Database - Feature #6720

lazy hydration

08/29/2022 06:39 AM - Greg Shah

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Alexandru Lungu	**% Done:**	90%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

Related issues:

Related to Database - Feature #6695: Multi-table preselect query may underper...	**WIP**
Related to Database - Feature #2137: runtime support for FIELDS/EXCEPT record...	**New**
Related to Database - Bug #7185: H2 in-memory lazy hydration	**Test**

History

#1 - 08/29/2022 07:21 AM - Greg Shah

In query processing, it is my understanding that a substantial percentage of time is spent in hydration of returned rows from a result set. Some of the hydration process (SQLQuery.hydrateRecord()) could be improved slightly (e.g. pre-calculate some things, iterate over a pre-calculated array of "field descriptors" rather than iterating over the Map.entrySet() results...). Regardless of these small (but safe) improvements, the majority of the hydration time would still be needed.

The more costly part is the hydration itself. It seems to me that it is likely that the majority of hydration work that is done is not needed. In other words, for most buffer loads it is likely that only a small number of fields in a given row are actually accessed. The various buffer copy cases are exceptions but I think these will be a small percentage of the overall buffer loads in most use cases.

I think we should measure the following:

- how much query time is spent in hydration
- how often hydration is really needed

We've often discussed that it is likely that implementing the runtime parts of field list support (#2137) and calculating field lists where we can (#6721) will have some major performance benefit. Field lists would allow implementation of a partial hydration approach. I would expect that a field list would be rendered down to an array of "field descriptors" which would be used for the hydration loop as described above. By implementing partial hydration, we could get a large amount of the hydration time back for some number of statically calculable cases. This won't help us in the cases that cannot be calculated.

Lazy hydration is more of a runtime concept. The idea: only hydrate a given field when it is actually accessed. Doing this would require us to carefully map the lifecycle of the returned result set. Lazy hydration is possible so long as the result set still exists, but this won't be valid for cases where:

- the buffer scope is at an enclosing block from the block to which a query is associated (e.g. FOR EACH)
- the query is a "one and done" associated with something like a FIND, such that it doesn't outlive the FIND itself

It seems to me that the FOR EACH case (including the very costly multi-table cases) doesn't need to worry about the lifetime of the entire result set because only a single row can survive the scope of the FOR EACH block. Worst case, we could hydrate the last set of buffers as we exit the block and the rest of the result set can never be accessed again anyway. This seems like we could avoid a huge amount of hydration with this lazy approach.

Implementing laziness for FIND would require retaining the result set for the lifetime of the buffer.

Is it possible that the buffer could be scoped outside of the database transaction? If so, that would be another problem to resolve.

#2 - 08/29/2022 10:16 AM - Alexandru Lungu

- Related to Feature #6695: Multi-table preselect query may underperform due to repetitive fetching added

#3 - 09/02/2022 02:14 PM - Eric Faulhaber

If we are to lazily hydrate records, any column data which was not loaded in the initial hydration pass will have to be loaded while our result set cursor is still "on" that record in the result set. A JDBC ResultSet is not a random-access construct; all movement among records is relative to the cursor's current position. The typical idiom of reading a set of results is:

```
ResultSet rs = <execute query>;

while (rs.next())
{
   // access column data in the result set for the next result record,
   // using an individual method call for each column from which we
   // want to read a datum
}
```

Once the cursor leaves the current record, it is not likely we can get back to that result row efficiently. If certain parameters are passed with the query execution to enable non-forward movement through the result set, one can move backward and forward relative to the current cursor position, but the backend needs to support this, and it will not be fast to get to some random record.

We will need to check whether the column we are reading has changed in the DMO and know whether the record has been deleted, so that we are not loading stale data from the result set.

#4 - 09/05/2022 01:48 AM - Greg Shah

- Related to Feature #2137: runtime support for FIELDS/EXCEPT record phrase options added

#5 - 09/29/2022 03:52 PM - Greg Shah

> If we are to lazily hydrate records, any column data which was not loaded in the initial hydration pass will have to be loaded while our result set cursor is still "on" that record in the result set.

As I noted above, more many cases like FOR EACH, isn't this already the normal lifetime of the record in the buffer? Unless I am misunderstanding things, there are some number of cases where we already match the lifetime of the buffer to the time during which we would have access to the result set. Such cases could easily access just the fields needed, when they are needed. Presumeably, this is a subset for most cases. Only some rare code cases and things like BUFFER-COPY/BUFFER-COMPARE would access all fields.

> Once the cursor leaves the current record, it is not likely we can get back to that result row efficiently.

I'm not suggesting this. The idea is that for the cases where we must have access to a given record that is scoped longer than the result set, then we can copy that record only when we leave the scope of the result set. It should only affect a single record because only that one record referenced by the buffer can ever be accessed again. I'm thinking of a FOR EACH case as an example. Worst case, all records but one can be lazy.

#7 - 10/19/2022 09:42 AM - Greg Shah

Code posted in #5731-399 implements partial record hydration.

#8 - 10/19/2022 03:18 PM - Sergey Ivanovskiy

Greg, I think that #5731-399 doesn't implement lazy hydration. It just extends the usage of FWD Persistence API for external java applications in which "select" queries with incomplete list of fields are used.

#9 - 10/19/2022 06:24 PM - Greg Shah

Yes, that is what I calling partial record hydration.

#11 - 01/03/2023 03:48 AM - Eric Faulhaber

- Assignee set to Eric Faulhaber

- Status changed from New to WIP

#12 - 01/20/2023 03:33 AM - Eric Faulhaber

Ovidiu, Constantin, Alexandru, can you please help me understand this snippet of code in SQLQuery.hydrateRecordImpl?

```
         pk = resultSet.getLong(rsOffset);
         Record cachedRec = session.getCached(recordClass, pk);
         if (cachedRec != null && !cachedRec.checkState(DmoState.STALE))
         {
            SIMPLE_QUERY_PROFILER.updateCacheHits(resultSet, 1);
            // we found an unSTALE CACHED record, do not bother reading from result set
            return cachedRec;
         }

         if (count == 1)
         {
            SIMPLE_QUERY_PROFILER.updateCacheHits(resultSet, 1);
            cachedRec = session.get(recordClass, pk);

            return cachedRec;
         }
```

In the first conditional block, we check the session cache for a DMO with the primary key retrieved from the result set, and we return it if it is not null and not marked STALE. However, only the null check is really useful, because if a STALE record was found by Session.getCached, it would have been evicted during that call, and null would have been returned. Session.getCached will not return a STALE DMO.

So, if we reach the second conditional block, we must have had a session cache miss. Then we check if count is 1, meaning we only have a primary key and no other data in the result set. At this point, even though we've had a cache miss, the first thing we do in that block is register a session cache hit, with a call to SIMPLE_QUERY_PROFILER.updateCacheHits. Then, we check the session cache again, using the same parameters as before, so we must get the same result back: null. I suppose the return of the null would be appropriate here, since there's nothing we can do to hydrate the record, if we only have the primary key. However, the JMX call seems wrong, and the second cache check seems unnecessary.

Unless I've misunderstood something (please let me know if you think so), I think the correct logic would be:

```
         pk = resultSet.getLong(rsOffset);
         Record cachedRec = session.getCached(recordClass, pk);
         if (cachedRec != null)
         {
            SIMPLE_QUERY_PROFILER.updateCacheHits(resultSet, 1);

            // we found a cached record, do not bother reading from result set
            return cachedRec;
         }
         else if (count == 1)
         {
```

```
            SIMPLE_QUERY_PROFILER.updateCacheMisses(resultSet, 1);

            // we found no record in the session cache for the given primary key and
            // we have no data with which to hydrate a new DMO, so return null; let
            // the caller work it out
            return null;
        }
```

Please let me know if you see any problems with this. Thanks.

#13 - 01/20/2023 04:08 AM - Alexandru Lungu

Eric Faulhaber wrote:

> In the first conditional block, we check the session cache for a DMO with the primary key retrieved from the result set, and we return it if it is not null and not marked STALE. However, only the null check is really useful, because if a STALE record was found by Session.getCached, it would have been evicted during that call, and null would have been returned. Session.getCached will not return a STALE DMO.

