Database - Feature #7061

Enable the use of lazy result sets in H2

01/23/2023 04:40 AM - Alexandru Lungu

Status:	Closed	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Radu Apetrii	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

Related issues:

Related to Database - Feature #6582: implement multi-table AdaptiveQuery	**WIP**
Related to Database - Feature #7066: Implement multi-table indexed query in H2	**WIP**
Related to Database - Bug #7047: Problem with AdaptiveQuery and ScrollableRes...	**New**

History

#1 - 01/23/2023 05:36 AM - Alexandru Lungu

Prerequisites
FWD converts most of the index-sorted single-table FOR queries into AdaptiveQuery. With the effort from #6582, we should expect to see multi-table FOR queries also converted as AdaptiveQuery.

The limitation of such queries is the invalidation lifecycle. This means that:

- It can't decisively retrieve all records as invalidation can occur, so many fetched records will be dropped. [USELESS FETCHING]
- It can't iterate the records one by one due to the high number of interactions with the database. [DATABASE STRESSING]

To handle this, we use ProgressiveResults that tries to be the best of both worlds (using exponentially growing LIMIT clause). Even so:

- usually it generates more than 1 SQL to retrieve the full set of records.
- invalidation will always mean dropping some fetched records.
- most of the queries will use OFFSET X. This keyword won't magically skip X records, but it will still iterate the index until the X-th matching record is found.

Finally, an invalidated single-table AdaptiveQuery transforms into a RandomAccessQuery that by nature does DATABASE STRESSING. This isn't extremely bad as revalidation usually kicks in pretty fast, but it should be taken into account.

Idea for indexed single-table queries
We should use something like a H2 server cursor allowing us to fetch the next record only if we need it without executing a whole new SQL. This way we fetch exactly the records we need, when we need them. Keeping a "database cursor* avoids the whole SQL generation, parsing, execution, row skipping, etc. and it will magically allow us to fetch the next record on demand. Of course, this should work only if we traverse an index to guarantee the ORDER BY clause.

Technically, this should work for multi-table queries too. Multi-table AdaptiveQuery implementation heavily relies on having such optimization. The downside is that H2 doesn't yet have a multi-table index selection based on the ORDER BY clause, which should be implemented first.

A major conceptual issue is the "invalidation" of lazy fetching. Basically, what happens if we insert/update/delete records in the queried table?

- We either force all lazy results to do an eager selection of all records before the update. This is a complete solution, but the whole effort of avoiding extra fetching is lost.
- We close the result-set and it is FWD's responsibility to handle this. This is a sound solution and we don't do USELESS FETCHING. However, invalidation is not always happening when a change is made to the table, so FWD may reach an awkward state: wanting to retrieve the next record of a closed result-set.
- We leave the lazy result-set as it is. This looks like a very risky solution. However, in our current implementation, AdaptiveQuery doesn't do anything special anyways when the query is not invalidated even if a record is inserted/updated/deleted. So most probably, continuing using a "database cursor" which wasn't invalidated from the FWD side should be safe enough. **This works as long as we do lazy fetching conditionally**

Implementation
Adopting this idea means replacing ProgressiveResults with ScrollingResults backed by H2-level lazy fetching. I don't think that we need a new type of query or FWD Results implementation. H2 already has org.h2.result.LazyResult, but it is mainly used internally. Maybe we can enable it as a JDBC

ResultSet and make use of the lazy fetching as a mean of implementing a database cursor. This way we can transparently allow H2 to do the lazy fetching stuff. The only concern is that we may want to have this lazy fetching conditionally, so that PreselectQuery can do the whole fetching at once. Maybe do some syntax for this, like SELECT [...] ORDER BY [...] LAZY and SELECT [...] ORDER BY [...] PRESELECT, with PRESELECT being the default?

#2 - 01/23/2023 05:36 AM - Alexandru Lungu

- Related to Feature #6582: implement multi-table AdaptiveQuery added

#3 - 01/24/2023 06:13 AM - Alexandru Lungu

- Related to Feature #7066: Implement multi-table indexed query in H2 added

#4 - 01/25/2023 05:35 AM - Eric Faulhaber

Alexandru Lungu wrote:

> Adopting this idea means replacing ProgressiveResults with ScrollingResults backed by H2-level lazy fetching.

