Database - Feature #6695

Multi-table preselect query may underperform due to repetitive fetching

08/23/2022 12:18 PM - Alexandru Lungu

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Radu Apetrii	**% Done:**	30%
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 #6720: lazy hydration	**WIP**

History

#1 - 08/23/2022 12:19 PM - Alexandru Lungu

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

#2 - 08/23/2022 12:25 PM - Alexandru Lungu

There is a case in which the multi-table PreselectQuery works slower than a CompoundQuery:

```
def temp-table tt field f1 as int field f2 as int.
def temp-table tt2 field f1 as int field f2 as int.

def var i as int.
do i = 1 to 5000:
    create tt.
    tt.f1 = i.
    create tt2.
    tt2.f1 = i.
end.

open query q1 preselect each tt, each tt2. // also tried with for instead of preselect
GET FIRST q1.
do while available(tt):
    get next q1.
end.
```

There is already some discussion in #6582-19 and further. An initial profiling with VisualVM shows orm.BaseRecord.readProperty and orm.SQLQuery.hydrateRecord as hotspots. We suspect this is related to the fact that tt is fetched multiple times in preselect mode. In compound queries, each tt is fetched only once. A place to start:

- Find more realistic examples (using where such that less tt2 are associated with each tt on average)
- Investigate if there are such cross queries (closer to many-to-many than many-to-one) in large applications.
- Currently there is a session cache of 1,024 elements. Check if this cache or other cache can help sooner in the stack trace, such that we avoid running a lot of code to fetch/cache hit the same tt again and again.

#3 - 08/23/2022 06:25 PM - Eric Faulhaber

I wanted to see where the time was being spent in both the PreselectQuery case and the CompoundQuery case. I also wanted to see what the cost was when we have a lot of session cache misses.

So, I took your example as a starting point and made the following changes:

- instrumented it with:
  - a record counter
  - ETIME to measure elapsed time for 3 stages:
    - e1: time to run the equivalent of OPEN QUERY
    - e2: time to get only the first result
    - e3: time to iterate all remaining results
- Reduced total number of records created to 6,000 (3,000 each tt and tt2), to make the test slightly more manageable.
- Wrapped the record creation block in a DO TRANSACTION block, to enable UNDO tracking. This was to change the session cache behavior. When a DMO is being tracked for UNDO processing, it cannot be evicted from the session cache. This means all DMOs should cause cache hits during record hydration, unless UNDO processing is disabled by using NO-UNDO temp-tables.

The resulting program:

```
def temp-table tt  /*no-undo*/ field f1 as int field f2 as int.
def temp-table tt2 /*no-undo*/ field f1 as int field f2 as int.

def var c1 as int.
def var e1 as int.
def var e2 as int.
def var e3 as int.

def var i as int.

do transaction:
  do i = 1 to 3000:
    create tt.
    tt.f1 = i.
    create tt2.
    tt2.f1 = i.
  end.
end.

c1 = 0.
etime(yes).
open query q1 /*for*/ preselect each tt, each tt2.
e1 = etime(yes).
GET FIRST q1.
e2 = etime(yes).
do while available(tt):
    c1 = c1 + 1.
    get next q1.
end.
e3 = etime.
message "recs:" string(c1) "e1:" string(e1) "e2:" string(e2) "e3:" string(e3).
```

By toggling what was commented out, I ran 4 tests:

- UNDO-able temp-tables; PRESELECT query
- UNDO-able temp-tables; FOR query
- NO-UNDO temp-tables; PRESELECT query
- NO-UNDO temp-tables; FOR query

I also instrumented FWD with the attached patch (based on 3821c/14190), to let me know how many session cache hits and misses I was getting within SQLQuery.hydrateRecord for each case.