I am not sure about the STALE state of a DMO when doing a cache check. Session.getCached evicts STALE records before returning, but only if versioning exists (only available for non-temporary DMO). Therefore, is it safe to say that we cache STALE records only from the temp database?

> So, if we reach the second conditional block, we must have had a session cache miss. Then we check if count is 1, meaning we only have a primary key and no other data in the result set. At this point, even though we've had a cache miss, the first thing we do in that block is register a session cache hit, with a call to SIMPLE_QUERY_PROFILER.updateCacheHits. Then, we check the session cache again, using the same parameters as before, so we must get the same result back: null. I suppose the return of the null would be appropriate here, since there's nothing we can do to hydrate the record, if we only have the primary key. However, the JMX call seems wrong, and the second cache check seems unnecessary.

The second call is a session.get, not the same session.getCached. This means that, we actually want a second SQL query to retrieve the rest of the DMO (to hydrate). Also, I don't think that the "caller" will work it out if we return null; I guess it will consider that the primary key doesn't belong to an existing record anymore. I can agree however that the JMX is wrong; it is a cache miss (I was mislead by the variable name cachedRec).

#14 - 01/20/2023 04:55 AM - Eric Faulhaber

Thanks for the second set of eyes, Alexandru! I was staring at this too long and completely missed that the second call was get and not getCached.

> I am not sure about the STALE state of a DMO when doing a cache check. Session.getCached evicts STALE records before returning, but only if versioning exists (only available for non-temporary DMO). Therefore, is it safe to say that we cache STALE records only from the temp database?

No, we only ever mark/detect a record is stale when versioning is enabled (as you noted, only for persistent DMOs). The javadoc for DmoState.STALE is misleading; staleness has to do with a context updating its version/instance of a DMO while that DMO simultaneously is being used in a different context (as a different instance representing the same record). Any change made in another session (which requires an exclusive lock) will make all other copies in other sessions stale. This is not possible for temp-table records, which are all private/local.

#15 - 03/13/2023 07:38 AM - Alexandru Lungu

- Related to Bug #7185: H2 in-memory lazy hydration added

#16 - 05/25/2023 11:03 AM - Eric Faulhaber

Constantin: in SQLQuery.hydrateRecordImpl (line 834 in trunk rev 14585), we are checking if the current result set row contains only a single field/column. If so, it is presumed to be the primary key, and we are either getting a cached DMO (if there is one in the cache with that PK), or allowing Session.get to instantiate and fully hydrate a DMO for that PK:

```
          if (count == 1)
          {
             SIMPLE_QUERY_PROFILER.updateCacheHits(resultSet, 1);
             cachedRec = session.get(recordClass, pk);

             return cachedRec;
          }
```

If count is 1, doesn't that mean we have a projection (i.e., primary key only) query? Something like (in FQL) this:

```
select foo.recid from FooImpl__ [where ...]
```

In that case, we should only be getting back the recid, not a fully hydrated record. Are we actually hitting this case in SQLQuery.hydrateRecordImpl for projection queries, or do they never get this far (i.e., the primary key is returned earlier and hydrateRecordImpl is never reached)? Maybe I'm misunderstanding what type of query gets us here?

For context, this specific case (i.e., to handle the single field case with Session.get) was added in rev 11347.1.137:

```
    revno: 11347.1.137
    author: Constantin Asofiei <ca@goldencode.com>
    committer: Ovidiu Maxiniuc <om@goldencode.com>
    branch nick: 4011a
    timestamp: Thu 2020-06-11 01:19:25 +0300
    message:
      Fixed SUBSELECT parsing and Fql2Sql usage.  Other misc fixes.
    modified:
      src/com/goldencode/p2j/persist/FieldReference.java
      src/com/goldencode/p2j/persist/QueryComponent.java
      src/com/goldencode/p2j/persist/orm/FqlToSqlConverter.java
      src/com/goldencode/p2j/persist/orm/SQLQuery.java
```

```
      src/com/goldencode/p2j/persist/orm/fql.g
```

It was a while ago, and went in with a huge update ultimately, but any recollections about the purpose of this change would be helpful.

#17 - 05/25/2023 11:36 AM - Constantin Asofiei

Eric, I don't recall exactly which one, but this case of having a PK in hydrateImpl is originating from a kind of AbstractQuery - there was a case where this query was retrieving only the record PK. One of my status reports around that time mentioned this:

> (fixed some issues, blocked with a ScrollableResults.get where the caller expects to get a full DMO).

See this javadoc in ScrollingResults:

```
   /** Scrollable result set;  primary keys or {@code Record}s */
   private final ScrollableResults<Object> results;
```

So there is cases when a query computes its results with only PKs instead of full records. I can track down a specific case, if is really needed.

#18 - 05/25/2023 11:57 AM - Eric Faulhaber

OK, thank you. I've been reworking that method a lot and this didn't look right to me. I just wanted to know that there was a real use case behind it. I will retain it.

#19 - 05/25/2023 02:47 PM - Ovidiu Maxiniuc

Eric,
Maybe a bit of history here helps. The code from SQLQuery.hydrateRecord() (now SQLQuery.hydrateRecordImpl()) evolved a bit over time at the same steps with RowStructure.

Initially, there was no RowStructure, just its dmoClass because that was enough for basic generated queries to instantiate and populate the fields of the record with data from the columns. The method was keeping track of the number of properties processed in case of row composed of multiple tables joined.

Then the RowStructure was added and it was keeping only the number of properties. There were two cases for automatically generated queries: either the full record was requested or only the PKs. In former case, the PK was, by convention, the first property in the result set and the method is able to fully rebuild/hydrate the original record. In the later case the data is evidently the PK and the algorithm goes as you noted above: first look for the record in cache then fetch it independently. Actually, SQLQuery.list(Session, List<RowStructure>) will do a more heuristic work. Analysing the number of expecting records in the row it will decide whether a row in the ResultSet represent a tuple of PKs or the expanded records. There are some javadocs with a bit of explanations.

Recently, the fields was added to RowStructure in order to support EXCEPT/FIELDS 4GL options. The FqlToSqlConverter will create the list of fields

requested and populate the RowStructure accordingly. When the result is received, the hydrateRecord() will use this list for assigning only the restricted fields or use the full property map in its absence.

The current implementation relies heavily on initial conventions, primarily to minimize the data structure involved. So, if the count is one the value is PK. If you ant only one field, now the PK is mandatory (to identify the right recoed) so the count will be at least two. If needed, the RowStructure can be enhanced to provide additional information.

However, in case of hand-written SQL queries executed via the new persistence API the hydration is not possible if the caller does not pass the right RowStructure.

#20 - 09/01/2023 02:55 AM - Eric Faulhaber

- Assignee changed from Eric Faulhaber to Alexandru Lungu

#21 - 09/08/2023 08:43 AM - Alexandru Lungu

I've reread this issue and refreshed my mind. There are some key observations before moving on:

- Eric, 6720b exists (as you said), but didn't got the chance to check-out and review. Please pardon if my following points may contradict your current implementation.
- In #7045, the FOR-EACH queries are now parameters of forEach and forBlock. That means that the query is scoped to such block and FWD has full-control over its iteration. I think this helps a lot! We can basically control the scoping of every FOR-EACH query.
- This means that OPEN QUERY is the last issue here. However, I don't think we have a straight-forward solution for this - such queries can lose its records "in the wild". I am still to check Eric's solution, maybe there is a handling of such thing there.
- I found tons of examples where static inference of FIELDS clause would make lazy hydration less appealing. I thought we have a related task, but I may be wrong. This is about conversion time identifying the FIELDS clause for a query. I agree static approach is only sound, but no complete - however, we can buy some time here from the effective query time (less fields selected, serialized and returned).

#22 - 09/08/2023 09:33 AM - Greg Shah

> I found tons of examples where static inference of FIELDS clause would make lazy hydration less appealing.

My original concept was that this was not needed. The idea was that we only hydrated when a field was actually accessed. That way we only ever do what we know we need to hydrate.

Eric's prototype (which he still intends to document) intentionally over-hydrates to match the existing design of FWD which assumes in many places that hydration has happened. I would hope that we would cut these back significantly to really minimize what has to be hydrated. Perhaps we can find different ways to solve the same problems or we can limit the extra hydration to a subset of cases that really need it. Anyway, I think that is the primary challenge of making this idea work.

