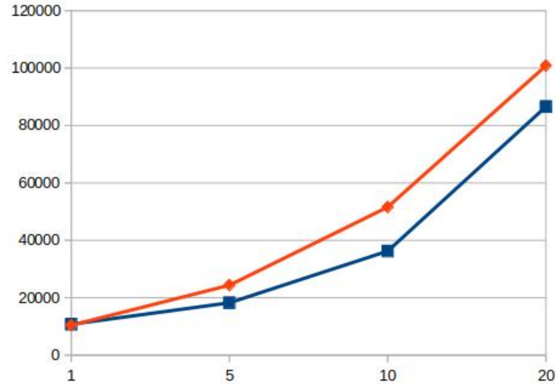


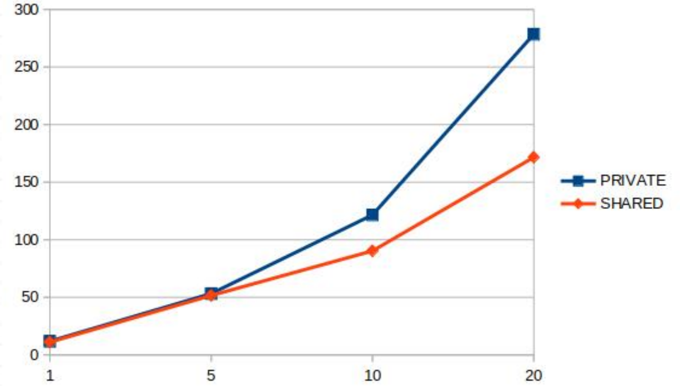
- Read scenario



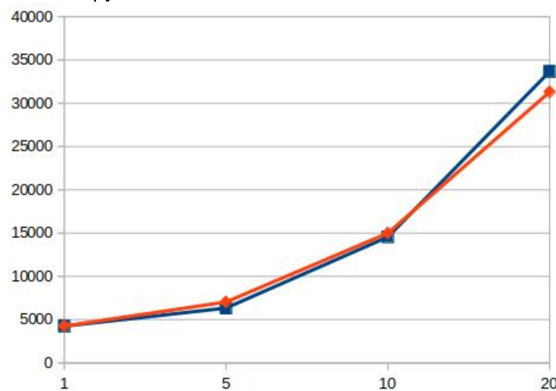


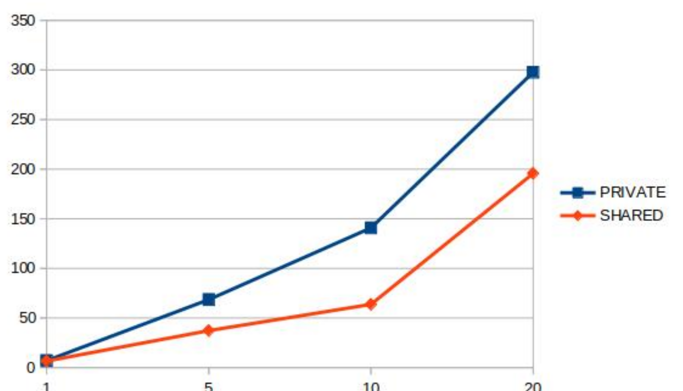
- Delete scenario





- Buffer copy scenario





The results confirm the conclusions drawn in #4843-1, but with extra insights. The performance difference is:

1. 29%-35% time decrease and 10%-18% memory increase for the create scenario
2. 0%-9% time decrease and 23%-25% memory increase for the read scenario (note that on some cases there is a time increase of at most 2%)
3. 14%-30% time decrease and 3%-38% memory increase for the delete scenario
4. 0%-10% time decrease and 34%-55% memory increase for the copy scenario (note that on some cases there is a time increase of at most 8%)

These testcases were added in https://proj.goldencode.com/projects/p2j/wiki/Performance_Testing_the_H2_Database. As a major highlight, all scenarios show an increase in memory when using private databases, but not in all cases the time will be improved.

Greg Shah wrote:

> Please build the necessary changes. It would be best to implement this such that we can set a flag and choose "per-user vs global temp database", but if that is not reasonable then do it the simple way.