Have you tested this replacement on its own (i.e., without the lazy fetching)? I'm curious whether it makes a difference with real applications.

ProgressiveResults was developed to deal with an idiom where one or a small number of results was iterated and then the rest discarded with say, a conditional or unconditional LEAVE in the FOR EACH loop. However, although we saw this idiom in real application code, I don't know how prevalent it is, and the assumption was that we didn't pay a huge penalty for using LIMIT/OFFSET for pages/brackets of results after the first. You've since found the latter assumption to be wrong, at least for H2.

I ask because we need a generally performant, default approach to use not only for temp tables with H2 (where we can implement lazy fetching), but also for persistent tables in the primary databases.

#5 - 01/25/2023 06:11 AM - Alexandru Lungu

Eric Faulhaber wrote:

> Alexandru Lungu wrote:
>
>> Adopting this idea means replacing ProgressiveResults with ScrollingResults backed by H2-level lazy fetching.
>
> Have you tested this replacement on its own (i.e., without the lazy fetching)? I'm curious whether it makes a difference with real applications.

I used only dummy testcases in #6582-33 and #6582-35 (compare PreselectQuery with AdaptiveQuery, as the first uses ScrollingResults and the second ProgressiveResults). I have a summary below. I will do some tests with real applications.

> ProgressiveResults was developed to deal with an idiom where one or a small number of results was iterated and then the rest discarded with say, a conditional or unconditional LEAVE in the FOR EACH loop. However, although we saw this idiom in real application code, I don't know how prevalent it is, and the assumption was that we didn't pay a huge penalty for using LIMIT/OFFSET for pages/brackets of results after the first. You've since found the latter assumption to be wrong, at least for H2.

ProgressiveResults are used exclusively by AdaptiveQuery (currently single-table) only when no resort is required. This means that ProgressiveResults are used only in LIMIT queries over indexed ORDER BY. These are decent in performance, because usually an index is used for traversal, so the LIMIT is resolved very fast.

Instead of LEAVE use-case, I think the invalidation use-case occurs more often. We select only a slim number of records at first to avoid USELESS FETCHING. The risk of using ScrollingResults straight-away is that we may fetch tens of records and drop them right after because we invalidated.

> I ask because we need a generally performant, default approach to use not only for temp tables with H2 (where we can implement lazy fetching), but also for persistent tables in the primary databases.

The primary databases, at least the ones handled by PostgreSQL, are quite good at solving LIMIT. I had an experiment in #6582-35 with PostgreSQL 14 and it really proved a point with its plan to solve LIMIT fast. At worst, ProgressiveResults was 27% slower than ScrollingResults iterating a whole result-set of one million records. In tables with thousands of records, the slow-down was only 8%. Invalidating on ScrollingResults means we lose 50% of the time on a query selecting records we actually drop afterwards. Of course, we can't know beforehand if the query is invalidated, so we pick the ProgressiveResults "just in case".

So, moving to ScrollingResults (without LIMIT) will give us a small boost when iterating the whole result-set, but it will be disastrous if we invalidate early (unless it is lazy fetching). At best, we can make this configurable. I've seen large customer applications with few invalidations (maybe tens in a POC) and others with many invalidations (hundreds in a POC).

#6 - 01/30/2023 09:52 AM - Alexandru Lungu

- Assignee set to Radu Apetrii

- Status changed from New to WIP

#7 - 02/07/2023 07:23 AM - Alexandru Lungu

- Related to Bug #7047: Problem with AdaptiveQuery and ScrollableResults added

#8 - 02/14/2023 06:48 AM - Radu Apetrii

- File 7061-fwd.patch added

I've added the option to execute queries in a lazy manner. Alongside this change, AdaptiveQuery now uses only ScrollingResults (I have removed ProgressiveResults). This was done because ScrollingResults with lazy performs better than the current implementation of ProgressiveResults.

The code for H2 was committed to 7061a_h2, rev.10. The patch I have attached is for FWD, 6129c, rev. 14808.

I have to mention that **these changes are still in need of testing and may suffer modifications.** I've tested all from adaptive_scrolling folder, the H2 tests from #7058-12 and some other tests that I have created myself. Even though the results were promising, **I still encounter a problem when running a large customer application.**

The plan is to continue the testing process and fix the issues that appear.