The results:

Temp-Table	PRESELECT	FOR
UNDO-able	recs: 9000000 e1: 6065 e2: 0 e3: 4298	recs: 9000000 e1: 0 e2: 7 e3: 3265
	hit/miss: 18000000/0; rate = 1.0; total = 18000000; cached: 6000	hit/miss: 9003000/0; rate = 1.0; total = 9003000; cached: 6000
NO-UNDO	recs: 9000000 e1: 4959 e2: 0 e3: 20749	recs: 9000000 e1: 0 e2: 6 e3: 7321
	hit/miss: 8997000/9003000; rate = 0.49983335;	hit/miss: 0/9003000; rate = 0.0; total =

	total = 18000000; cached: 1023	9003000; cached: 1023

The session cache data for the FOR cases is being reported for the outer component in the CompoundQuery:

```
from Tt_1_1__Impl__ as tt where (tt._multiplex = ?0)  order by tt._multiplex asc, tt.id asc
```

The data for the inner component was: hit/miss: 0/0; rate = 0.0; total = 0; cached: 0. The inner component query was:

```
from Tt2_1_1__Impl__ as tt2 where (tt2._multiplex = ?0)  order by tt2._multiplex asc, tt2.id asc
```

I initially wrote the instrumentation with a single PreselectQuery in mind (the variables where the data is collected are static), so these cache stats most likely are incorrect for a multi-query CompoundQuery.

Nevertheless, I can draw some conclusions:

1) The session cache misses are indeed expensive, as you concluded from the original test case. The e3 time (when iterating the results) are much higher when the records have to be hydrated from result set data, rather than retrieved from the session cache. The trade-off here is of course the additional memory needed to hold the DMOs in the cache.

2) It is not always obvious when DMOs will be available in the cache to avoid the hydration cost. I forced a situation with the test cases, such that I could control this, but in a real world application, the situation can vary. In fact, in the original test case, without the forced UNDO processing caused by the addition of the DO TRANSACTION block, we most likely were seeing the result of having far fewer DMOs in the cache, and therefore seeing more time spent in hydration.

3) A significant amount of time in the PRESELECT cases is spent executing the query (e1 time):

```
select
    tt_1_1__im0_.id as id0_, tt_1_1__im0_._multiplex as column1_0_, tt_1_1__im0_._errorFlag as column2_0_, tt_
1_1__im0_._originRowid as column3_0_, tt_1_1__im0_._datasourceRowid as column4_0_, tt_1_1__im0_._errorString a
s column5_0_, tt_1_1__im0_._peerRowid as column6_0_, tt_1_1__im0_._rowState as column7_0_, tt_1_1__im0_.f1 as
f8_0_, tt_1_1__im0_.f2 as f9_0_, tt2_1_1__i1_.id as id10_, tt2_1_1__i1_._multiplex as column11_0_, tt2_1_1__i1
_._errorFlag as column12_0_, tt2_1_1__i1_._originRowid as column13_0_, tt2_1_1__i1_._datasourceRowid as column
14_0_, tt2_1_1__i1_._errorString as column15_0_, tt2_1_1__i1_._peerRowid as column16_0_, tt2_1_1__i1_._rowStat
e as column17_0_, tt2_1_1__i1_.f1 as f18_0_, tt2_1_1__i1_.f2 as f19_0_
from
    tt1 tt_1_1__im0_
cross join
    tt2 tt2_1_1__i1_
where
    tt_1_1__im0_._multiplex = ? and tt2_1_1__i1_._multiplex = ?
order by
    tt_1_1__im0_._multiplex asc, tt_1_1__im0_.id asc, tt2_1_1__i1_.id asc
```

I have not profiled to see where the time is being spent. The query is not especially complex; however, the result set is very large. We are not setting an explicit maximum number of results in the query statement itself, so I guess H2 is computing them all? Here it would be interesting to know what happens with a database like PostgreSQL, where we apply a JDBC fetch size of 1,024 results at a time.

4) In this temp-table case, we are fetching full records. If the cache hit rate is very high AND the same records are represented in the result set many times, this is wasteful. I suspect we can cut down on the query execution time considerably if the query is a projection query (i.e., it brings back only the primary key), simply because the result set will contain far less data. However, the tradeoff there is that this approach will require an extra query for each record, to hydrate and cache the record for the first time. In this stylized test case, this could be avoided for cases where UNDO-able temp-tables are in use, because we know the records are already in the cache, from when they were created. But in real scenarios, this will not always be the case.