Started working on this. I think that FWD will benefit from a "per-user vs global temp database" flag as not all applications will show the same statistics in terms of time improvement against memory consumption.

#48 - 08/10/2020 09:30 AM - Adrian Lungu

By now, I delayed the initialization of the _temp database. Until now, FWD was initializing the _temp table at server startup - which was reasonable as long as there is only one temporary database for all users. I removed this early initialization and allowed ersistenceFactory.getInstance to eventually initialize the _temp database when firstly needed. I can't tell if there is any performance improvement just due to the _temp database delay (I guess the server startup shall be faster).

This was needed for the "per-user database" model, which should allow the database initialization at the user connection phase - and not at the server startup phase. However, I can't identify a way to differentiate users.

The PersistenceFactory is holding a single cache for all users, which means that there should be named databases for each user and eventually query the cache based on the current user. Therefore, the naming of the in-memory databases and the access of them should rely a user id. At this point, I can't find a way to identify the current user which requests a persistent instance and I need some help on this matter.

My plan is to create separate in-memory databases (_temp1, _temp2, _temp3) for each user. To make it transparent, the users will work with the so called _temp database on the high-level, while lower persistence layers will associate users to the correct databases. The lacking part right now is the identification of the "current user".

#49 - 08/10/2020 09:56 AM - Eric Faulhaber

Can you help me understand the proposed "high" and "low" level implementations?

I guess we are really differentiating by user session, not user, since the same user can have multiple sessions, and these will require distinct, unrelated sets of temp-tables.

There are several areas of the persistence code which already are arranged into context-local objects. Persistence$Context instances each naturally exist within a user context, so this may be a good place to hide this implementation detail. You could use SecurityManager.getInstance().getUserId() to get a unique ID per user session for the H2 URL. However, this is an expensive call, so you would want to call it once and cache the value, where possible.

#50 - 08/10/2020 10:00 AM - Constantin Asofiei

Eric Faulhaber wrote:

> You could use SecurityManager.getInstance().getUserId() to get a unique ID per user session for the H2 URL. However, this is an expensive call, so you would want to call it once and cache the value, where possible.

This can't work - in most applications, we have the same FWD account for all UI sessions. Use getSessionId() instead.

#51 - 08/10/2020 10:48 AM - Eric Faulhaber

Constantin Asofiei wrote:

> Eric Faulhaber wrote:
>
>> You could use SecurityManager.getInstance().getUserId() to get a unique ID per user session for the H2 URL. However, this is an expensive call, so you would want to call it once and cache the value, where possible.
>
> This can't work - in most applications, we have the same FWD account for all UI sessions. Use getSessionId() instead.

Sorry, that is what I meant. I didn't read the code carefully enough.

#52 - 08/10/2020 11:11 AM - Adrian Lungu

SecurityManager.getInstance().getSessionId() is what I needed.

Eric Faulhaber wrote:

> Can you help me understand the proposed "high" and "low" level implementations?

I was thinking that there will be only one Persistence for the _temp database, and any higher persistence layers (RecordBufer) will work on a "virtual" _temp. Only when the database connections are made, the Persistence$Context.getSession() will deliver per-user sessions to the real databases: _temp1, _temp2 etc.

Of course, the second idea is to replace Database TEMP_TABLE_DB = new Database(TEMP_TABLE_SCHEMA, Database.Type.PRIMARY, true); with something like:

```
Database getTempTableDB()
{
   int sessionId = SecurityManager.getInstance().getSessionId();
   String name = DatabaseManager.TEMP_TABLE_SCHEMA + sessionId;
   return new Database(name, Database.Type.PRIMARY, true, true);
}
```

This can be done in Persistence$Context and eventually cache the database instance. Right now, I am working on the second idea, so ignore the things about "high" and "low" level implementations for now.

#53 - 08/10/2020 11:33 AM - Eric Faulhaber

This approach makes me a bit nervous.

Note that the Database object is used as a key in a number of maps across the persistence framework. This new implementation detail (private vs shared temp-table H2 database) must be completely transparent to other parts of the persistence framework. To the framework, it must look as if we still have a single, logical _temp database. Please make sure this configuration choice does not "leak" through the use of many vs one Database object instance, such that we have to adjust for it anywhere else in the persistence code.