#23 - 09/27/2023 03:04 AM - Eric Faulhaber

This entry is intended to document the prototype implementation of lazy hydration in 6720b/14632-14634: what is implemented, what is planned, concerns... The branch needs a rebase to latest trunk, which I am working through currently.

Prototype Design

As discussed in previous notes, the idea is to fetch records from the database normally. However, instead of hydrating an entire DMO immediately before application logic uses it, we set up enough infrastructure to defer hydration of the properties of the DMO until they actually are accessed. **The prototype is unfinished at this time and relies on some classes which are only partially implemented.** It works only for the most basic test cases (simple FINDs and FOR EACH loops).

When Lazy Hydration is Applied

Only certain data access patterns support lazy hydration. Persistence APIs which gather and return a list of DMOs before using them cannot be lazily hydrated. Those which hydrate one record at a time, then use it with application logic, such as a single record retrieval or a scrollable result set, can support the concept of lazy hydration. We should strive to migrate cases which use a list pattern to use a scrolling pattern instead, where possible.

Currently, the database dialect determines whether lazy hydration is supported for a certain database type at all (see Dialect.useLazyHydration()).

If both the dialect and the access pattern supports lazy hydration, it is applied.

Implementation

Once a query is executed and a result set has been returned, and if lazy hydration is supported by the current dialect and the data access pattern, the JDBC result set is wrapped by an instance of LazyResultSet. This result set delegates every method call to the original result set. It also maintains an integer token which is incremented every time a method is invoked which would change the backing result set's state in such a way that would invalidate reading data from the current row. This includes moving the current row, closing the result set, etc.

The LazyResultSet is created by the following methods and is passed on, eventually, to SQLQuery.hydrateRecordImpl:

- ScrollableResults.get()
- ScrollableResults.get(DmoMeta, String, String[])
- SQLQuery.uniqueResult(Session, RowStructure)

In SQLQuery.hydrateRecordImpl, if a LazyResultSet has been passed in as the result set parameter, instead of fully hydrating the DMOs as we do currently, a Hydrator object is created by the LazyResultSet for each BaseRecord (i.e., DMO) created (or retrieved from the Session cache) for the row on which the result set currently is positioned. The Hydrator object receives and stores a copy of the LazyResultSet token, created at that moment. As long as the Hydrator token and the LazyResultSet token match, that Hydrator instance remains valid. Once the LazyResultSet token is incremented by repositioning, closing, etc. the backing result set, that Hydrator instance becomes invalid.

Hydrator stores a weak reference to the LazyResultSet. It is attached to the BaseRecord object, so that it later can be used to read data from the result set (assuming it is still valid at that time). When lazy hydration is active, the only datum that is read from the result set and stored in a DMO in SQLQuery.hydrateRecordImpl is the primary key of the current record. If the DMO already exists in the Session cache, it is retrieved from there instead.

All code paths to access a property must be re-routed to now go through the BaseRecord.getDatum method, where the actual, lazy hydration logic is implemented.

When application logic accesses (or sets) a DMO property and a Hydrator is present for that BaseRecord object, a bit set is consulted to determine whether that property is "live" (BaseRecord.liveProps). If so, the existing value of the property is returned from the BaseRecord.data array. If not, the Hydrator is tested for validity. If it is valid, the missing datum is read from the hydrator's LazyResultSet. It is stored in the data array, and liveProps is updated to set the corresponding bit. A similar operation must occur when setting a property (it must be read from the result set if not yet "live", so that the related ChangeSet object contains the correct baseline data).

In the event the hydrator is determined to be invalid (via a token mismatch with its LazyResultSet), it can no longer be used to retrieve a property from the backing result set and it is discarded (detached from the BaseRecord). In this case a LazyHydrationException is thrown. This is an unchecked exception. **A handler for this exception is not currently implemented**, but the idea was to catch this exception either in a common data retrieval method within the Record class, or perhaps further up the stack. The handler for this exception would force a re-fetch of the DMO's data by primary key from the database and perform a full hydration. However, the more I think about this use of an exception, the less I like it, both from a performance and an architecture standpoint. TODO: can we dispense with LazyHydrationException and handle the re-fetch within the orm package?

My initial plan was to forego further lazy hydration for this re-fetch and perform a full hydration in this scenario, since any additional round trip to the database must surely overshadow the performance benefit of lazy hydration, and we certainly would not want to make multiple such trips. Note that we need to ensure that use cases where a DMO snapshot or copy operation would cause such a re-fetch to occur do not represent common code paths. Currently, I think they do, as we take snapshots of DMOs at multiple places.

Unfinished Work

- Many LazyResultSet methods are not yet implemented. Those critical for initial testing of the prototype were implemented, but many methods are simply stubs. The good news is that these are quite simple, in that they merely delegate to the wrapped result set object, possibly incrementing the token if the state is changed for the current row.
- Record needs its data access methods reworked (only one or a few data types currently are handled) to use BaseRecord.getDatum instead of directly accessing the BaseRecord.data array. In fact, all logic which currently accesses the BaseRecord.data array directly needs to be

refactored to go through BaseRecord.getDatum. Direct access to the data array will corrupt lazy hydration.
- LazyHydrationException needs to be handled or replaced with a cleaner/better/faster idea. The DMO re-fetch needs to be fully figured out and implemented.
- Logic which currently snapshots/copies all DMO data (which happens, for instance, as each record is unloaded from a buffer) needs to be re-evaluated and refactored to deal with this requirement differently. If not, we are still fully hydrating every DMO and taking even more of a performance hit because this is being done with an extra database round trip.
- More advanced testing, once the prototype is further along, to confirm that there is a measurable performance benefit from lazy hydration in real use, and with different database dialects.

Other Concerns

- We currently support to a large degree the 4GL FIELDS/EXCEPT options (implemented previously). This is in some ways similar to lazy hydration, but I want to keep these implementations separate. The underlying assumption of lazy hydration is that the 4GL application logic does not specify which fields are needed (or excluded) for a particular query, so we only hydrate those which are needed, in a just-in-time fashion. When a FIELDS/EXCEPT option is used, we assume that the 4GL developer explicitly intended the specified fields to be included/excluded. In fact, OE reports errors if a field is used that is not part of the developer's specification, it does not fetch those fields as lazy hydration would.
- How should DMOs already cached in the Session be handled in the context of lazy hydration? How should lazy hydration and session caching be handled generally? How useful is a partially hydrated DMO fetched from the cache sometime later (when the Hydrator almost certainly is invalid)? Is it defeating the purpose of caching if we very likely have to make another round trip to the database to re-fetch missing data?
- We need to ensure we are not introducing memory leaks with the additional Hydrator objects and any possible strong references to result set objects that should be cleaned up.
- My initial decisions regarding which dialects should support lazy hydration are over-simplified:
  - Persistent databases are assumed to benefit from it, though we have no confirmation of this, except for PostgreSQL (which test did not consider the cost of re-fetching).
  - It is assumed that H2 should not use (this form of) lazy hydration, because we only support in-memory use of H2 for production environments. However, this was based on the assumption that in-memory access would already be faster, but this was not confirmed with testing.

#24 - 10/02/2023 12:53 PM - Eric Faulhaber

Task branch 6720b has been rebased to trunk rev 14762. New 6720b revision is 14766.

Alexandru, please review the code and the notes above and post any questions/comments you may have.

#25 - 10/11/2023 03:22 AM - Alexandru Lungu

Eric, I prepared some questions here last week, but it seems I missed submitting them. I was kind of waiting for feedback without questions :/

When Lazy Hydration is Applied

This is my first concern so I will start with this. From the initial discussion, I thought this will only apply to very specific use-cases, rather than general cases. It seems that you rule out only scenarios where list is used, which makes sense. However, I was thinking of allowing this only for FOR EACH, FOR BLOCK, REPEAT kind of constructs, which have a very well defined scope. From your comment, I see you are planning to extend this to OPEN QUERY use-cases as well. This concerns me, as we can easily lose the DMOs in the wild with OPEN QUERY. For FOR-kind blocks, we will always do next until the last record. For OPEN QUERY, we can reposition at arbitrary times, so **only the** buffer used for the query will be hydrated properly at all times. All DMOs that were retrieved using the OPEN QUERY, but now reside in other buffers, will be invalidated, if they access unhydrated fields.