#4 - 08/23/2022 06:32 PM - Eric Faulhaber

- File 6695a_ecf_20220823.patch added

Eric Faulhaber wrote:

> I also instrumented FWD with the attached patch (based on 3821c/14190),

Forgot to attach the patch. This must not be committed.

#5 - 08/24/2022 05:33 AM - Alexandru Lungu

These are some very insightful results!

0.499 rate for PRESELECT NO-UNDO is exactly what I expected: only the records from the first table are hitting the cache. The second table has more records than the cache size, so the cache won't help tt2. In FOR NO-UNDO, the records from tt are retrieved rarely, therefore no cache hit makes sense for this scenario.

However, I think that the e3 difference is concerning as long as 13,428ms is just between no-fetch and cached-fetch of the first table records. Basically, 13,428ms are used just to do extra 8,997,000 cache hits. As a matter of fact, this time is way bigger than 7,321ms done by FOR NO-UNDO doing 9,003,000 cache misses :)

I agree that e1 is something to worry about. This case may have a simple solution (like you mentioned - fetch size). I will take a look into H2 to see if it allows such setting. My plan is to:

- run your patch against v2.1.210 H2 patched to see if there are consistent differences (especially for e1)
- do the same tests for PostgreSQL
- retrieve some times from 4GL for the test in #6695-3 (for reference)

#6 - 08/24/2022 06:47 AM - Greg Shah

> I also instrumented FWD with the attached patch (based on 3821c/14190),

> Forgot to attach the patch. This must not be committed.

If you use our JMX instrumentation instead of custom counters, it would be something safe to commit and it would be available at any time instead of only during times when the patch is applied.

#7 - 08/24/2022 09:28 AM - Alexandru Lungu

I locally added JMX beans for cache hits and misses and I get the same results (4.99 on PRESELECT NO-UNDO, 0.0 on @FOR NO-UNDO, etc.). Let me know if you want these committed. I also get very similar times (around 20sec for PRESELECT NO-UNDO, around 7 sec for FOR NO-UNDO, etc.)

I tested with the patched version of v2.1.210 H2 and no consistent difference was noticed. I also used this same test (with 3000 records in each table) on 4GL and I got the following:

4GL Temp-Table	PRESELECT	FOR
UNDO	e1: 9537 e2: 0 e3: 9641	e1: 0 e2: 0 e3: 15729
NO-UNDO	e1: 9728 e2: 0 e3: 9672	e1: 0 e2: 0 e3: 15817

This is really curious as our dynamic query outperforms the original 4GL dynamic query. Also, the PRESELECT UNDO testcase seems to work faster in FWD. However, the PRESELECT NO-UNDO seems to struggle. Even if it succeeds in opening the query faster, the iteration of the records is slower. I think this is the hot-spot we should prioritize. Note that these 4GL tests were run inside a VM.

Finally, it is encouraging that e1 is lower in FWD when in PRESELECT. This ensures us that it is safe to waste some time in the join SQL (mentioned in #6695-3) as long as we provide a faster mean of iterating through results (which we don't do yet).

Working now on the PostgreSQL tests.

#8 - 08/24/2022 09:41 AM - Greg Shah

> I locally added JMX beans for cache hits and misses and I get the same results (4.99 on PRESELECT NO-UNDO, 0.0 on @FOR NO-UNDO, etc.). Let me know if you want these committed.

Are you using the "standard" approach implemented in #4785-746 and lightly described in #1847?

Igor: Please add details to JMX Instrumentation to allow the rest of the team to easily follow our standard approach.

> Note that these 4GL tests were run inside a VM.

Make sure that the 4GL code is compiled before you run it. Otherwise the compilation cost in the 4GL may make that code appear slower than it really would be in production.

#9 - 08/24/2022 09:45 AM - Igor Skornyakov

Greg Shah wrote:

> Igor: Please add details to JMX Instrumentation to allow the rest of the team to easily follow our standard approach.

Sure. Is it OK f I will do it a little later? I'm in the middle of debugging a complex MariaDB UDF now and want to finish it first.
Thank you.

#10 - 08/24/2022 09:50 AM - Greg Shah

Yes, no problem.

