Database - Bug #7185

H2 in-memory lazy hydration

03/09/2023 09:03 AM - Alexandru Lungu

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

Description

Related issues:

Related to Database - Feature #2137: runtime support for FIELDS/EXCEPT record...	**New**
Related to Database - Feature #6721: calculate field lists at conversion time...	**New**
Related to Database - Feature #6720: lazy hydration	**WIP**
Related to Database - Bug #7174: Resolve simple CAN-FIND statements faster	**Test**

History

#1 - 03/13/2023 07:37 AM - Alexandru Lungu

There is already some effort into lazy hydration as a general topic:

- supporting FIELDS/EXCEPT clause, so we can fetch "incomplete" records (#2137)
- a lead to calculate field lists at conversion times to avoid fetching unnecesarry properties (#6721)
- lazy hydration as expected from #6720

Most probably, #6720 will help a lot the adjacent "lazy hydration" related work through some complex result-set lifecycle management. However, the H2 _temp database can simply benefit from any naive kind of "lazy hydration" due to the fact that the data is already in the memory. My point is that we can avoid hydrating a whole DMO at once just because the data is already somewhere in the memory, but wrapped in a H2 specific representation. There is no need to eagerly deserialize/serialize it into the FWD representation if not needed.

1) Retrieving data from H2 using SQL provides us a result-set containing record snapshots. A JDBC ResultSet is in fact an instance of org.h2.result.LocalResultImpl, wrapping a ArrayList<Value[]> rows.

2) The FWD hydration process traverses **all** fields (unless FIELD/EXCEPT is specified) and adapts the data representation to BaseDataType. This way we are mostly working with fully hydrated DMOs. This process actually traverses an entry from rows (basically a Value[] instance) and converts from H2 representation (ValueString) to Java representation (String) to FWD representation (character).

3) We are eagerly fetching, so we are always adapting representation for all fields, even if only some (2/3 fields) are going to be used. By now, we can see why #6720 is relevant. However, the H2 _temp database itself is already storing the data in memory, so we are not facing the network and serialization overhead from #6720. We can tackle this naively!

4) I've encountered several examples in large customer applications with for-each loops over temporary tables with ~20 fields (without _temp specific columns like multiplex) that accessed only 2 fields. There is some work in H2 to retrieve a single field. It is not much, but it is called million of times redundantly. The FWD work for a single field is also non-trivial as BaseDataType instances are created and, sometimes, complex logic is triggered (e.g. handle.fromResourceId).

- check the column index is between bounds
- check that the connection, session, statement, result-set are not closed
- check that result-set is not out of out of bounds
- convert to Java representation (rarely, only for data-types like ValueTime, ValueDate, ValueTimestamp and ValueTimestampTimeZone)

5) I used "naively" several times already. This is because we can use the readFields mechanism (from #2137) straight ahead:

- store a reference to the Value[] retrieved by the result-set in TempRecord (lets name it "fallback")
- presume that any TempRecord has initially empty fetchedFields (very similar to readFields)
- override checkIncomplete or create checkFetched, to automatically fetch the data at the specified index from the stored "fallback". Mark it as read. The fetchedFields should always be a subset of readFiels.
- BaseRecord.data is *almost* fully encapsulated
  - any complex logic can trigger complete eager fetching to ensure consistency with the current implementation. Remove the fallback after complete eager fetching to free memory faster.
  - getData is not that widely used. From my POV, it is best to remove it and encapsulate behavior in BaseRecord.

6) There is a quite big difference to #6720: there is no need to match the result-set lifecycle to the buffer. The row is in memory, so we need just to reference it through H2 direct-access.

7) I acknowledge there are some downsides of this approach:

- the H2 representation will live as long as the DMO lives. Therefore, there will be a slight memory increase. However, if we do the fallback management responsible enough, we can make this limitation negligible.
- the H2 representation is bound to the TempRecord. So we have an "external" dependency right from the DMO. However, it is basically a composite relationship to the H2 instance, so our DMO will exclusively dominate the snapshot, making it self-contained.

I think point 5 can also apply to #6720 if we use a proper interface. However, there is a way more complex mechanism to keep track of ResultSet lifecycle there, as we don't have direct-access.