#54 - 08/12/2020 08:46 AM - Adrian Lungu

- % Done changed from 30 to 50

Created branch 4701a and committed the changes regarding per-user temporary databases rev. 11624. Please review.

I will try to see if there is any change in performance with rev. 11624 . I suspect that there may be a performance issue due to context look-ups in the new TemporaryDatabaseManager at each PersistenceFactory.getInstance or Settings.getString() for Settings.URL. Hopefully they will be motivated by the numbers in #4843.

Finally, I am working now on a flagging system to indicate if the per-user databases should be enabled or not.

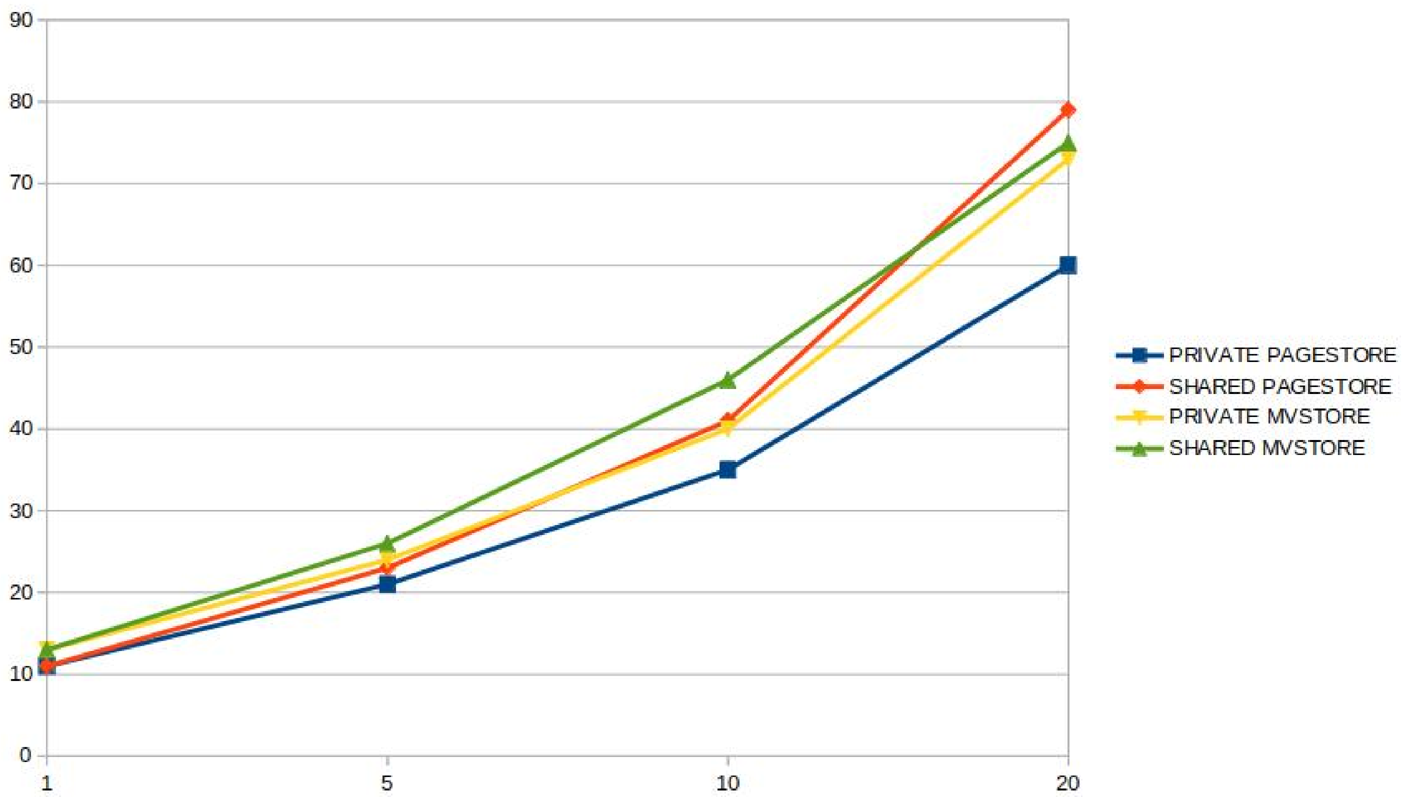
#55 - 08/13/2020 07:59 AM - Adrian Lungu

Committed 4701a rev.11625 - added switch between shared and private temporary database.