#11 - 08/24/2022 09:52 AM - Alexandru Lungu

Greg Shah wrote:

> > I locally added JMX beans for cache hits and misses and I get the same results (4.99 on PRESELECT NO-UNDO, 0.0 on @FOR NO-UNDO, etc.). Let me know if you want these committed.
>
> Are you using the "standard" approach implemented in #4785-746 and lightly described in #1847?

I used the specification from Profiling, Adding Tracepoints. I added two com.goldencode.p2j.jmx.SimpleCounter named SessionCacheHits and SessionCacheMisses which are updated in SQLQuery. I use VisualVM. I enabled both to see the cache instrumentation for the FWD server.

#12 - 08/24/2022 10:24 AM - Alexandru Lungu

Greg Shah wrote:

> Make sure that the 4GL code is compiled before you run it. Otherwise the compilation cost in the 4GL may make that code appear slower than it really would be in production.

Just checked; I am using an environment (Developer Studio) where a source file is compiled each time it is changed. To ensure, I compiled the test and run it separately (as .r). I get the same measurements as in #6695-7.

#13 - 08/24/2022 02:50 PM - Eric Faulhaber

Alexandru Lungu wrote:

> I locally added JMX beans for cache hits and misses and I get the same results (4.99 on PRESELECT NO-UNDO, 0.0 on @FOR NO-UNDO, etc.). Let me know if you want these committed.

It depends on whether it is smarter than the throw-away implementation I did. Mine was only meant to get an answer for the very limited test case (only one query) and will not be useful in any more complex situation.

What we really would like to be able to track on a per-query level is:

- how much time was spent executing the query (from submission of the query up to the point of getting a result set back);
- how many rows were processed from the result set;
- how many session cache hits and misses there were for that result set;
- how much time was spent hydrating records from that result set.

If this can be implemented with JMX such that it has minimal overhead when active and zero overhead when inactive, that would be ideal. How close is your implementation to meeting these requirements?

We can gather some of this type of information generically using a standard profiler, though not exactly organized this way, and it is much more invasive to profile a running server with such a tool.

#14 - 08/24/2022 03:01 PM - Eric Faulhaber

Alexandru, I had missed your description of the tracing in #6695-11 when I posted my note.

If the hit/miss counts are tracked globally, that could be useful on a macro level (e.g., perhaps as a high level input to inform session cache configuration). It would be more useful still to see these on a per-query basis (along with the other performance aspects noted in my previous post).

#15 - 08/25/2022 07:45 AM - Igor Skornyakov

Greg Shah wrote:

> Yes, no problem.

Greg,
It looks that I need some time to recall the details. Maybe I will prepare the documentation at the weekend?
Thank you.

#16 - 08/25/2022 09:28 AM - Greg Shah

Igor Skornyakov wrote:

> Greg Shah wrote:
>
> > Yes, no problem.
>
> Greg,
> It looks that I need some time to recall the details. Maybe I will prepare the documentation at the weekend?
> Thank you.

Actually, it looks like the documentation already exists in Profiling. Please review that and add any missing details there. It is OK to do that on the weekend.

I will remove the other document from the Internals book.

#17 - 08/25/2022 09:36 AM - Igor Skornyakov

Greg Shah wrote:

> Actually, it looks like the documentation already exists in Profiling. Please review that and add any missing details there. It is OK to do that on the weekend.

Thank you! I had vague memories that I've prepared some documentation but I was not sure. I will review it on the weekend.

#18 - 08/25/2022 10:05 AM - Greg Shah

I had forgotten too. It was Alexandru's note in #6695-11 that reminded me. ;)

#19 - 08/28/2022 04:30 AM - Igor Skornyakov

Greg Shah wrote:

> Actually, it looks like the documentation already exists in Profiling. Please review that and add any missing details there. It is OK to do that on the weekend.

The documentation looks accurate. I've added a few words describing recent add-on for using 'specialized' MBeans.