Eric, please let me know of your progress in #6720 to avoid having two implementations for lazy hydration. I can match this "H2 direct-access fallback" on your structure if needed. Otherwise, we can go with something similar to point 5..

#2 - 03/13/2023 07:38 AM - Alexandru Lungu

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

#3 - 03/13/2023 07:38 AM - Alexandru Lungu

- Related to Feature #6721: calculate field lists at conversion time where possible added

#4 - 03/13/2023 07:38 AM - Alexandru Lungu

- Related to Feature #6720: lazy hydration added

#5 - 03/14/2023 07:14 AM - Alexandru Lungu

- Related to Bug #7174: Resolve simple CAN-FIND statements faster added

#6 - 03/14/2023 07:15 AM - Alexandru Lungu

- Assignee set to Dănuț Filimon

- Status changed from New to WIP

#7 - 03/14/2023 07:33 AM - Alexandru Lungu

Danut, please focus on implementing a demo for point 5 - we should have a fast mock to test its performance. Therefore, make sure that SQLQuery.hydrateRecord is actually using direct-access to generate a fallback for the TempRecord to the values provided by the result-set. This way, we don't need to go through all columns and read the properties, but just reference the value array provided by the result-set. We will hydrate when we will access data.

Make sure you use DirectAccessHelper as a gateway to the H2 internals; we should avoid spreading H2 dependencies through the FWD code.

The "fallback" from TempRecord (or more specific, TempTableRecord) should be a higher level representation (an interface) to allow any further work on this area with other means of lazy hydration.
Make sure you hydrate from the fallback when needed. That is, when you access a specific index from data that wasn't fetched yet. Override checkIncomplete for this, at is already spread in all getters.
There is some extra work with data (like in the case of buffer-copy), where you will need to do a complete hydration. Implement such mechanism to do complete hydration and use it when needed. Make sure you don't hydrate the same field multiple times.

#8 - 03/16/2023 09:34 AM - Alexandru Lungu

> override checkIncomplete or create checkFetched, to automatically fetch the data at the specified index from the stored "fallback". Mark it as read. The fetchedFields should always be a subset of readFiels.

This is wrong. I see that readFields is set exactly when a field is hydrated from the result-set, not when required through FIELDS. Therefore, for lazy hydrated records, readFields is empty at the start.
There shouldn't be two bitsets (fetchedFields and readFields) but only one which stores the read fields. Basically, readFields should be enough.

I think you can use fallback only for fully hydrated records for now. It is an overshoot for now to support this with partially hydrated records anyways.

#9 - 03/28/2023 02:57 AM - Dănuț Filimon

Committed 7185a_h2/rev.14 and **7185a/rev.14528** as part of the initial implementation for lazy hydration.

I've made the following changes:

- Instead of storing a reference to the Value[] from the current row of a ResultSet in TempRecord, I store a ResultSet that only contains the respective row and is obtained through DirectAccessHelper.getJdbcResultSet.
- Instead of fetchedFields, I use the **fallback** and the BaseRecord.readProperty method to update BaseRecord.data.
- Added a new boolean property to DmoMeta called hasExtentFields, which will be used to avoid retrieving the ResultSet for DMOs that have extent fields.
- I relied on DirectAccessHelper for any operations requiring H2 and avoided importing new packages inside FWD.

Please review the changes.

#10 - 03/28/2023 05:27 AM - Alexandru Lungu

Review for 7185a/rev. 14528

- In TempRecord:
  - readFields.set(i); is redundant as readProperty already sets the readField.
  - allowsFallback is redundant; you already have a fallback variable, so you allow fallback.
  - Make sure you don't read a property twice. Therefore, use readProperty only when readFields.get(i) is false.
  - Are you sure for (int i = 0; i < data.length + 3; i++) is right? You do readProperty(i, fallback, i + 3);, so you may read for data.length + 2 offset, which is out of bounds. I guess you should just use for (int i = 0; i < data.length; i++). Please double-check.
- In BaseRecord:
  - return this instanceof TempRecord; this is not close to OOP standards. Please override allowsFallback in TempRecord and return true from there; return false from BaseRecord.
- In DirectAccessHelper:
  - Change the direct access error message; use something like "can't extract current row from result-set". I think is better to throw a DirectAccessException instead of IllegalStateException. This will allow a nice recovery to the JDBC/SQL method of hydrating.
- In SQLQuery:
  - If you catch DirectAccessException, just do a SEVERE log and allow the standard JDBC/SQL way of hydration; don't associate and return an invalid record; just try again with the other way of hydrating.