#9 - 02/15/2023 08:51 AM - Radu Apetrii

- File 7061-2-fwd.patch added

Update: I've committed a fix to 7061a_h2, rev. 12. **I no longer have issues when testing a large customer application**. Also, the other tests execute correctly.

I modified the patch from the last post a bit, because the change with the removal of ProgressiveResults should've happened only in temporary cases. I worked with 6129c, rev. 14808.

If you encounter any further errors, please let me know.

#10 - 02/15/2023 01:54 PM - Eric Faulhaber

Radu Apetrii wrote:

> I modified the patch from the last post a bit, because the change with the removal of ProgressiveResults should've happened only in temporary cases. I worked with 6129c, rev. 14808.

One clarification to this point. It is not so much the fact that the new lazy mode should be used for temp-table cases, but rather that it should be used for the H2 dialect, which is the only dialect that supports this mode, due to our changes within H2. I haven't looked at the implementation within H2, but can we say that it is further limited to embedded, in-memory use cases (as opposed to a separate H2 server process, connected using TCP/IP)? Either way, we should define the condition whereby lazy results mode is chosen more specifically.

Currently, the condition in AdaptiveQuery is:

```
      boolean lazyMode = persistence.isTemporary();
```

If the dependency is the H2 dialect on its own, then I think the condition should be something like:

```
      boolean lazyMode = persistence.getDialect().supportsLazyQueryResultsMode();
```

...assuming Dialect.supportsLazyQueryResultsMode() is implemented to return false by default and true for the H2-specific subclass.

If the dependency is an H2 database running in-process (embedded) and in-memory, but not as a standalone server in a separate JVM process, then I think the condition should be something like:

```
      boolean lazyMode = persistence.supportsLazyQueryResultsMode();
```

...where Persistence.supportsLazyQueryResultsMode() is implemented something like this:

```
   return dialect.supportsLazyQueryResultsMode() && (temporary || database.isMeta() || database.isDirty());
```

This change does two things:

- it expands the use of lazy results mode to the meta and dirty H2 embedded database instances; and
- it ties the use of lazy mode to only the dialect(s) that support it (currently only our forked H2).

The second point may seem redundant, since we are tightly coupled to H2 for temp-table support in so many other ways, but it lowers technical debt, should we ever have to disentangle this coupling and use some other dialect to support temp-tables. It was not long ago I was considering moving temp-table support back into the primary database, at least for some temp-tables, to enable server-side joins between persistent and temporary tables (avoiding CompoundQuery for those cases). Instead, we have gone the other direction.

#11 - 02/16/2023 09:34 AM - Alexandru Lungu

- % Done changed from 0 to 80

Eric Faulhaber wrote:

> One clarification to this point. It is not so much the fact that the new lazy mode should be used for temp-table cases, but rather that it should be used for the H2 dialect, which is the only dialect that supports this mode, due to our changes within H2. I haven't looked at the implementation within H2, but can we say that it is further limited to embedded, in-memory use cases (as opposed to a separate H2 server process, connected using TCP/IP)? Either way, we should define the condition whereby lazy results mode is chosen more specifically.

I think that the implementation is not necessarily bound to any H2 mode (in-memory/persistent/server):

- However, I expect that the non-embedded mode (separate process) is not ideal for lazy as it should communicate each record "one-by-one". Doing 100 next will actually generate 100 requests to retrieve the records. In non-lazy mode, the whole result-set is retrieved at once in memory.
- Also, I won't go with lazy statements in concurrent environments anyways. So I think lazy should not be used with persistent or server H2.

> If the dependency is the H2 dialect on its own, then I think the condition should be something like:
> ...assuming Dialect.supportsLazyQueryResultsMode() is implemented to return false by default and true for the H2-specific subclass.

Makes sense.

> This change does two things:
>
> - it expands the use of lazy results mode to the meta and dirty H2 embedded database instances; and
> - it ties the use of lazy mode to only the dialect(s) that support it (currently only our forked H2).

meta and dirty databases are multi-user (correct me if I am wrong). Extending current lazy implementation to cover concurrent environments doesn't look easy (for now).