Your current approach is "bottom-up", in the sense that we will lazy hydrate almost everything and by performance tests will eventually reduce some cases (**or maybe not**). I was rather thinking of a "top-down" approach, cherry-picking some use-cases and eventually extending the solution to other constructs.

Implementation

Really good approach with the token versioning. I think this is light-weight as long as we carefully care for the DMOs and Result-Sets, not to cause memory leaks. I have no concerns here. Initially I thought of a listener approach where DMOs are eagerly invalidated if the lazy result-set moves, but this will require a listener list that **may** easily leak.

Unfinished Work

- I've seen performance decrease before when using getter instead of direct data access, but so be it - we can't handle lazy hydration without explicit getter calls.
- "Refetching" is the scariest word around this task. Without proper testing, it is hard to understand how often will these "lazy hydration" problems occur.

Other Concerns

- Session cache is a very delicate matter here. We risk using a stale information with high chances of refetching **when** we actually have the full-data, but unhydrated yet. This is something like (20% changes of going very fast as we don't refetch + 80% changes of going slow as we refetch **vs** 100% of going fast as we hydrate). Maybe we can decide based on the DMO (if it has >50% of its field hydrated, use it from cache, otherwise no).
- For FWD-H2 (in memory), we have some implementation that is quite good, but is slow due to the getter/setter things. Anyway, I would go ahead only with persistent dialects and let FWD-H2 in-memory to be implemented separately. When we have direct-access, we can even have a more special Hydrator.

With this being said, I am mostly good with the approach. The only concern is related to the use-cases we actually use this for and the session cache relevance.

#26 - 10/23/2023 11:16 AM - Alexandru Lungu

- % Done changed from 0 to 30

Committed 6720b/rev. 14767:

- ported some changes from 7185a designed specially for private BaseRecord.data access using getters.
- finished the implementation of LazyResultSet replacing the stubs.

I was wondering if we can approach something similar with Cursor. I mean, maybe we can replace the "token" architecture with a "position" one. If we know that the result-set sits on a specific position that matches the position of the DMO in that result-set, then we can hydrate. This covers the cases where we do something like FIRST - NEXT - FIRST (instead of token 3, we can have position 0 and still hydrate properly). We can do this switch after we have a fully working solution, as the change shall be slim.

#27 - 10/23/2023 04:09 PM - Eric Faulhaber

I like this idea.

#28 - 11/02/2023 11:33 AM - Alexandru Lungu

- % Done changed from 30 to 50

Committed 6720b/rev. 14768:

- **FIX** Made toString ignore the hydrator and simply show what is in the current data array.
- **FIX** Honored multiplex for TempRecord (+1 when computing the prop offset inside the result-set). Also, the multiplex should have been aggressively hydrated (just like the PK).
- **FEATURE** Added a weak reference to a loader inside the hydrator. This way, the BaseRecord can request the hydrator to do a refresh based on that loader. If the loader doesn't exist anymore (because the session closed), the LazyHydrationException will be raised. I wonder if we need to catch this, open a new session and reload - does it make sense to use a DMO when the session is closed or in another session than the one that created it (?).
- **ISSUE** There is a (big) issue with datetimetz data type. It is stored on 2 columns inside the result set. Thus, we can't randomly access fields from the result-set solely on they property offset, as it may mismatch
  - **I applied lazy hydration only to tables where we know that the SQL number of columns matches the number of properties**. I designed DmoMeta.sqlFieldCount for this exact purpose some time ago. Planning to use it now.
- **ISSUE** Another (big) concern is that the underlying result-set may close without notifying LazyResultSet (once with the statement). For ScrollableResults, we control when we close the statement, so we can close the result-set before hand just to have it going through LazyResultSet. For uniqueResult, I removed the lazy hydration, as the statement is closed right after (closing the result-set under the hood). **Any idea here to make it better?** For ScrollableResults things are a bit better, because usually such results have larger scopes.
- **ISSUE** Partial hydration and lazy hydration is not really getting along. If we have a DMO which was generated after a partial query, the hydrator will be able to hydrate only the requested fields (at that time)
  - We either store the rowstructure in the hydrator so we know before-hand if we attempt to lazy hydrate a field that exists or not in the underlying result-set. However, this is a bit tricky, because we also need to identify its position in the partial result-set solely based on its property index.
  - We lazily hydrate only when we are facing full-records. **I chose this variant until we find a way to leverage partial hydration**. The switch to the other approach is a TODO - we still need to investigate if we can benefit from having lazily hydrated partial records.

Good news: I have a customer application POC, that I use for profiling, working with the current 6720b. There is still work to do to address the remaining TODOs, but I want to extract some baseline statistics on this new technique we use now.

#29 - 11/02/2023 12:11 PM - Greg Shah

> I applied lazy hydration only to tables where we know that the SQL number of columns matches the number of properties. I designed DmoMeta.sqlFieldCount for this exact purpose some time ago. Planning to use it now.

Does that also exclude extent fields when in expanded mode? It seems like this "offset" calculation can be done once at conversion time and then

used always. In other words, can't we remember this offset and use it instead of a simple index?

#30 - 11/03/2023 07:42 AM - Alexandru Lungu

Greg Shah wrote:

> I applied lazy hydration only to tables where we know that the SQL number of columns matches the number of properties. I designed DmoMeta.sqlFieldCount for this exact purpose some time ago. Planning to use it now.

> Does that also exclude extent fields when in expanded mode? It seems like this "offset" calculation can be done once at conversion time and then used always. In other words, can't we remember this offset and use it instead of a simple index?

I need to reanalyze the extent field. We can do some kind of mapping (FWD offset to SQL offset) to save us from ruling out potential lazy hydration candidates.

I've done some tests with 6720b/14768 on a customer application POC:

JMX	Start-up	Testing run
VALID TOKEN	426	45942
INVALID TOKEN	0	1142
RECORD LAZILY HYDRATED	266	15525
RECORD FULLY HYDRATED	186	803
LAZY RESULT SETS CREATED	759	17593
NON-LAZY RESULT SETS CREATED	262	3562
SUCCESSFUL REFRESH	0	1142
FAILED REFRESH	0	0

Some conclusions:

- the hot-spot of the investigation is how many times we need a second round trip to the database to load due to token mismatch: ~**2.4%** of cases we need to load the record again. All of these cases are successfully refreshed.
- there are no failed refreshes.
- indeed, some result sets are not lazily created due to the mismatch between FWD offset and SQL offset: ~**16.8%**. This is big enough to require fixing now, but small enough to be able to see some performance differences without fixing.
- note that there are less records than queries. This is because, I tracked only records that weren't cached / were cached but are lazily rehydrated. Only ~**5%** are not subject to lazy hydration / re-hydration, because the scrollable results was not lazy **or** the query was partial.

I will do a profiling test now, to have an intermediate status and set some expectancy.

#31 - 11/03/2023 10:54 AM - Alexandru Lungu

Committed 6720b/rev. 14769:

- Added JMX to do some tracking for lazy hydration. Hopefully they aren't slowing the process too much.

I've done some profiling with 6720b/14769 and got exactly -2% improvement.

#32 - 11/06/2023 10:15 AM - Alexandru Lungu

Optimization 1: Refresh only fields that weren't hydrated already
Optimization 2: Discard the hydrator if we hydrated everything due to successive calls to get/set

Comitted 6720b/14770

- This will improve the refreshing time, aiming only for the fields we didn't hydrate already
- This will decrease the number of useless rehydration.
- A negigable overhead is added in finding out if all fields are live.

Unfortunately, the changes didn't prove to be that drastic, improving insignificantly (-0.1%). This is mostly because of the fact that refreshes are happening very rarely on the persistent database + the unhydrated fields are usually >90% of the total number of fields. Bad lead :/

Avoid lazily hydrating from H2

I omitted the start-up because it was volatile (depending on the appserver start-up time).