There are many usages of data straight-away that I missed in the first iteration. Please make BaseRecord.data private. Enforce any access to data through getters. getDatum should hydrate eagerly. **Don't use** checkIncomplete for getDatum, as in some cases getDatum should work with incomplete records. Just ensure you always honor lazy fetching when you do getDatum. Also, do full-hydration when using getData. Replace all direct usages of data with getters (on one element or on all elements).

Please mind the setters. setDatum(i) should be done only after you've hydrated (this is implicit after you do getDatum(i)). Now that I am thinking, we may see a slowdown if we enforce access to the low-level data array through getters. I think this was something we tried to avoid already. Please pend until some further advice.

Eric, Constantin, please advice. If we attempt to implement lazy-hydration in any form (including this naive in-memory technique for H2), we may need to enforce access to data using getters so that we honor the "fallback" (the result-set). Is this something acceptable, as long as we see that data is used directly in so many places (ScollableResults, Persister, Validation, etc.)? There are encounters like dmo.data[offset] null in Validation that should be replaced with dmo.getDatum(offset) null to honor lazy fetching.

Review for 7185a_h2/rev. 14
The changes are meant to support the operation of "extracting the current row from a result-set and wrapping into another dummy result-set"

- Please rename FWDDirectAccessDriver.getJdbcResultSet to FWDDirectAccessDriver.extractRow or something similar.
- There is an implicit cast for ((JdbcResultSet) rs).getResultInterface(). I guess H2 will always use JdbcResultSet as their result-set implementation, but a FWD developer is not constrained to pass any kind of ResultSet implementation there. Please make a type check. If rs is not JdbcResultSet, throw an SQLException. This is more friendly then a ClassCastException.
- Please do a double-check for what can be null or not. I am concerned that ((JdbcResultSet) rs).getResultInterface() may return a null result. Or the current row may be null, as the result-set cursor is out of bounds.

When doing direct-access, it is best to take the safest route, throwing SQLException whenever an unexpected state is reached (something is null, database objects are not found, etc.). The SQLException will be wrapped into a DirectAccessException by DirectAccessHelper. Ultimately, anytime you catch a DirectAccessException, you should log a severe and fallback to the "JDBC/SQL way".

#11 - 03/28/2023 07:16 AM - Alexandru Lungu

- % Done changed from 0 to 50

#12 - 03/28/2023 09:50 AM - Dănuț Filimon

Based on #7185-10 I committed **7185a/rev.14529** and **7185a_h2/rev.15.**.

- Fixed TempRecord.
- Made BaseRecord.java private which resulted in several changes to use getters and setters.
- Created setSimpleDatum which is used by BaseRecord to update it's data property directly.
- Renamed DirectAccessHelper.getJdbcResultSet to DirectAccessHelper.extractCurrentRow and made changes to error messages and logs.
- Added null checks in FWFDirectAccessDriver.extractCurrentRow.

#13 - 03/29/2023 05:19 AM - Dănuț Filimon

Committed 7185a/rev.14530. Fixed a NPE in TempRecord.getDatum() where fallback was assigned but readFields was null.

#14 - 03/29/2023 06:03 AM - Alexandru Lungu

- % Done changed from 50 to 70

Review for 7185a/rev. 14529 and 14530

This is close to a conceptually correct solution. However, there are still some functional concerns:

- copyArray is slow. You should still use System.arraycopy; no need to use setSimpleDatum on each index. Just do System.arrayCopy, set all readFields and set fallback on null.
- In setDatum, extract the datum in a separate variable and use it in the conditional. Avoid doing 3 calls to getDatum in the same conditional. There are several places in which you can do such optimization. In Record.setPropertyValues, you can extract getData() in a separate data variable and use it in each for iterator. This avoids calling getData for each property.
- You can just return ResultSet instead of JdbcResultSet from extractCurrentRow, no need to explicitly cast.
- Line is too long in SqlQuery. (123 chars)
- Improve TempRecord.fallback javadoc: "Dummy result-set used for lazy hydration. Can be {@code null}. Lazily hydrated fields are marked in {@code #readFields}.".
- Don't expose getFallback; fallback shouldn't leave TempRecord. In setFallback, please throw an IllegalStateException if fallback was already set (fallback != null).
- In TempRecord, you override only getData(PropertyMapper); getData() and getData(RecordBuffer) are not aware of the fallback. Please override only getData and make the other two depend upon it.
- Please respect the "method access level" order. Private methods should be always last, not in between public methods. Eclipse, for instance, has a nice outline perspective with method access level colored; keep it consistent.
- I think extending checkIncomplete becomes redundant, as the fallback is honored by getDatum anyways. You can remove TempRecord.checkIncomplete; correct me if I am wrong.

#15 - 03/29/2023 06:28 AM - Eric Faulhaber

Alexandru Lungu wrote:

> **Eric, Constantin, please advice**. If we attempt to implement lazy-hydration in any form (including this naive in-memory technique for H2), we may need to enforce access to data using getters so that we honor the "fallback" (the result-set). Is this something acceptable, as long as we see that data is used directly in so many places (ScollableResults, Persister, Validation, etc.)? There are encounters like dmo.data[offset] null in Validation that should be replaced with dmo.getDatum(offset) null to honor lazy fetching.

The idea when I first implemented BaseRecord was to make the data array directly accessible as an instance field to other classes in the package and to subclasses. This was done with performance in mind, knowingly sacrificing encapsulation.

However, at that time, I did not contemplate the complication of lazy hydration, nor other use cases which would make direct access to data less practical. In places where lazy hydration makes that direct access inadvisable, go ahead and do what needs to be done. That being said, I want to be very careful about adding a lot of logic around each access to an element in the data array, which could impact performance negatively. The point of lazy hydration is performance, after all.

Where we can continue to support faster access to the data array, even if is through fast, minimal methods, rather than directly as a field, I want to do that. In other words, just use the lazy-hydration-aware access where absolutely necessary. Hopefully, that doesn't turn out to be everywhere, or if it is, hopefully the cost savings we get from doing that outweighs whatever we are giving up by removing direct access to the array.

#16 - 03/29/2023 07:15 AM - Alexandru Lungu

Eric Faulhaber wrote:

> The idea when I first implemented BaseRecord was to make the data array directly accessible as an instance field to other classes in the package and to subclasses. This was done with performance in mind, knowingly sacrificing encapsulation.

Makes sense.

> However, at that time, I did not contemplate the complication of lazy hydration, nor other use cases which would make direct access to data less practical. In places where lazy hydration makes that direct access inadvisable, go ahead and do what needs to be done. That being said, I want to be very careful about adding a lot of logic around each access to an element in the data array, which could impact performance negatively. The point of lazy hydration is performance, after all.

I agree. I didn't have a mean of comparing easy access without lazy hydration vs getter/setter access with lazy hydration. I was curious if we can have a stable implementation so we can continue exploring our options.

> Where we can continue to support faster access to the data array, even if is through fast, minimal methods, rather than directly as a field, I want to do that. In other words, just use the lazy-hydration-aware access where absolutely necessary. Hopefully, that doesn't turn out to be everywhere, or if it is, hopefully the cost savings we get from doing that outweighs whatever we are giving up by removing direct access to the array.

Well, right now:

- for Record, we have basic getters for getDatum and getData. This falls into "fast, minimal method" category.
- for TempRecord, we have:
    - If fallback exists and the datum is not read, a procedure to read the property. This is time consuming OFC.
    - If the datum was read or the fallback was consumed, a super call; this falls into "fast, minimal method" category with a simple if check.

#17 - 03/29/2023 09:02 AM - Dănuț Filimon

Based on #7185-14 I committed **7185a/rev.14531**. I confirmed that TempRecord.checkIncomplete is redundant and removed it, I've done the same to TempRecord.getFallback because it isn't used anywhere.