As the patch and H2 modifications are mostly complete, in the sense that I have no obvious regression in large customer applications, I started a profiling session. The results are not quite in the expected direction as I encounter a +0.4% time increase. I will do some profiling to identify the bottlenecks here. Anyways, as lazy is stable enough for temp-tables, I think we can go ahead and disregard invalidation of AdaptiveQuery for temp-tables. The lazy traversal is already handling index reconstruction, so there is no need to go into RAQ mode to revalidate back. I think this can do trick here.

#12 - 02/17/2023 09:49 AM - Alexandru Lungu

Alexandru Lungu wrote:

> As the patch and H2 modifications are mostly complete, in the sense that I have no obvious regression in large customer applications, I started a profiling session. The results are not quite in the expected direction as I encounter a +0.4% time increase. I will do some profiling to identify the bottlenecks here. Anyways, as lazy is stable enough for temp-tables, I think we can go ahead and disregard invalidation of AdaptiveQuery for temp-tables. The lazy traversal is already handling index reconstruction, so there is no need to go into RAQ mode to revalidate back. I think this can do trick here.

I've done an error while profiling. I've rebased now 7061a_h2 to the latest trunk to match by baseline. The lazy mode is highly dependent upon having as few cursor reconstructions as possible. Our recent H2 UPDATE improvement reduced the number of cursor reconstructions a lot, so the lazy mode is now more suitable.

After redoing the tests, allowing lazy only for non-scrolling queries and avoiding invalidating AdaptiveQuery for lazy H2 queries, I've got a -0.2% time drop. This is not the best, but it is in the right away now.
Eric, as the modifications prove to increase performance (even by a bit), I think we can also include this change (7061a_h2) into our next FWD-H2 build. This way, we can work exclusively in FWD to make the most out of it.

Finally, my concern now is that:

- all dynamic queries (generated by CREATE QUERY) are SCROLLING by default, so this lazy optimization only targets static non-scrolling queries. Ugh :(
- we should cut-out the whole query registering as listener for ChangeSet if the query is lazy. I just avoided invalidation in my tests, not the whole change-event reporting overhead for AdaptiveQuery.
- **a review for the changes by now**:
    - there is a truncate method for indexes in H2. This method means "clear", so all nodes should be removed/inactivated. For the lazy part, this should be handled, as right now, the index cursor still traverses the stale tree. This doesn't affect FWD at all (because truncating is due to some unused features from H2), but should be fixed at some point without any overhead. If this can't be easily handled or may take some time, please document this issue in the H2 code where truncate is used - we can handle it latter.
    - I think there are some weird cases where equal tree nodes (records matching on the indexed columns) are not iterated correctly. If a node is removed, I think your lazy cursor reconstruction will skip all records which were equal to that removed node. This is a rare use-case, but exists in large customer applications. Please test this use-case.
    - Check if there is an optimization opportunity for UPDATE which doesn't change any data. Basically, skip updating if oldRow.equals(newRow) and avoid lazy cursor invalidating.
    - For lazy mode, in TreeCursor.previous you have something like node.active straight-away. Is there a case where you have a null node? Maybe when you are out of the result-set? My point here is that, previous is quite rare, so it is hard to find customer examples to test your implementation. Therefore, make sure we avoid awkward situations like NPE straight from H2.

#13 - 02/20/2023 03:32 AM - Alexandru Lungu

Radu:

- I've created 7061a branched from trunk to commit your changes to.
- Please notify me once you address the review and finish 7061a_h2. I am planning to merge it to the H2 trunk.
- As further work, consider doing some profiling for your use-case: mainly FWD, but also FWD-H2.
  - Extract any obvious bottle-neck
  - Check how many prev operations are done in a large application; note than non-lazy queries have a very slow "prev" performance in H2 (reset and do several next).
  - Check if by making **all** AdaptiveQuery queries TYPE_SCROLL_INSENSITIVE affects performance!!
  - Count how many actual lazy queries are there in a large application: static vs dynamic, scrolling vs non-scrolling, temp-table vs persistent, preselect vs adaptive vs compound. I am curious of a percentage of the actual queries that are lazy. Hopefully, >20% overall are lazy queries for now. If not, we need to move on with supporting scrolling queries as well - ugly :(

#14 - 02/22/2023 03:44 AM - Radu Apetrii

Alexandru Lungu wrote:

> If the dependency is the H2 dialect on its own, then I think the condition should be something like:
> ...assuming Dialect.supportsLazyQueryResultsMode() is implemented to return false by default and true for the H2-specific subclass.
>
> Makes sense.