JMX	Testing run
VALID TOKEN	4097
INVALID TOKEN	44
RECORD LAZILY HYDRATED	9855
RECORD FULLY HYDRATED	4687
LAZY RESULT SETS CREATED	7915
NON-LAZY RESULT SETS CREATED	13240
SUCCESSFUL REFRESH	44
FAILED REFRESH	0
+ RECORD LAZILY HYDRATED, NOT REHYDRATED	1277

The last statistic represents how many times we lazily hydrate, but excluding the times we "re-hydrated" cached items (replacing the old hydrator with a new one)

Some conclusions:

- There are way less lazy hydration attempts on persistent database (~4.1k lazily hydrated fields)
- Suprisingly, there are still >60% records lazily hydrated / rehydrated. This means that each hydrator is used to hydrate 0.5 fields on average (comparing to the previous 3 fields on avg). **This is something that can be used in our further solution research**.
- As expected, there are <30% lazy result-sets. Most of the result-sets that were lazy before were on the H2 database.
- Again suprisingly, 3% of the refresh attempts were for the persistent database. All other ~1k loads were for H2 database.
- The new statistic (RECORD LAZILY HYDRATED WITHOUT REHYDRATE) shows that only 12% of the hydrators are for new DMO, ther other 88% are used to rehydrate (replace an old hydrator).

I will attempt a profiling round with lazy hydration only for persistent database. I am curious if the whole process of managing lazy hydration on the H2 database was an overhead all along. I also want to stress out that only 8% of the lazy field access attempts were on the persistent database.

Unique result

Further, I have some concepts of supporting lazy-results for uniqueResult:

Scope result-sets to DMO lifecycle

- Attach the result-set to the hydrator just like we do now
- When the DMO is evicted from the session cache, it will also close its underlying result-set

This may be dangerous for UNDO tables which can bypass the dmo cache limit in Session. This will guarantee that no refresh will happen. However,

this may be a performance concern as we need to reparse statements of result-sets that were not closed yet.

Queue the eviction of result-sets

- Don't close the result-set and let it generate hydrators when requested.
- Add these result-sets in a queue (size 1024 for a start)
- Discard (i.e. close) the result-sets when evicted from the queue.
- The same lifecycle is used for the statements.

This way, we keep the result-sets opened to support lazy hydration, but evict them periodically to avoid memory leaks. At worse, we end up with 1024 result-sets opened, but their SQL statements are still unusable.

Cache result-sets

- This is similar to the previous attempt, but use a LRUCache
- The key is the SQL used when computing the statement
- Each time we run uniqueResult, we check if we can close an in-use statement (so that we reuse it). This will close the underlying result-set to close and force refreshing

This is an optimized way of the previous idea, but we add the LRUCache overhead. Further, we either limit the size of this cache **or** let it expand to match the lifecycle of the DMO

Approach

If you have any feedback, it is much appreciated. I will attempt some of the ideas above next.

#33 - 11/07/2023 10:03 AM - Alexandru Lungu

Avoid lazily hydrating from H2

Tested and had only -0.8% time improvement comparing to baseline. This means that the _temp database benefits from lazy hydration (curiously, even more than the persistent database). This makes sense from the POV of the statistics in #6720-32, but are quite odd as _temp is not quite slow due to hydration (de-serializing data from an external server). Anyway, the only fact that we don't have to iterate the full _temp record is now a performance bonus.

Conclusion here: we should also keep lazy hydration for H2.

Unique result

The first solution of scoping the lifetime of the result-set to the DMO is not quite logical: the session should have notified the dmo, that notified the hydrator that notified the lazy result set etc. This whole chain is quite reversed comparing to the dependency logic we have. Thus, I dropped Scope result-sets to DMO lifecycle.

New analysis after implementing Queue the eviction of result-sets is below:

JMX	Testing run
VALID TOKEN	49804
INVALID TOKEN	1222
RECORD LAZILY HYDRATED	16604

RECORD FULLY HYDRATED	694
LAZY RESULT SETS CREATED FROM SCROLLABLE	17593
NON-LAZY RESULT SETS CREATED FROM SCROLLABLE	3562
LAZY UNIQUE CREATED	8619
NON-LAZY UNIQUE CREATED	2375
SUCCESSFUL REFRESH	1222
FAILED REFRESH	0
RECORD LAZILY HYDRATED, NOT REHYDRATED	5792

The clearing queue has the size of 2048 in my tests:

- there are 8.6k new lazy result-sets; 2.3k others are not lazy due to field order mismatch (to be handled)
- the number of valid tokens increased with ~4k and invalids with 80. This is reasonable increase. I expected **way** more valid tokens to pop-out, but I think less is better, right? (less fields being hydrated)
- the number of records lazily hydrated increased with 1k and fully hydrated decreased with 100 (most probably from the previous optimizations). This is not promising :/ I suspect that out of the 8k new unique queries, only 1k end up being hydrated and not found in the cache **or** found in the cache but were rehydrated.

I attempted some profile, but unfortunately 2048 cache size determined tons of "reparsing warnings". All of these signaled a downgrade of almost +5% from baseline. I am planning to rework it to actually use the Cache result-sets variant where we ensure that we don't reparse statements. Before doing that, I want to recheck this approach with a very small queue (size 16 eventually).

Fix datetime-tz and extent cases

Next on my list is to fix the the datetimze-tz and extent fields that may cause offset mismatches. AFAIK, expanded extent fields are not a problem, as the DMO is actually considering each expanded field as a different index. However, I need to do some more testing in this area. Anyway, by fixing this, we can push almost 20% queries into lazy hydration (for my test cases).

#34 - 11/07/2023 12:17 PM - Ovidiu Maxiniuc

Alexandru Lungu wrote:

> **Fix datetime-tz and extent cases**
> Next on my list is to fix the the datetimze-tz and extent fields that may cause offset mismatches. AFAIK, expanded extent fields are not a problem, as the DMO is actually considering each expanded field as a different index. However, I need to do some more testing in this area. Anyway, by fixing this, we can push almost 20% queries into lazy hydration (for my test cases).

I am not sure I understand why dtz are a problem for (partial|lazy) hydration. The name/type is available for each fetched column even before the request (select) being sent to DBMS and it should be also available in RowStructure companion object.

Talking about RowStructure. I think **we can improve performance** here by removing the fields Map<String, Property> and replacing it with a simpler and faster BitSet, if we impose that columns in a request are ordered by their property id. As you might already guessed by now, the BitSet would contain the indices of the requested properties. The properties will be extracted directly from the RecordMeta.getPropertyMeta(false) array.

#35 - 11/07/2023 01:02 PM - Alexandru Lungu

> I am not sure I understand why dtz are a problem for (partial||lazy) hydration. The name/type is available for each fetched column even before the request (select) being sent to DBMS and it should be also available in RowStructure companion object.

Well, it is not a **big** problem. It is just that it wasn't trivial to have it set for these kind of columns, so I skipped the tables using such fields just to reach some performance analysis / analytics asap. TL;DR, the datetime-tz fields have two SQL fields in the underlying result-set. Thus, all properties after a datetime-tz property (made out of 2 columns) will have their offset not in line with the FWD property order (e.g. char field no. 15 is on the 16th column in the result-set, because 5th and 6th columns were representing the same datetime-tz property).

Mind that in RowStructure we have an empty fields if we want **all** fields (which is the common case in fact).

> Talking about RowStructure. I think we can improve performance here by removing the fields Map<String, Property> and replacing it with a simpler and faster BitSet, if we impose that columns in a request are ordered by their property id. As you might already guessed by now, the BitSet would contain the indices of the requested properties. The properties will be extracted directly from the RecordMeta.getPropertyMeta(false) array.

You are right, but this brings to my second point. If we partially hydrate (request only some fields from the DB), then I don't have a solution yet to integrate it with lazy hydration (extract the data from the result-set later on). Again, this is not a **big** problem - it was my intention to rule it out so that I can get some results asap.

For both problems above, we need a mapping between the FWD property position in data and SQL column position in the DB tables.

#36 - 11/07/2023 01:59 PM - Ovidiu Maxiniuc

The hydration works now by iterating the columns of the ResultSet. The DataHandler s from com.goldencode.p2j.persist.orm.types automatically advance the necessary number of columns (for the moment all 1 with the exception of DatetimetzType) and return the number (methods readProperty() and setParameter()) of places processed. There is propertySize() property which can return the same value for you, if the previous ones are not involved.