#20 - 08/29/2022 05:40 AM - Alexandru Lungu

I added 3821c/rev. 14206 including a new MBean for query profiling: QueryProfiler. It stores a map from an SQL prepared query to a map of counters (cache hits, cache misses, result row count). The MBean isn't an overhead when it is disabled, but when enabled for the test #6695-3, it greatly slows down the application (maybe because there are a lot of database queries executed). This is not an issue when doing small tests for cache hits/misses, but it is irrelevant for hydrate time measurements. When the hydration is profiled, the execution time is 4/5 times bigger and ~80% of it is the hydration time. We can't rely on the accuracy of this time profiler. I implemented the hydrate profiling, but did not added to SQLQuery yet - thinking of a solution to get more accurate results.

Note that for #6695-3, I got the exact same cache hit/miss ratio (PRESELECT NO-UNDO : 8,997,000 / 9,003,000, FOR NO-UNDO : 0 / 18,000,000).

#21 - 08/29/2022 10:16 AM - Alexandru Lungu

- Related to Feature #6720: lazy hydration added

#22 - 09/06/2022 11:24 AM - Alexandru Lungu

- removed the comment -

It was meant to be posted in #6582

#23 - 09/16/2022 05:12 AM - Alexandru Lungu

- Assignee set to Radu Apetrii

- Status changed from New to WIP

#24 - 10/04/2022 05:26 AM - Radu Apetrii

After some analysis, I have identified a spot in PreselectQuery where things could be optimized. The problem was as follows: when the iteration through records took part, the function HydrateRecords would be called for each QueryComponent of the query. That means that no matter what the buffer for a specific table contained, the program would fetch from the Results the requested record and it would attach it to the buffer. In other words, if the buffer already contained the requested record from the previous fetch, the program would execute the fetch anyways, even though it is unnecessary.
The change that I made consists of a method that checks whether hydration is necessary or not. This method would look at the buffer's current record and compare it's ID with the one that will be fetched from Results. If they match, then there is no need for hydration, and if they don't, then hydration is necessary.
As far as testing goes, there are 2 notes to be made here:

- Previously, the e3 time for the Preselect NO-UNDO was somewhere in between 21 and 22 seconds on my computer, but now, with this change, it is reduced to somewhere in between 14 and 15 seconds. This method has the same cache rate as the one in the table (#6695-3), the only difference being that it applies to a different part of the program (not between ResultSet and Results, but between Results and the table's fetching).
- I have tested all of adaptive_scrolling, but with the keyword Preselect instead of For inside the Open query instruction and they all generate the same results as the original program (without the change).

Right now I'm doing some more testing to see that this change doesn't break something else and make sure that this is, in fact, an improvement.
I'm planning to commit these changes to 6582a.

#25 - 10/04/2022 10:55 AM - Eric Faulhaber

Radu, please help me understand this better (perhaps it will make more sense to me when I review the update).

My confusion is about how the hydration is being bypassed, based on what already is in the buffer. If a record is in a RecordBuffer instance, it also must be in the Session cache. That is, a record is added to the session cache when it is first hydrated, and we veto eviction of records from the cache as long as they remain in a buffer. The first thing SQLQuery.hydrateRecord does is read a primary key from the current row in the result set, and look for it in the session cache. If it is found (which it must be, if it currently exists in a RecordBuffer), we do not create/hydrate a new DMO.

It sounds like you have found something different. Can you help me understand why the mechanism I have described is not working in the case you have found? Again, maybe seeing the update itself will explain it.

#26 - 10/06/2022 05:23 AM - Radu Apetrii

There was a mistake in explanation on my part in the last post. In order to make this more clear, I will describe this process step by step.
The action of taking a Record from the ResultSet and putting it in the RecordBuffer requires two steps:
Step 1: Taking the Record from the ResultSet and storing it temporarily inside an Object array called data. This is where the Session cache comes in play, because the Record is simply given from the cache (if it exists there) instead of being unnecessarily hydrated. All of this happens inside the SQLQuery.hydrateRecord method, as you said.
Step 2: Placing the Record from the Object array called data inside the RecordBuffer. This happens inside the PreselectQuery.coreFetch method.

The problems were:

- In step 1, even though the caching did happen, retrieving the Record from the cache was still quite costly.
- In step 2, there is a place in which this process stops if the buffer already contains the required Record, but that happens pretty late and a lot of instructions are already executed.