#56 - 08/15/2020 08:06 AM - Adrian Lungu

- File create_scenario_parallel.png added

I've done a final profiling of FWD in regard with h2_performance/perf-create.p. I've added to the testcase 4 configurations based on private vs shared database and pagestore vs mvstore engine. For the multi-user part, I generated multiple client instances in parallel and kept track of the highest user time. The x-axis represents the number of users and the y-axis is the number of seconds.



The PageStore engine still has better times than MVStore. Encouraging is the fact that the private temporary databases show better times than the shared database approach. In fact it shows the same time difference as in #4701-47. The ratio which in #4701-47 was of 29%-35%, on this testcase it is 9%-24%. These are lower, but they justify the fact that this improvement is only for H2, not the whole FWD database workflow.

#57 - 09/08/2020 03:35 PM - Greg Shah

Is there anything more to work on in this task or do we just need to get it rebased for merge to trunk (or merge into 3821c)?

#58 - 09/09/2020 04:42 AM - Adrian Lungu

- % Done changed from 50 to 100

- Status changed from WIP to Review

This task in finished. Waiting to rebase 4701b and eventually merge to trunk/3821c. However, I suggest to keep #4843 open for further testing (in case new H2 versions are released in the near future).

#59 - 09/13/2020 11:13 AM - Greg Shah

How risky is 4701a?

#60 - 09/14/2020 05:07 AM - Adrian Lungu

I find 4701a safe enough. The major change is the new private vs shared temporary database. Using a shared temporary database will mean no considerable change. Per-session temporary databases were manually tested in a customer application and no regressions were found.

#61 - 09/14/2020 05:51 AM - Greg Shah

Does this depend upon a new version of H2?

#62 - 09/14/2020 05:53 AM - Adrian Lungu

4701a is independent upon the H2 version.

#63 - 09/15/2020 07:24 PM - Eric Faulhaber

Code review 4701a, rev 11624-11627:

This review is for the private vs. shared temp-table database implementation. Someone else should review the handle-related changes which also are in this branch.

In general, it looks fine, but I think the assumption that a temp-table database instance could be represented by a RemotePersistence object may have complicated the design of this feature. RemotePersistence is never used for temp-tables. Since the temp-tables contain private data for a user's session, they will only be available to that user and thus will be in an embedded database (shared or otherwise) in the local JVM process. RemotePersistence was implemented to enable server-to-server connections for remote access to persistent databases.

Thus, this code in the implementations of TempDbManager.initializeMyDatabase:

```
Persistence instance = (tempDb.isLocal()
                        ? new Persistence(tempDb, dialect)
                        : new RemotePersistence(tempDb, dialect));
```

...should be simplified to:

```
Persistence instance = new Persistence(tempDb, dialect);
```

This simplifying assumption might be used to further simplify the design of the overall implementation (by subclassing Persistence perhaps?), but I'm not sure and I don't want us to spend the time redesigning things, since the current implementation is working, according to your testing.

I am a little bit concerned by the addition of another ContextLocal variable (since we've been trying to get rid of this semantic as much as possible, for performance), but hopefully this is not being hit too hard.

I do have some questions:

- Why the change to HQLPreprocessor.registerFunction?
- Why change DatabaseManager.getDialect to delegate to TemporaryDatabaseManager.getMyTempDbDialect?

#64 - 09/16/2020 03:36 AM - Adrian Lungu

Eric Faulhaber wrote:

> Code review 4701a, rev 11624-11627:
>
> This review is for the private vs. shared temp-table database implementation. Someone else should review the handle-related changes which also are in this branch.

The "handle-related" changes are for #4656 and fix the infinite loop on application close (which right right now is fixed by commenting out the root cause). This commit was reviewed there by Constantin.