#37 - 11/08/2023 04:00 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> The hydration works now by iterating the columns of the ResultSet. The DataHandler s from com.goldencode.p2j.persist.orm.types automatically advance the necessary number of columns (for the moment all 1 with the exception of DatetimetzType) and return the number (methods readProperty() and setParameter()) of places processed. There is propertySize() property which can return the same value for you, if the previous ones are not involved.

Indeed! The "difficulty" here was that we can't compute the proper offset on random access withuot actually iterating all properties from before. Thus, we need to do the mapping at DmoMeta generation. This can be a simple array (i-th position is the array holds the proper position in a ResultSet that selects all properties in order - the very usual case). For partial hydration, we can't compute it in advance, so we need to compute the mapping for each LazyResultSet / Hydrator.

Ovidiu, I have the idea pretty clear in my head; thank you for the guidance. I delayed this approach just because it represented only 20% of the cases in my tests, so I wanted to have some preliminary results on the other 80%. Currently, the improvement is -2%. If we are to tackle the rest of the 20%, maybe we can reach -2.5%. Of course, I will get them implemented sooner rather than later. My initial hopes were rather close to -5% :), so that is why I am "squeezing" the current 80% of the tests. Note to myself: Pareto was right, work really revolves around the 80/20 ratio :)

#38 - 11/08/2023 10:48 AM - Alexandru Lungu

Unique result

I improved the solution for unique result. It uses a LRUCache per session to "delay" the closing of statements and result-sets. This way, it gives the lazy hydration a chance. Results are quite good, improving the performance to -2.5% comparing to the baseline:

- There is a LRUCache (of size 1024 now, but it seems way too big) in session. The key is the SQL to be run and the value is the last result-set obtained on this session by running that SQL.
- Each uniqueResult will attempt to remove and close the cached result-set for the statement it is going to run (if any).
- It will run the statement and if the result-set will be lazy, it will register it to the session result-set cache.
- Of course, eviction policy, clearing and closing of the session triggers result-set closing.

ProgressiveResults

This is a new topic of discussion here. I've seen Eric disregarded .list for bucket 0 in ProgressiveResults to honor lazy hydration. However, this first batch is **very** used. Due to these changes, there are lots (~200) messages from c3p0 signaling it has to reparse some statements. This makes sense as .list was closing the statement immediately and now with .scroll, the statement is closed when the FWD query ends (or the delegate is changed).

I readded .list to see the performance difference. With this added back, c3p0 is running fine (without the need to reparse statements).

JMX	Testing run	Delta
VALID TOKEN	45099	-4705
INVALID TOKEN	1172	-50
RECORD LAZILY HYDRATED	10119	-6485
RECORD FULLY HYDRATED	1024	+330
LAZY RESULT SETS CREATED FROM SCROLLABLE	12107	-5486
NON-LAZY RESULT SETS CREATED FROM SCROLLABLE	2779	-793
LAZY UNIQUE CREATED	8613	-6

NON-LAZY UNIQUE CREATED	2375	0
SUCCESSFUL REFRESH	1172	-50
FAILED REFRESH	0	0
RECORD LAZILY HYDRATED, NOT REHYDRATED	4940	-852

- There are 6.5k less records (re-)hydrated due to the fact that there are 5.5k less scrollable queries made lazily. This makes sense, as these are now .list operations. These implies less valid tokens, etc.
- From observation, there are less statements reparsed

The performance I get by reintroducting list for ProgressiveResults is way worse (+3.5%). **However**, I am concerned by the fact that having list for ProgressiveResults and lazy hydration for other items is that bad comparing to the baseline. It means that the lazy hydration is an overhead for other items after all (I just need to detect where the lazy hydration under-performs). Are that 1.1k refreshes that bad? Or maybe my checks regarding datetime-tz fields?

Before moving on, I want to detect if we are falling behind with all these changes. I will recheck the changes first thing tomorrow morning and retest - hopefully there is no testing mistake I've done here.

#39 - 11/08/2023 09:23 PM - Ovidiu Maxiniuc

Alexandru Lungu wrote:

> Indeed! The "difficulty" here was that we can't compute the proper offset on random access withuot actually iterating all properties from before. Thus, we need to do the mapping at DmoMeta generation. This can be a simple array (i-th position is the array holds the proper position in a ResultSet that selects all properties in order - the very usual case). For partial hydration, we can't compute it in advance, so we need to compute the mapping for each LazyResultSet / Hydrator.

If we switch to BitSet representation and the columns are in a fixed given order the we need to iterate the BitSet, not the RecordSet. You can find the offset of the property k in current row by the formula:

```
i_rs(k) = SUM( property(i).getType().propertySize(), for all i < k )
```

However, it is not recommanded to do this in random access, but rather incrementally, updating i_rs as the properties are hydrated.

If incremental processing of the properties in BitSet is not possible, the i_rs(k) can be computed at first access and cached as an index array (i_rs[k] = @i_rs(k).

#40 - 11/09/2023 11:11 AM - Alexandru Lungu

- % Done changed from 50 to 70

> If incremental processing of the properties in BitSet is not possible, the i_rs(k) can be computed at first access and cached as an index array (i_rs[k] = @i_rs(k).

Nice idea! "Computing the offsets lazily while lazily hydrating" sounds like a top-notch optimization :) Understood and will do!

ProgressiveResults

I tested with and without .list - this was the only difference between the patches I profiled.

- baseline: ~8.415s
- lazy hydration without list: 8.244s
- lazy hydration with list: 8.722s (**way slower**)

This confirms my suspicion that lazy hydration changes are slowing down the execution, unless it is used by the first brackets of ProgressiveResults to counter-balance. Furthermore, it means that the other 80% of the records that are lazily hydrated (from other .scroll or .uniqueResult operations) are not actually providing a boost. This is something I need to further investigate. Hopefully we can turn around that +5% performance decrease into improvement.

Fix datetime-tz and extent cases + Refresh hydrator session

This was way more complex than I expected. I finally used RecordMeta.columnIndex to identify the proper places of the properties in result-set (for cases where all props are loaded). Also, the extent case was a bit tricky especially for "non-expanded" (i.e. normalized) cases. In this case, we couldn't use readProperty, but we need to load the extents. I didn't find a way to load only the extent we needed, so I just loaded all norm. extents of that record. Maybe we can fix this at the same time with the partial hydration (that also eagerly fetches the normalized extents). Long story short, the whole hydration logic was moved to Hydrator, that now is aware of the RecordMeta. BaseRecord is only calling hydrator.hydrate(this, offset, getHydratorOffset()) (getHydratorOffset is 1 for base record and 2 for temp record).

Also, I had implemented a routine to "refresh" the session of a hydrator when a DMO is attached back to a new session. Basically, Persistence.getSession will reassign the new session to the hydrator (if a new session is created). This is because I had failing refresh attempts (~100) due to a non-existing session to be used.

The statistics were made without .list calls in ProgressiveResults.

JMX	Testing run	Delta (comparing to 6720-33)
VALID TOKEN	53277	+3473
INVALID TOKEN	1253	+31
RECORD LAZILY HYDRATED	19627	+2034
RECORD FULLY HYDRATED	360	-334
LAZY RESULT SETS CREATED FROM SCROLLABLE	21155	+3562
NON-LAZY RESULT SETS CREATED FROM SCROLLABLE	0	-3562
LAZY UNIQUE CREATED	10988	+2375
NON-LAZY UNIQUE CREATED	0	-2375
SUCCESSFUL REFRESH	1253	+31
FAILED REFRESH	0	0
RECORD LAZILY HYDRATED, NOT REHYDRATED	5599	+193

- Having 0 on non-lazy scrollable or unique-result is the best we could achieve (all unique and scroll queries can generate lazy hydrated records).
- Still having 0 failed refresh attempts - this is on the right track.
- Remaining concerns:
  - there are still records that are fully hydrated. My best guess is that all of these are DMOs that are partially hydrated. **This is where Ovidiu's suggestions will kick in**.
  - of course, the topic of ProgressiveResults

I will have to clean up the code to make it more visually appealing and start a round of profiling tests. Tomorrow morning I will have the commit.

#41 - 11/10/2023 10:37 AM - Alexandru Lungu

I run some regression testing and they passed. I also got them a second round to extract some insights.