The changes I have applied are as follows:

- In step 1: before retrieving the Record from the ResultSet (right before the call of SQLQuery.hydrateRecord), the program verifies if the RecordBuffer needs fetching or not. If it requires fetching, it then moves forward to SQLQuery.hydrateRecord where it checks the Session cache (as described before). If the RecordBuffer does not require fetching, the Object array called data stores null. This whole process is done because checking the necessity of fetching requires less time than looking inside the Session cache and retrieving the Record. Thus, this change adds a layer before checking the cache and does not replace that cache.
- In step 2: If the Object array called data has a null on a specific position, the program does not continue placing the Record inside the RecordBuffer, but instead it moves on to the next position because we know that fetching is not necessary for that buffer.

Short recap:

- Before: Step 1. Object[] data -> Session cache (return if Record is found) -> ResultSet (= hydration)
- After: Step 1. Object[] data -> check fetching (return if the buffer does not need fetching) -> Session cache (return if Record is found) -> ResultSet (= hydration)
- Before: Step 2. buffer -> Object data
- After: Step 2. buffer -> Object data (only when needed)

The thing that I want to point out here is that this change only applies to temporary no-undo tables for the moment.
If this proves to be a significant change I will try to apply it to different kinds of tables. In the meantime I will continue testing the correctness and performance of it.

#27 - 10/10/2022 07:41 AM - Radu Apetrii

- % Done changed from 0 to 30

I committed on 6582a, rev.14168. This change targets the improvement in performance regarding fetching for PreselectQueries on temporary no-undo tables.

#28 - 10/20/2022 09:10 AM - Radu Apetrii

There is already some work for #6720 which is close to this thread.
I am planning to move my changes from 6582a to 3821c as long as they are easy to integrate now.
I am setting up a large customer application right now to do a last check of my changes.

#29 - 10/31/2022 07:16 AM - Radu Apetrii

- File 6695.patch added

I have attached a patch that includes the changes regarding preselect queries on temporary no-undo tables. They were tested with 3821c, rev. 14326 on a large customer application and some other (relatively small) performance examples. The changes are committed to 6582a, rev. 14171.
Performance-wise, the test from #6695-3 runs in between 13 and 15 seconds (previously, the running time was between 21 and 22 seconds).

#30 - 10/31/2022 05:38 PM - Eric Faulhaber

Code review 6695.patch:

Interesting solution. Why is it only applicable for NO-UNDO temp-tables?

The patch does not include the LazyFetching interface, so it won't compile.

Many files are missing header entries; please add them.

Please use wildcards with import statements instead of importing individual classes, unless resolving a conflict between packages.

In the implementations of public Object[] get(LazyFetching fetching), we throw IllegalStateException with the message "This kind of Results does not support lazy fetching". It seems like we can reach these when fetching is not null and its isFullRecordOk method returns false. Can this be refactored to avoid that possibility?

PreselectQuery$PreselectLazyFetching:

- shouldFetch javadoc discusses full vs. partial fetching, but I don't see any distinction in the implementation. Is the distinction important here?
- Why are the comp parameters for shouldFetch and setOldId Integer instead of int? Are there any cases where null is a valid argument? The parameter is just the index of the query component, so I would think not.

SimpleResults: this is just a wrapped ArrayList of Record instances (or primary keys). The nature of this object is that the values already have been fetched. Isn't any attempt to lazy fetch them irrelevant for this type?

BaseRecord: Please removed the new getId() method and use primaryKey() instead.

#31 - 10/31/2022 06:14 PM - Eric Faulhaber

Radu, in my last post I started right into the code review comments, but I want to note that I appreciate your recognition and diagnosis of the problem and your creativity in coming up with a solution. Thank you!

#32 - 11/09/2022 12:47 PM - Radu Apetrii

- File 6695-2.patch added

I have taken into account what you said and I have made some changes that appear in this new patch. They were tested on 3821c, rev.14354.
Eric Faulhaber wrote:

> Interesting solution. Why is it only applicable for NO-UNDO temp-tables?