I've added this function to Dialect and P2JH2Dialect and I've changed the condition in AdaptiveQuery.executeQuery.

> - we should cut-out the whole query registering as listener for ChangeSet if the query is lazy. I just avoided invalidation in my tests, not the whole change-event reporting overhead for AdaptiveQuery.

I've added a check in AdaptiveQuery.initialize. Now the registration is conditioned by the Dialect.supportsLazyQueryResultsMode function.

> - I think there are some weird cases where equal tree nodes (records matching on the indexed columns) are not iterated correctly. If a node is removed, I think your lazy cursor reconstruction will skip all records which were equal to that removed node. This is a rare use-case, but exists in large customer applications. Please test this use-case.
> - Check if there is an optimization opportunity for UPDATE which doesn't change any data. Basically, skip updating if oldRow.equals(newRow) and avoid lazy cursor invalidating.

These two points were tied together. Modifying the UPDATE allowed me to get rid of that while that skipped through records.

> - For lazy mode, in TreeCursor.previous you have something like node.active straight-away. Is there a case where you have a null node? Maybe when you are out of the result-set? My point here is that, previous is quite rare, so it is hard to find customer examples to test your implementation. Therefore, make sure we avoid awkward situations like NPE straight from H2.

I've added null checks to make sure these things won't happen.

The changes were committed to 7061a, rev. 14485 for FWD and 7061a_h2, rev. 17 for H2. This is more of an update, since there are other points that I'm yet to complete.

#15 - 02/22/2023 11:23 AM - Alexandru Lungu

I reverted your latest changes to H2 because I wanted to merge a tested, profiled version (rev. 15).

- The null checks are right, but are not required right now. As a matter of fact, unless #7047 is fixed, the null checks won't be even reached; nevertheless to say, even if #7047 is fixed, this is a very specific edge case.
- The while skipping similar records is again a very specific use-case. I am aware that this is some sort of regression, but rushing to fix it now may cause other performance issues (which may be worse than lazy altogether). My point is that, to fix this regression we just need to eliminate the possibility to "fake-update" records (UPDATE tt SET f1 = 2 WHERE f1 = 2). To summarize, the regression happens only when there are at least 2 equal records, lazy mode is active, fake-update happens on an indexed field, one continues to use the lazy cursor and there is an exact need of traversing similar records. This couldn't be caught in large applications, so I think we can leave it as it is right now and open a separate task regarding the short-circuit of fake-update. This way, we don't bother having lazy specific logic for this use-case, as long as we can eliminate fake-update.

This being said, I merged 7061a_h2 into H2 trunk as rev. 12.

#16 - 02/24/2023 04:21 AM - Alexandru Lungu

Alexandru Lungu wrote:

> - Count how many actual lazy queries are there in a large application: static vs dynamic, scrolling vs non-scrolling, temp-table vs persistent, preselect vs adaptive vs compound. I am curious of a percentage of the actual queries that are lazy. Hopefully, >20% overall are lazy queries for now. If not, we need to move on with supporting scrolling queries as well - ugly :(

I think this is the last point to cover here. Most important is to identify if supporting "lazy 4GL scrolling queries" may actually help. Let's say that if >30% of the adaptive queries over temp-table are for 4GL scrolling queries, we should really consider integrating SCROLLING queries for LAZY. However, there is a complex logic behind the cursor/caching that is done over scrolling queries, so we need to be sure that the time spent on supporting lazy scrolling is worth it.

#17 - 02/24/2023 05:46 AM - Radu Apetrii

Alexandru Lungu wrote:

> Alexandru Lungu wrote:
>
> > - Count how many actual lazy queries are there in a large application: static vs dynamic, scrolling vs non-scrolling, temp-table vs persistent, preselect vs adaptive vs compound. I am curious of a percentage of the actual queries that are lazy. Hopefully, >20% overall are lazy queries for now. If not, we need to move on with supporting scrolling queries as well - ugly :(
>
> I think this is the last point to cover here. Most important is to identify if supporting "lazy 4GL scrolling queries" may actually help. Let's say that if >30% of the adaptive queries over temp-table are for 4GL scrolling queries, we should really consider integrating SCROLLING queries for LAZY. However, there is a complex logic behind the cursor/caching that is done over scrolling queries, so we need to be sure that the time spent on supporting lazy scrolling is worth it.