> In general, it looks fine, but I think the assumption that a temp-table database instance could be represented by a RemotePersistence object may have complicated the design of this feature. RemotePersistence is never used for temp-tables. Since the temp-tables contain private data for a user's session, they will only be available to that user and thus will be in an embedded database (shared or otherwise) in the local JVM process. RemotePersistence was implemented to enable server-to-server connections for remote access to persistent databases.
>
> Thus, this code in the implementations of TempDbManager.initializeMyDatabase:
>
> [...]
>
> ...should be simplified to:
>
> [...]
>
> This simplifying assumption might be used to further simplify the design of the overall implementation (by subclassing Persistence perhaps?), but I'm not sure and I don't want us to spend the time redesigning things, since the current implementation is working, according to your testing.

The initialization "protocol" of a persistence was based on PersistenceFactory.getInstance(), which was making a difference between Persistence and RemotePersistence for all databases (including the temporary database). I presumed that by having this in place by now, it will be suggestive to keep it in this same shape. Finally, the DatabaseManager.TEMP_TABLE_DB is always local, so the result of the conditional is always new Persistence(tempDb, dialect). I will do the change to avoid redundancy.

> I am a little bit concerned by the addition of another ContextLocal variable (since we've been trying to get rid of this semantic as much as possible, for performance), but hopefully this is not being hit too hard.

Context lookups will be done when temporary databases are removed (when a session is disconnected) or when the URL of a temporary database settings is queried (when a database connection is made). As long as the private temporary databases are per-session, I couldn't find another way to identify the current sessionId without a context lookup. I can do a count test on this matter.

> I do have some questions:
>
> - Why the change to HQLPreprocessor.registerFunction?

The temporary database (DatabaseManager.TEMP_TABLE_DB) is initialized several times (once for each session), because this is the only logical temporary database for all sessions. An important step in the initialization is to register the database functions. FWD is checking if DatabaseManager.TEMP_TABLE_DB has a specific function already registered and fails after the query of HQLPreprocessor.overloadedFunctions (in a scenario with at least two sessions). It is safe to ignore the sanity check here, because the temporary database can be initialized multiple times - the difference is made only at the connection time, where the URLs differ.

> - Why change DatabaseManager.getDialect to delegate to TemporaryDatabaseManager.getMyTempDbDialect?

The temporary database is no longer initialized and registered at the server startup. In case a PrivateTempDbManager is used, each user should initialize/register its own temporary database, and for this step the dialect is required. Evidently, DatabaseManager.dialects doesn't have the temporary database mapped yet, so DatabaseManager.TEMP_TABLE_DB has a null dialect. Because this type of database has a lazy initialization, the dialect will be computed through TemporaryDatabaseManager. Note that, in order to create a persistence, the database and the dialect are the only information required. On this, I think is more logical to have:

```
public static Dialect getDialect(Database database)
{
```

```
    Dialect dialect = dialects.get(database);
    if (dialect == null && database.equals(DatabaseManager.TEMP_TABLE_DB))
    {
       dialect = TemporaryDatabaseManager.getMyTempDbDialect();
       dialects.put(DatabaseManager.TEMP_TABLE_DB, dialect);
    }

    return dialect;
}
```

#65 - 09/16/2020 05:16 AM - Adrian Lungu

Just rebased 4701a using 3821c.

#66 - 09/17/2020 01:36 PM - Eric Faulhaber

Adrian, thanks for the explanations. Greg, I am good with the update going into 3821c.

#67 - 09/17/2020 02:50 PM - Greg Shah

Adrian: Go ahead and merge into 3821c. If there are any special requirements for rebuilding/deployment, please notify the team.

#68 - 09/18/2020 07:14 AM - Adrian Lungu

Done. Committed in 3821c as revision 11549.

#69 - 09/18/2020 02:33 PM - Eric Faulhaber

Adrian Lungu wrote:

> Done. Committed in 3821c as revision 11549.

Adrian, I don't see your commit in 3821c. 11549 is a commit from Constantin. Was your branch bound when you did the commit?

#70 - 09/19/2020 07:16 AM - Adrian Lungu

Probably the branch was not bound. I did the commit now as revision 11553.

#71 - 09/20/2020 03:51 PM - Greg Shah

Adrian: Please document the configuration changes to enable the separate database mode.

#72 - 09/21/2020 09:36 AM - Adrian Lungu

In directory.xml, the following empty container node should be added (usually right before the database specific container node): <node class="container" name="private-temp-dbs"/>. The logger will print: Using private (per-session) temporary databases..