At first, when I thought about this solution I had in mind only NO-UNDO temp-tables (because that was the most worrying test regarding performance out of the 4 -> preselect with no-undo temp-tables, #6695-3). After that, the solution became more general, but I hadn't tested for the other tables (in the beginning).
Since your reply, I had done tests for UNDOABLE temp-tables and persistent tables and there were no errors. If you find any problems, please let me know.
I have done tests with adaptive_scrolling, large customer application, some personal tests and some performance tests.

> The patch does not include the LazyFetching interface, so it won't compile.

Sorry for that, I completely forgot to add the interface.

> Many files are missing header entries; please add them.

I added the headers. At first, I wasn't sure if they should also be included in the .patch or just in commits (my bad).

> Please use wildcards with import statements instead of importing individual classes, unless resolving a conflict between packages.

Added wildcards.

> In the implementations of public Object[] get(LazyFetching fetching), we throw IllegalStateException with the message "This kind of Results does not support lazy fetching". It seems like we can reach these when fetching is not null and its isFullRecordOk method returns false. Can this be refactored to avoid that possibility?

This was done intentionally. There is a message at the bottom where I'll try to explain this part.

- shouldFetch javadoc discusses full vs. partial fetching, but I don't see any distinction in the implementation. Is the distinction important here?

No, I think my explanation was not clear enough. The method shouldFetch checks if LazyFetching can be applied. If that is not the case, then the record should be fetched (either partially or fully, it doesn't matter). It is not important (for that method) how the record gets fetched, only if it needs fetching or not. In other words, it checks if the layer of LazyFetching should be bypassed or not.

- Why are the comp parameters for shouldFetch and setOldId Integer instead of int? Are there any cases where null is a valid argument? The parameter is just the index of the query component, so I would think not.

You are right. They should've been int, I have changed that.

> SimpleResults: this is just a wrapped ArrayList of Record instances (or primary keys). The nature of this object is that the values already have

been fetched. Isn't any attempt to lazy fetch them irrelevant for this type?

The answer is included in the message at the bottom.

BaseRecord: Please removed the new getId() method and use primaryKey() instead.

Done. I don't know how I've managed to miss the primaryKey() method.

This is an explanation of how things are supposed to work inside a get method with a LazyFetching parameter (kudos to Alex for helping me with this).
This is a template of a get method:

```
public Object[] get(LazyFetching fetching)
{
   if (fetching == null || fetching.isFullRecordOk())
   {
      //code for returning record
   }

   throw new IllegalStateException("Lazy Fetching is used, but the record is already hydrated");
}
```

There are 4 possible cases here:

- 2 of them contain fetching being null. In this case, LazyFetching was not intended, thus we can return the record without further implications.
- fetching != null and fetching.isFullRecordOk() is true means that we intend to use LazyFetching and we are perfectly fine with returning the full record. In other words, having full record is OK. This is going to be the case most of the times (almost all the times).
- fetching != null and fetching.isFullRecordOk() is false means that we intend to use LazyFetching but we are not OK with having the full record (for example: OPEN QUERY q FOR EACH tt FIELDS f1.).

For the last case, we return an exception as a safety net. There is an explanation for this behavior. For the moment, the plan is to notify the developer when such a case is being encountered (meaning that we work with the full record even though we might only need a subset of fields from it). After that, the developer might take this into consideration and implement the required behavior (working with subsets) or make the method isFullRecordOk return true, thus working with the full record (meaning that the exception acts, again, as a safety net). If the developer decides to implement that behaviour, the exception will be replaced by proper code that deals with the situation.
For SimpleResults I have decided to remove the check for LazyFetching because the need of providing database results as fast as possible is greater than the need of safety of a future bug with a very slim chance.
If you have any more questions or you think that some aspects could be explained better, let me know. Also, I hope you won't encounter any problems, but if you do, (again) let me know.

Files

6695a_ecf_20220823.patch	3.55 KB	08/23/2022	Eric Faulhaber
6695.patch	24.5 KB	10/31/2022	Radu Apetrii
6695-2.patch	33.3 KB	11/09/2022	Radu Apetrii