I've done a little investigation and counted how many adaptive queries over temp-tables are scrolling:

- For a large customer application in which I spent 15-20 minutes, **only ~2% (114 / 5693) of those queries were scrolling**.
- In the hotel app, ~30% of the queries were scrolling.

I'll wait for a final decision on this.

#18 - 02/24/2023 06:02 AM - Alexandru Lungu

> I'll wait for a final decision on this.

Well, it definitely looks like it really depends on the converted code. I will also do a test on some large customer applications to have a wider overview.

I have thought about multi-table AdaptiveQuery for a while now. Such structure can help us compute joined statements for FOR-EACH queries and still honor the invalidation/revalidation cycle.

AFAIK, the EACH components of a CompoundQuery are converted to AdaptiveQuery, so if they are executed lazily, we basically have a multi-table AdaptiveQuery which honors invalidation/revalidation cycle right in H2.
I can't imagine if a joined query (with invalidation/revalidation cycle) is faster than a server-side managed query working over database cursors. Worst-case, we can think of supporting lazy joined statements!
We can "revive" #6582 at this point and continue our discussion there.

#19 - 02/27/2023 03:15 AM - Alexandru Lungu

Radu, I merged your changes from 7061a to 7026a. Note that 7026a is frozen right now.

First of all, I moved supportsLazyQueryResultsMode at the Persistence level to disallow its use for meta and dirty which are **concurrent** environments. Allowing it for the H2 dialect may break meta and dirty.

Secondly, I added a scrolling check to disallow scrolling queries to use lazy.

Further in my tests, I've got a crash when checking the dialect of the buffer in AdaptiveQuery.initialize, saying that the buffer is not active due to RecordBuffer.checkActive. Please check if this also happens for you with a large application using 7061a changes. Otherwise, I will try to recreate it in a smaller example.

That being said, as a temporary solution, I've commented your changes and added a short-cirucit in AdaptiveQuery.stateChanged which checks if the query is not scrolling and temporary to cut-out invalidation. This is not ideal, because it doesn't use Persistence.supportsLazyQueryResultsMode and still allows the listener registration.

Please review my changes in 7026a and continue making a patch for 7026a with the fix for my temporary solution - **don't commit**.

#20 - 03/01/2023 03:46 AM - Radu Apetrii

- File 7061-3.patch added

Alexandru Lungu wrote:

> Further in my tests, I've got a crash when checking the dialect of the buffer in AdaptiveQuery.initialize, saying that the buffer is not active due to RecordBuffer.checkActive. Please check if this also happens for you with a large application using 7061a changes. Otherwise, I will try to recreate it in a smaller example.

Here's a patch with the changes. As you pointed out, in AdaptiveQuery.initialize a buffer might not have been active when the registration happened, thus resulting in an exception being thrown.
Initially, I tried to move the registration in AdaptiveQuery.execute, but it was too late in the program and some examples failed.
In this patch, the registration happens in AdaptiveQuery.open. Although it still seems a bit too early, I believe there is access to a buffer due to the call of getRecordBuffers() that happens at the start of the method.

Alex, please test the example that failed as I was not able to reproduce it and let me know if this fixed the problem. Thank you!
The patch was done with 7026a, rev. 14507.

#21 - 03/01/2023 05:19 AM - Alexandru Lungu

This is not the way to do it. AdaptiveQuery.open is called only by explicit queries, usually dynamic (QUERY-OPEN / OPEN QUERY). On the other hand, AdaptiveQuery.initialize is called by implicit queries (FOR EACH). That is why we have different registrations:

- explicit adaptive queries can be defined/created in a scope, prepared in another scope (dynamic) and opened elsewhere. We register at opening as there is no initialization.
- implicit adaptive queries are generated for some loops like FOR EACH. We register at initialization as there is no OPEN.

Your change removes the registration for implicit adaptive queries. It also adds a second registration for explicit queries.