#18 - 03/29/2023 11:09 AM - Alexandru Lungu

Done some tests with latest 7185a/rev. 14531 and latest 7185a_h2:

- FWDDirectAccess.retrieveResponse you created is used now in two use-cases: the former "wrap a row found by unique index in a dummy JDBC result-set" and the new "wrap a row from another result-set in a dummy JDBC result-set"
- Unfortunately, even if these look similar, they bahave differently. In the first case, a row from H2 is provided (so it includes computed columns - beginning with __i or __s). In your case, the provided result-set doesn't have computed columns. This causes an ArrayIndexOutOfBoundsException as it tries to append to the result-set the column i and the value i. However, cols.length != values.length.

Note that the computed columns are marked as invisible in H2. Therefore, you can append a boolean flag to retrieveResponse named valuesHaveInvisible and fix it this way (skip values only if they include invisible).

#19 - 03/29/2023 11:40 AM - Alexandru Lungu

Committed 7185a/rev. 14532 and 7185a_h2/rev. 16: handle cases when result-set is composed out of multiple row structures. Add an offset from which the row is extracted. Fixed #7185-18,

#20 - 04/05/2023 09:13 AM - Alexandru Lungu

Rebased 7185a with trunk (rev. 14525). 7185a is now at 14530.

#21 - 04/05/2023 10:45 AM - Alexandru Lungu

I have troubles testing 7185a. One customer application works flawlessly, while the other in hanging right at the start-up.
I will double-check the changes, in case I missed something critical.

Radu, can you please check-out 7185a and test it against adaptive-scrolling/non-scrolling tests. Note that you should do the local FWD-H2 dev. set-up (to locally load, compile and use 7185a_h2). Ensure the tests are using temp-tables. Please report any regressions you encounter there.
UPDATE: Danut, please retest in-depth the changes on a customer application. If you can't find any issues, consider doing some extensive testing with FWD, especially multi-table compound query optimized into an adaptive query.

#22 - 04/06/2023 04:36 AM - Radu Apetrii

Alexandru Lungu wrote:

> Radu, can you please check-out 7185a and test it against adaptive-scrolling/non-scrolling tests. Note that you should do the local FWD-H2 dev. set-up (to locally load, compile and use 7185a_h2). Ensure the tests are using temp-tables. Please report any regressions you encounter there.

I've done multiple tests (including parallel runs) over these changes but I could not find any regression.

#23 - 04/11/2023 09:59 AM - Alexandru Lungu

Committed 7185a/rev. 14531: small fixes. Danut, please review.
I am close to finding the regression we are chasing for some time.

#24 - 04/11/2023 10:13 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Committed 7185a/rev. 14531: small fixes. Danut, please review.
> I am close to finding the regression we are chasing for some time.

I am ok with the changes.

I also retested a customer application and found no signs of regressions. Several tests that I had were in the same situation. It is good that you are getting closer to the root of the problem.

#25 - 04/12/2023 05:56 AM - Alexandru Lungu

- % Done changed from 70 to 100

- Status changed from WIP to Review

Fixed regressions in 7185a_h2/rev. 18 and 7185a/rev. 14532

- In H2, the cursors were not properly initialized to honor the offset. Note that a result-set can contain multiple records in the same time; you need to hydrate only from a specified offset.
- In FWD, it is not enough to rule out extents, but also TZ offset (computed column for datetime_tz data type). The _i and _s columns (computed columns for character types) won't be extracted into the result-set anyway, so it is safe to lazy hydrate for such cases.

If we are going to extend such implementation, we should carefully handle the computed columns part: TZ offset column, _i and _s columns and extents. For example, if you request the 45th column to be hydrated, you may need to load the data from the 47th position in the result-set, as the computed columns cause inconsistent indexing. Right now, we apply lazy hydration only for tables which satisfy the "hasConsistentPropertyOrder" requirement (no extents and tz columns).

I run my in-depth regression testing and everything is fine now.

Danut please review. I will start the profiling session asap. I would like to know the performance change by now and retrieve a statistics of how often is the lazy hydration used and how many times the "retrieve all columns procedure" is triggered. I will post the update here.

#26 - 04/12/2023 06:30 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Danut please review. I will start the profiling session asap. I would like to know the performance change by now and retrieve a statistics of how often is the lazy hydration used and how many times the "retrieve all columns procedure" is triggered. I will post the update here.