JMX	Testing run
VALID TOKEN	858.088
INVALID TOKEN	005.613
RECORD LAZILY HYDRATED	305.656
RECORD FULLY HYDRATED	001.821
LAZY RESULT SETS CREATED FROM SCROLLABLE	100.807
NON-LAZY RESULT SETS CREATED FROM SCROLLABLE	0
LAZY UNIQUE CREATED	062.484
NON-LAZY UNIQUE CREATED	0
SUCCESSFUL REFRESH	005.613
FAILED REFRESH	0
RECORD LAZILY HYDRATED, NOT REHYDRATED	162.167

- The same "good" ratio between valid/invalid tokens is kept here.
- There are way more lazily hydrated records and lazy result sets (over scrollable and unique) in this test-case.
- Still no failed refresh.

With the latest changes (including lazy hydration for records with datetime-tz fields and normalized extents), the performance improve is only -1.5% now. My previous attempt was -2.5% (that excluded datetime-tz and norm. extent fields).
Even if the statistics are better, I am still struggling "descending on the performance ladder". I've run the LTW AOP tests we have and I couldn't spot anything out of the ordinary from Hydrator or BaseRecord.

Committed 6720b/rev. 14471

#42 - 11/23/2023 10:02 AM - Alexandru Lungu

Status update:

Ovidiu, I followed your suggestion on having a BitSet to store the selected fields. However, I quite over-engineered this, but hope it is for the best:

- RowStructure: interface which allows defining a row structure (how the fields are selected from the database for a single DMO)

- AbstractRowStructure: base implementation. This stores only some base information as the DMO class.
- FullRowStructure: row structure that is used when not using any FIELDS clause (we want all properties). This will be used in most of the cases where we don't have partial hydration. We can even cache this instance (one per DMO).
- OrderedRowStructure: row structure that is implemented using a BitSet. This is right according to your suggestion of avoiding maps and keeping a bitset for the selected fields, presuming the fields are in order.
- UnorderedRowStructure: row structure that is implemented using LinkedHashMap. This is what we had now and keeps the columns of a DMO unordered.
- AdaptiveRowStructure: row structure that starts as an OrderedRowStructure, but if the order is broken, it will invalidate towards an UnorderedRowStructure.

This is an optimistic approach, hoping for a good row order. I am doing this to avoid any unexpected direct requests to our persistence layer that doesn't honor the fields order (from out-side FWD maybe). The only down-side here is that the AdaptiveRowStructure may invalidate and the properties are reiterated.

I just finished setting this up, I am now reworking SQLQuery to honor this new suite of row-structures. The goal here is to make lazy-hydration work only with FullRowStructure and OrderedRowStructure in order to have a proper identification of fields. In case of FullRowStructure, we keep the current implementation. In case of OrderedRowStructure, we do the lazy offset mapping.

#43 - 11/23/2023 07:37 PM - Ovidiu Maxiniuc

That is an interesting approach. I did not realized that level of specialization would be necessary. This class was a simple int parameter in the initial implementation :-). I assume these classes have specific methods used for interpreting the result (including the hydration). There were some discussions related to calling related APIs from hand-written code, in which the result-set is not mandatory a projection (PK list) or (partial) records. In this case the Java data types should (probably) be kept and not mandatory wrapped as BDTs. Having various RowStructure allows the caller to specify this, too, maybe even with anonymous classes/ lambdas.

Is there are reason for having a RowStructure interface and an AbstractRowStructure abstract class? If not maybe we can collapse them to a flatter inheritance tree.

Looking forward to see the new code!

#44 - 11/29/2023 07:45 AM - Alexandru Lungu

Ovidiu, I finished the implementation completely and tested it over a large POC. However, I created a separate branch I intend to merge independently from lazy hydration. The changes were hard to test with lazy hydration on anyway.
Thus, please review 6720c - I didn't profiled them yet, but they are mostly refactoring, so I don't except a large performance change - but a way to allow us couple lazy hydration with partial hydration.

My goal before merging 6720c is to ensure the overly used FullRowStructure is as fast as possible. I am already doing lots of presumptions comparing to the previous implementation to allow us to read the properties without any overhead.

Mind that all structures were tested on POC - only 3 invalidations happened in the process, but they were very slim (after 2 properties).

#45 - 11/29/2023 10:21 PM - Ovidiu Maxiniuc

I grabbed 6720c and looked at the latest changes. Indeed, I do not see a reason for a change in performance. The hydration (and some related operations) was moved to new classes, but it's mainly the same.
However, I am not sure I fully understand the new mini-architecture. I need to read the new code one more time for that. I will write the detailed review after that.

#46 - 12/04/2023 04:14 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> I grabbed 6720c and looked at the latest changes. Indeed, I do not see a reason for a change in performance. The hydration (and some related operations) was moved to new classes, but it's mainly the same.
> However, I am not sure I fully understand the new mini-architecture. I need to read the new code one more time for that. I will write the detailed review after that.

Just to be clear of the intention. The further 6720b (that has partial hydration) requires different approaches for different row structures:

- In case of a full row structure, we can access the result-set mostly on the same indexes as the request. So, we can arbitrarily hydrate one property without much index mapping fuss.
- For incomplete records, so in case of an ordered / unordered row structure, we can compute the index mapping (and eventually cache it as you suggested). Thus, the partial hydration will do some kind of index mapping:
  - In ordered row structure, we keep a bitset. Checking if the property was retrieved from database is very fast.
  - In unordered row structure, it is quite hard to detect if a property was retrieved from the database or not - I will need to add a bitset to the unordered structure as well to manage that.

The whole overhead here is to be able to hydrate **only** one random property, but with different methods (faster in full row structure, slower in ordered row structure and even slower in unordered row structure). Maybe we can even rule out partial hydration in some cases (if the DMO is incomplete and the number of columns requested is less than a threshold?).

#47 - 12/08/2023 04:34 PM - Ovidiu Maxiniuc

Alexandru, sorry for the delay. The size and the nature of changes (class tree, virtual methods) required ore time than I initially expected.

First, a question: how slow is the hydration of a field? I did not do a profiling here (I might do it in the future), but you may have the answer. For example, hydrating only 50% of the fields of average sized record (let's say 10-20 fields). This is a difficult question: the integer, logical and may character data are fast, but date, decimals and the rest, may be visible on the timer.
I am asking because I see that the code for supporting lazy hydration (not partial hydration - when some fields of a record are never fetched from database; this is handled by a different task by detection of the filed set at conversion time) is getting more and more complex. I hope this does not weight more than the full hydration if a record is hydrated in two or more 'stages'.

Now the review:

Good job. A lot of new code, which should particularize each case and optimize hydration independently. The code seems logical now, after re-reading it several times. I thought at first to spot some issues with reserved properties (because they have negative property ids), but now I think they are correctly hydrated.

Generally, I think passing the DmoMeta object instead of the DMO interface as parameter, is better, if possible. This is because the table metadata is stored there, including the DMO itself, but when the meta data is needed, obtaining it knowing the DMO implies a lookup over a large map (note to self: investigate the registries of DmoMetadataManager from the viewpoint of (initial) dimension, load factor, in cases of large enough client

applications).
We agreed to avoid normalized/denormalized terms. Instead we should use not expanded/expanded.