In case the shared temporary database is desired, omit the specified note and the logger will print: Using public (shared) temporary database..

#73 - 11/29/2021 12:27 PM - Eric Faulhaber

FYI, H2 version 2.0.202 was released November 25, 2021. I noticed these two items which are relevant to us in the change log:

- PR 3124: Remove PageStore engine
- PR 2733: Replace h2.sortNullsHigh with DEFAULT_NULL_ORDERING setting

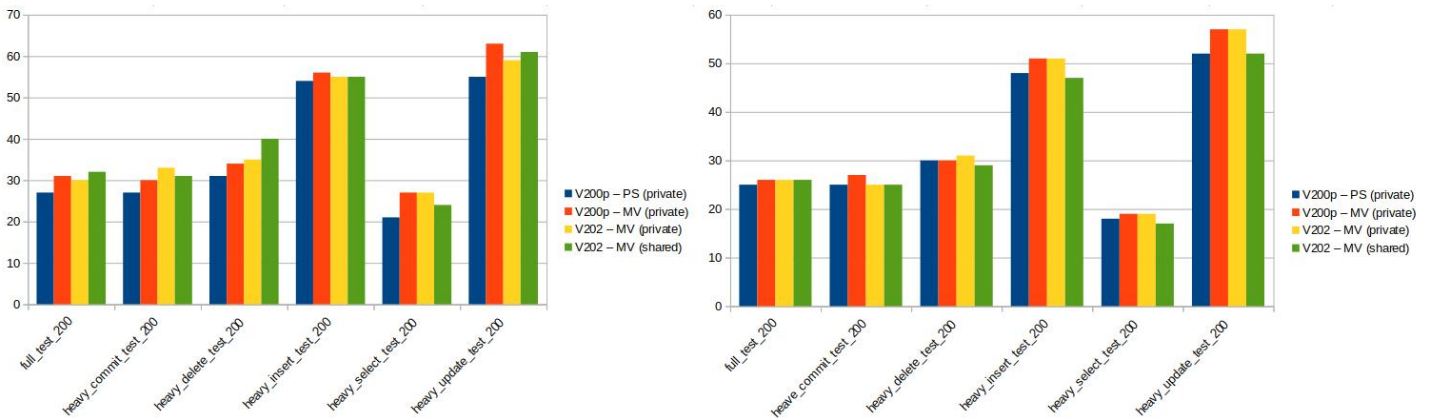
The first one at least is not good news for us. It means we are stuck on our current version of H2 until we determine the performance of the MVStore engine in this or some future version is at least as good the PageStore engine in the release we are using... or we hit critical bugs that are fixed in newer versions, the fixes for which we cannot do without... or the performance of the rest of FWD is improved to such a degree that the performance hit of switching to MVStore is not as important as it is now.

#74 - 12/08/2021 06:02 AM - Alexandru Lungu

- File profile_202.png added

- Assignee changed from Adrian Lungu to Alexandru Lungu

I did a quick profiling for H2's new release (202) (cold tests left, warm tests right):



The new release doesn't bring a real performance improvement to justify the switch to v202 and MVStore. While the cold tests clearly show that the private pagestore is still out-performing the other candidates, the warm tests show a slight preference for v202 shared MVStore. I don't think that, at this point, we should upgrade to v202 (unless one of Eric's point in #4701-73 is met - critical bug / performance improvement).

#75 - 12/21/2021 11:51 AM - Eric Faulhaber

https://groups.google.com/d/msgid/h2-database/18e458764a6d810175c894090e53b181541cfebd.camel%40manticore-projects.com

Another data point, though this is more about platform comparisons. It's hard to put this into context, because:

- this is on one particular laptop; and
- I have no idea what this benchmark does with H2 specifically, so it's hard to know whether this overlaps with FWDs use patterns.

#76 - 07/27/2022 03:33 AM - Alexandru Lungu

I just noticed that H2 is moving kinda fast with no more than 5 releases in the last couple of months. I can't tell if we can rely on one of these latest releases, especially if there is an effort to change to MVStore engine. There is some issue fixing, but mostly it is irrelevant for our use case (or maybe not?). I suggest to keep an eye on the evolution of the H2 releases and eventually hope for a stable and efficient release in the near future.