If you are testing with adaptive-scrolling from testcases/uast, they mostly have explicit queries, so this change of yours may look sound. I think it is best for you, if you continue working with AdaptiveQuery, to create some examples with implicit queries (generated by FOR EACH). Large customer applications rarely rely on reiterating records with FOR EACH, so you may not see an obvious regression when testing. However, not doing the things right (invalidating) may cause very subtle regressions.

As the overhead of changing the listener registration system for AdaptiveQuery seems risky and may cause regressions too easily, I think it is best to take a step back and use your first approach. You can use RecordBuffer.isActive to make sure you are able to access buffer's persistence without an exception being thrown. Because you should handle for explicit/implicit query cases, I think it is best to use registerRecordChangeListeners in both scenarios. Make scope as Integer. Having it null, means registration at ChangeBroker with no depth.

Sorry for messing you around the AdaptiveQuery change broker registration for so long. However, having a solid invalidation system and the gain of avoiding any listener registration saves us precious time of RecordChangeEvent generation overhead.

#22 - 03/02/2023 05:13 AM - Alexandru Lungu

I've done some tests with another large customer application and I have the following statistics:

Type	Temporary	Other
Scrolling	1.579	5.415
Not Scrolling	6.782	1.130

The numbers are the AdaptiveQuery.execute execution count, not the next count.
Therefore, 18% of the AdaptiveQuery over temp-tables are scrolling. This is not much, but significant enough.
From a profiling test, the lazy keyword shown a 0.2% boost on the 6.782 non-scrolling queries and a further 0.5% boost on the 1.579 scrolling queries. My point is that, there are less scrolling queries, but they happen to consume more time on average in the old ProgressiveResults.

Radu, please confirm that your lazy solution is eligible for the scrolling queries as well.

#23 - 03/02/2023 09:45 AM - Radu Apetrii

Alexandru Lungu wrote:

> Radu, please confirm that your lazy solution is eligible for the scrolling queries as well.

I ran a series of tests and the results look very promising:

- In adaptive_scrolling there were some tests (>10) that did not match the result obtained in 4GL. This is no longer the case, most of them (>80%) output the correct result. Also, there are no tests that were correct before and now are incorrect.
- I did not find any problems when running a large customer application.
- Again, no errors when testing various cases with reposition. I've tried all sorts of repositions to records that: were/weren't in cache, were forwards/backwards relative to the current one, get/don't get updated and combinations of these cases.

Long story short, when repositioning, if a record is in the cache, then it is immediately retrieved. Else, it computes the number of records it has to "jump over", whilst "peeking at them" (AdaptiveQuery.peekNext or AdaptiveQuery.peekPrevious). **This means that the lazy TreeCursor also gets moved around accordingly**.

Based on the results I've seen, I'm looking forward to see this integrated. For the moment, the changes are on 7061a, rev.14486.

#24 - 03/07/2023 08:59 AM - Alexandru Lungu

Radu, is the multi-table AdaptiveQuery considered for your implementation? I understood that lazy keyword works only for single-table queries. However, it looks like we avoid invalidation for any query (not only single-table). If this is the case, please commit a fix for this in 7061a to allow lazy keyword and invalidation suppression only for single-table AdaptiveQuery.

#25 - 03/09/2023 05:00 AM - Radu Apetrii

Alexandru Lungu wrote:

> Radu, is the multi-table AdaptiveQuery considered for your implementation? I understood that lazy keyword works only for single-table queries. However, it looks like we avoid invalidation for any query (not only single-table). If this is the case, please commit a fix for this in 7061a to allow lazy keyword and invalidation suppression only for single-table AdaptiveQuery.

AFAIK, the only way an AdaptiveQuery can become multi-table comes from CompoundQuery, when it decides to merge two components. Thus, a multi-table AdaptiveQuery:

- Gets registered as a listener when the merging happens.
- Is not executed lazy because there is a check in H2 that prevents that from happening.

Still, I've added a check in AdaptiveQuery.initialize just to make sure that everything is fine. I've committed this to 7061a, rev. 14488.

#26 - 03/09/2023 05:38 AM - Alexandru Lungu

- % Done changed from 80 to 100

- Status changed from WIP to Review

Rebased 7061a to trunk (rev. 14497). Planning to merge your latest changes to 7026a as they target a temporary fix from rev. 14517 in 7026a.