My notes below are mainly related to code formatting/style and small local optimizations than actual bugs:

- AdaptiveRowStructure.java:
  - the javadocs of methods addRecid(), addMultiplex(), addProperty() are missing the @return tag;
  - hydrate() method is missing the @throw tag;
  - extra parameter count in c'tor's javadoc;
  - the addProperty() parameter name in javadoc does not match the argument name;
  - in invalidate() method: instead of comparing the prop.name (strings), the property.id should be tested with ReservedProperty.ID_PRIMARY_KEY and ReservedProperty.ID_MULTIPLEX respectively. They are integers so a switch is optimal;
- DmoMeta.java, FqlToSqlConverter.java, ScrollableResults.java, DirectAccessHelper.java: missing history header entries
- RowStructure.java
  - imports, preferred * instead of individual classes. Are there name collisions I failed to see?
  - as for AdaptiveRowStructure, the javadocs lack the @return and @throw tags;
- SQLQuery.java:
  - missing history header entry;
  - line 798/799: if the rowStructure had DmoMeta as member, the dmoInfo and recordClass would be obtained directly (see previous paragraph in this note)
- AbstractRowStructure.java
  - * imports instead of individual classes;
  - I see you added DmoMeta as a member here, by lookup from DmoMetadataManager, but the best solution is to have this the other way around;
  - addProperty() method is missing the @return tag;
  - hydrateExtents() lacks and @throw tag;
- FullRowStructure.java:
  - see above (imports, javadoc tags);
  - in hydrate(), when skipping the PK and _multiplex I would test whether the next() property is the right one. This would add a bit of robustness to code;
  - isIncomplete(): missing the @Override annotation
- RowStructure.java: see above (imports, javadoc tags)
- OrderedRowStructure.java
  - see above (imports, javadoc tags)
  - addRecid(), addMultiplex(), adding the PK or _multiplex twice (or more) will return true and continue incrementing the count.
  - getProps(): RECID_PHASE and MULTIPLEX_PHASE can be also static. Maybe use ReservedProperty.ID_PRIMARY_KEY and ReservedProperty.ID_MULTIPLEX?
    - in hasNext(), the phase advances. I think that's incorrect. Calling hasNext() multiple times while in RECID_PHASE and MULTIPLEX_PHASE will advance the iterator;
- UnorderedRowStructure.java
  - see above (imports, javadoc tags);
  - hydrate():
    - line 241: fieldCnt cannot be 0. If it was (fieldCnt = fields.size()), then the execution would not have entered the for loop. What was the desired meaning of this code?

#48 - 12/11/2023 06:02 AM - Alexandru Lungu

Committed 6720c/14847 including the suggestions you made.

- Regarding fieldCnt, it should have been 1 to short circuit projection queries. Internally, all our projection queries are OrderedRowStructure, but anyway, I've honored this for UnorderedRowStructure as well. That change is irrelevant in our current state of affairs.
- Used ID_MULTIPLEX and ID_PRIMARY_KEY instead of the phase variables. The iterator works properly. If you call hasNext, the phase will advance only if it points to an invalid state (phase is recid but there is no recid). Otherwise, the method is idempotent.
- Renamed all denromalized/normalized references into not expanded/expanded.

I can't give you an answer on the hydration timing on the latest changes. On the initial code, there was clearly an improvement of ~2%, but was omitting the partial hydration (see #6720-41). I wanted to go all the way and support lazy hydration for partial structures as well. But this wasn't quite possible without this refactoring - allowing me to do the proper mappings between the requested fields and their result-set positions. Is it worth it? I don't know; maybe the mappings will slow down the process such that doing the lazy hydration will be in fact slower. At that point we can certainly rule out partial hydrations from lazy hydrations.

Anyway, I think the new architecture is worth having anyway, as we can optimize it "per-case", have a distinction between partial hydration / full hydration and "custom hydration" (outside FWD).

This was largely tested with a customer app and its POC (regression + performance). I intend to have it to trunk, rebase 6720b, and resume work on lazy hydration. Let me know what you think,

#49 - 12/11/2023 12:13 PM - Ovidiu Maxiniuc

I am glad you already tested the implementation and you found no regressions. I re-analysed the iterator from OrderedRowStructure and, indeed, it looks correct. I was fooled by the cascading if s which makes the 'cursor' jump directly to right phase. Nevertheless, it is a bit unusual to change the internal state of the iterator in this method.

I still have some suggestions, if possible to address before merging to trunk:

- are getDmoMeta() and getDmoClass() methods necessary? I understand that the values they return are used only 'internally' within the classes extending the RowStructure. I think a better approach would be to make those fields protected and final and access them directly (the dmoClass already is). I do not expect the result to be visible in profiler, but it might help;
- AbstractRowStructure.addProperty() still lacks the @return javadoc tag;
- AbstractRowStructure.toString() I would replace the RowStructure literal with getClass().getName() for easier debugging;
- FullRowStructure, line 211, the comment should read "skip _multiplex".

#50 - 12/12/2023 03:37 AM - Alexandru Lungu

Fixed the last concerns. Committed 6720c / rev. 14848.

Redoing quick profiling and regression tests now for 6720c and preparing for merge. I am planning to resume the work on 6720b asap.

#51 - 12/12/2023 10:40 AM - Alexandru Lungu

Completely fixed 6720c. I had some "invisible" regressions there. Also, it was quite confusing why the AdaptiveRowStructure was invalidated so often. I misused the RecordMeta API - there are close to no invalidations now.

Performance-wise, it is still on par with the baseline.

6720c is ready for merge.

#52 - 12/12/2023 10:59 AM - Greg Shah

You can merge to trunk now.

#53 - 12/12/2023 11:13 AM - Alexandru Lungu

Branch 6720c was merged to trunk rev 14869 and archived.

Resuming work on 6720b

#54 - 12/14/2023 10:07 AM - Alexandru Lungu

- % Done changed from 70 to 90

Adapted 6720b according to the row structure architecture. Continuing with testing and tracking.

#55 - 12/19/2023 03:58 PM - Greg Shah

How far along is 6720b?

#56 - 12/20/2023 08:29 AM - Alexandru Lungu

I am "battling" with a small regression when averaging the new row-structure architecture with lazy hydration. I was also ready to merge only lazy hydration of full records to have "something" on this merged, but even there we have a problem. I am roll-backing some changes to see where the problem occurred, but is a bit tedious. I will do my best to get a working solution by the end of this week.

#57 - 01/05/2024 05:12 AM - Alexandru Lungu

Eduard, please assist the regression hunting here. I am aware that you have a more suitable environment for regression testing and debugging, so feel free to check-out 6720b, move it to 7156b and start regression testing. I can do the rebase for you as I am more familiar with the details if needed. We can have a talk on the changes here. Thank you.

#58 - 01/19/2024 01:29 AM - Eric Faulhaber

Is this far enough along that we can measure performance, or are the regressions blocking testing? We need to understand the impact for both PostgreSQL and MariaDB.

#59 - 01/19/2024 03:21 AM - Alexandru Lungu

On the POC, it regresses and crashes. I didn't attempt the tests on the new test suite.

#60 - 02/13/2024 02:22 PM - Alexandru Lungu

Short update from a live discussion with Eduard: the regression seems to be related with the BaseRecord.getData calls that expose the "partial" data array out-side the record. This is not OK, because the data consumers are not aware of the lazy hydration encapsulated in BaseRecord. From my understanding, Eduard found this issue in a snapshot attempt of the record. The data was copied from a cached record to a snapshot record.

There are many places where getData is used, but I think that some of them don't actually require the fully hydrated data array **or** some of them can

share the same hydrator. The trivial fix is to fully hydrate when doing a getData, but won't this nullify a great amount of effort invested in lazy hydration?

Eduard is working on analyzing how often we will fall in the pitfall of fully hydraation due to getData calls.

#61 - 02/14/2024 02:09 AM - Eduard Soltan

On delete of a temporary buffer and a deleted buffer is placed in before table.

But to make this copy a snapshot of the current buffer is made, and this snapshot is copied into before buffer.
However, to create a snapshot copy method from BaseRecord is called, where data array is copied without copying hydrator and liveporps.

And if the current record has some fields that are not hydrated, and unknown value for this fields is propagated into before table.

#62 - 02/14/2024 04:02 AM - Alexandru Lungu

Lets go ahead with some statistics. From where is getData called and how many times. Please find the back traces of getData, add JMX counters and report how many times and from where it is called (in 7156b without 6720a). To test this, you can add getData() as in 6720a.

#63 - 02/14/2024 07:30 AM - Eduard Soltan

I changed BaseRecord.copy, to set hydrator and liveProps of the new record

```
public void copy(BaseRecord from)
   {
      // TODO: BLOB data is mutable, needs to be duplicated explicitly
      int len = this.getLength();
      this.id = from.id;
      copyArray(from.getData());
      dirtyProps.set(0, len); // TODO: nullProps?

      hydrator = from.hydrator;
      liveProps = from.liveProps;
   }
```

Running POC does not caused crashing any more.

And made some profiling on 7156 with and without changes from 6720.

1) clean 7156, number of times getData is called for running POC (warm and 1 test) is 1200000.
1) 7156 rebased with 6720, number of times getData is called for running POC (warm and 1 test) is 100000.

Majority of calls to getData are made to get data of a field of a buffer.