#77 - 08/17/2022 10:57 AM - Alexandru Lungu

- Related to Support #6679: H2 general performance tuning added

#78 - 01/16/2023 02:38 PM - Greg Shah

Alexandru:

Is #4701-72 still the correct way to enable private temp-table databases?

How to we enable LOCK_MODE=0?

#79 - 01/16/2023 03:28 PM - Eric Faulhaber

Greg Shah wrote:

> Alexandru:
>
> Is #4701-72 still the correct way to enable private temp-table databases?
>
> How to we enable LOCK_MODE=0?

You get it automatically when you add the following under server/standard/ in the directory:

```
        <node class="container" name="private-temp-dbs"/>
```

BTW, I think this configuration is better placed under the persistence node than just on its own in the server account, but that seems to be how it is implemented now, at least in a large GUI application.

#80 - 01/17/2023 03:40 AM - Alexandru Lungu

Eric Faulhaber wrote:

> Greg Shah wrote:
>
> > Alexandru:
> >
> > Is #4701-72 still the correct way to enable private temp-table databases?
> >
> > How to we enable LOCK_MODE=0?
>
> You get it automatically when you add the following under server/standard/ in the directory:

Exactly; for 6129b, LOCK_MODE=0 is automatically appended to the connection string when using private temp-databases.

> BTW, I think this configuration is better placed under the persistence node than just on its own in the server account, but that seems to be how it is implemented now, at least in a large GUI application.

We can do this switch. However, 6129b and trunk are already using private temp-databases differently (with and without LOCK). Introducing this now makes the gap larger (trunk expects the configuration in the server account; 6129b expects the configuration in persistence). Shall we wait to have a common modification point, or shall I move on with the configuration change for 6129b now?

#81 - 01/17/2023 03:38 PM - Eric Faulhaber

We can wait. I just noticed this and it is not a critical change, just a preference. I think it is more intuitive to have this configured within the persistence node, since it is a persistence feature.

For that matter, if it consistently performs better, we want private temp databases to be the default, and then the configuration would change to a means of disabling that default (perhaps for more memory-constrained environments). So, that would mean more change to the configuration.

#82 - 01/19/2024 01:23 AM - Eric Faulhaber

Is there anything left to do on this task, or can we consider the goals met?

#83 - 01/19/2024 03:22 AM - Alexandru Lungu

This can be closed.

#84 - 01/19/2024 10:49 AM - Eric Faulhaber

- Status changed from Review to Closed

Files

simple-stress.png	200 KB	07/02/2020	Adrian Lungu
200-stress.png	270 KB	07/03/2020	Adrian Lungu
chart-stress.png	27.6 KB	07/03/2020	Adrian Lungu
h2-197-patched.tar.gz	3.72 MB	07/03/2020	Adrian Lungu
h2-200-patched.tar.gz	3.63 MB	07/03/2020	Adrian Lungu
h2_1.4.200_synchronization_transactional_fix_20200703a.patch	8.16 KB	07/03/2020	Adrian Lungu
parsed_log.zip	214 KB	07/07/2020	Adrian Lungu
h2_1.4.200_local_temp_table_fix_20200710.patch	17 KB	07/10/2020	Adrian Lungu
Session.java	66.2 KB	07/11/2020	Adrian Lungu
chart-mv-vs-page.png	27.4 KB	07/14/2020	Adrian Lungu
profiling_mv_vs_page.png	39.5 KB	07/14/2020	Adrian Lungu
h2_1.4.200_meta_lock_fix_20200727.patch	17.7 KB	07/27/2020	Adrian Lungu
shared_vs_private.png	19.5 KB	08/05/2020	Adrian Lungu
copy_scenario.png	36.9 KB	08/09/2020	Adrian Lungu
create_scenario.png	40.5 KB	08/09/2020	Adrian Lungu
delete_scenario.png	35.5 KB	08/09/2020	Adrian Lungu
read_scenario.png	36.5 KB	08/09/2020	Adrian Lungu
create_scenario_parallel.png	44.4 KB	08/15/2020	Adrian Lungu
profile_202.png	43 KB	12/08/2021	Alexandru Lungu