#27 - 03/09/2023 05:52 AM - Alexandru Lungu

- File change_broker_lazy.patch added

Radu, this is the patch I am planning to integrate into 7026a to wrap-up the effort from #7061 and have a complete implementation making it to trunk. Please review it closely.
I am going to test it again with two large applications and do a quick profiling before committing.

#28 - 03/09/2023 07:46 AM - Alexandru Lungu

- File change_broker_lazy_update.patch added

After a self-review of the patch, I see that we still register the change broker for AdaptiveQuery.open and there is a regression with the registration itself as is doesn't override Joinable.registerRecordChangeListeners properly.
I've done some extensive changes. I am uploading the new patch.

Tested with one large application and I don't see any regression yet (with this updated patch).

Note that I also added a short-circuit for CompoundQuery.registerRecordChangeListeners to avoid computing the registration scope for queries which don't require registration (PreselectQuery and AdaptiveQuery for temp database).

Radu, please review and test with your adaptive test cases (scrolling and non-scrolling).
UPDATE1 also debug and ensure that stateChanged and invalidate are never called in your test cases.
UPDATE2 please test the persistent test-cases and ensure they are working properly.

#29 - 03/09/2023 07:48 AM - Constantin Asofiei

Alexandru, is this patch meant to solve regressions? Because we are getting close to finish testing 7026a, and unless the change fixes known regressions in applications, please don't commit anything to 7026a.

#30 - 03/09/2023 07:54 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Alexandru, is this patch meant to solve regressions? Because we are getting close to finish testing 7026a, and unless the change fixes known regressions in applications, please don't commit anything to 7026a.

Short answer, no.
Long answer:

- I am aware that 7026a is close to merge and wanted to ask you if such a small "code esthetic" can make it.
- #7061-27 was meant to be a risk-free change to avoid the ugly code currently in 7026a: if (components.size() == 1 && components.get(0).getBuffer().isTemporary() && !isScrolling()) - which is in fact safe and doesn't regress
- After testing, it seems that it wasn't just a "code esthetic", but a wide functional change. I am not planning to commit it to 7026a anymore, but I still want it fixed for the next sprint.

#31 - 03/10/2023 04:22 AM - Radu Apetrii

Alexandru Lungu wrote:

> Radu, please review and test with your adaptive test cases (scrolling and non-scrolling).
> *UPDATE2* please test the persistent test-cases and ensure they are working properly.

The results were on point. No problems found.

> *UPDATE1* also debug and ensure that stateChanged and invalidate are never called in your test cases.

I got very few invalidate calls, but I don't think it's "panic time". They were obtained from multi-component AdaptiveQueries (a CompoundQuery with two AdaptiveQuery components that got merged), because they are not executed lazy. This is a feature that is currently being worked on, so I have no worries on this point either.

#32 - 03/15/2023 05:05 AM - Alexandru Lungu

Radu, please commit #7061-28 patch to 7026b if there are no obvious problems in your tests. I am looking forward to start a testing and profiling session. Thank you!

#33 - 03/15/2023 05:26 AM - Radu Apetrii

Alexandru Lungu wrote:

> Radu, please commit #7061-28 patch to 7026b if there are no obvious problems in your tests. I am looking forward to start a testing and profiling session. Thank you!

Done. Committed to 7026b, rev. 14504.
If there are any problems, let me know.

#34 - 03/23/2023 04:27 AM - Alexandru Lungu

- Status changed from Review to Test

#35 - 07/21/2023 12:32 PM - Eric Faulhaber

Since this has since been merged to trunk, can we close this issue now?

#36 - 07/24/2023 03:21 AM - Alexandru Lungu

This can be closed.

#37 - 07/24/2023 09:21 AM - Greg Shah

- Status changed from Test to Closed

Files

7061-fwd.patch	18.1 KB	02/14/2023	Radu Apetrii
7061-2-fwd.patch	17.4 KB	02/15/2023	Radu Apetrii
7061-3.patch	2.92 KB	03/01/2023	Radu Apetrii
change_broker_lazy.patch	3.2 KB	03/09/2023	Alexandru Lungu
change_broker_lazy_update.patch	13.4 KB	03/09/2023	Alexandru Lungu