I understand the cause of the problem now and I am ok with the solution.

#27 - 04/12/2023 11:01 AM - Alexandru Lungu

- Status changed from Review to WIP

- % Done changed from 100 to 70

My profiling session shown a +0.7% performance decrease. This is bad.

- My first concern here is the extractCurrentRow. Its implementation iterates over all columns and values. It does a copy to support the fallback. This however doesn't seem to be faster then simply hydrating directly. We should avoid doing any kind of O(N) loops (N is the number of columns). Especially as we do new Column for each such column.
- ~~The second issue is that we do a second cache check, based on the retrieved recid. This is not needed, as we already know that the record is not in cache, so we need to hydrate it.~~ This is not happening.
- Iterate faster the fields which were not read when doing TempRecord.getData. Instead of for (i = 0; i < length; i)) readFields.get(i), you can use while ((nxt = nextClearBit(nxt)) != -1). If you have large chunks of set bits, you can skip them faster with nextClearBit then get.
- TempRecord.copyArray is creating lots of readFields instances. I think this is not required. In fact, readFields shouldn't be created, but removed if exists. It is concerning that all snapshots are doing such copyArray. Usually, the target doesn't even have a fallback (it is a clear instance).
- I will check how fast is hasConsistentPropertyOrder and if it should be optimized. It is cached in DmoMeta, so I doubt it is an issue.

My point is that the required data is already in row.getValueList(). extractCurrentRow should only return a weak reference to this Value[] and eventually an offset - no need to actually iterate the columns IMO. I really hope at this point that the whole getter/setter technique is not introducing such overhead alone to rule out lazy hydration.

I will do the changes and test again.

#28 - 04/13/2023 06:49 AM - Alexandru Lungu

Committed 7185a/rev. 14533:

- Optimized TempRecord.fallback usage. Redone the profile and got to +0.25% performance decrease, comparing to my baseline. This is manageable. It seems that the whole readFields instance creation on copyArray and the lack of readFields.nextClearBit made a difference.
- I am also optimizing FWDDirectAccess.retrieveResponse to avoid creating dummy Column instances for each column in the result-set and use the table's columns instead (lazily filtered if there are invisible/computed columns intertwined). Table.columns collection is immutable, so we shouldn't worry that columns may appear/disappear from the result-set. This way we also optimize #7062 (direct-access). In my solution, there are no O(N) loops anymore (unless JDBC ResultSet columns are searched by name - not something I encountered in FWD). This will be even faster after #7108 is merged, as we shouldn't worry about invisible/computed columns anymore (neither in direct-access nor in lazy hydration).

I have a stable solution, but I am still thinking of fine-tuning it. I am confident we will turn this around to -% performance increase.

#29 - 04/18/2023 10:06 AM - Alexandru Lungu

Committed 7185a_h2/rev. 19.

It includes optimization for FWDDirectAccess.retrieveResponse:

- If no record is found with direct-access, return a single EMPTY_RESULTS instance. This way we don't create redundant results instance.
- Count the invisible columns in a table: if there are no invisible columns, let the results reference the table columns to retrieve metadata when needed.
- If there are invisible columns, let them be filtered lazily (when needed).
- Make copies of the values/columns arrays only when needed.

Testing with 7185a_h2/rev. 19, I get -0.14% performance improvement. I will retest without the lazy hydration FWD changes, just to check if the FWD-H2 changes alone are more performant. At that point we can decide what to tackle next.

#30 - 04/21/2023 10:23 AM - Alexandru Lungu

Rebased 7185a_h2 to trunk rev. 15. Not it is at rev. 20.
Committed 7185a_h2/rev. 21 including some extra optimizations. Also included the EmptyResults and LazySingleResults implementations in FWD-H2.

Changes	7156a/rev. 14529	7156a/rev. 14540
with FWD-H2 trunk rev. 15	10.046	10.090
with FWD-H2 7185a_h2 rev. 21	10.000 (-0.45%)	10.018 (-0.7%)
with FWD changes and FWD-H2 7185a_h2 rev. 21	10.059 (+0.12%)	10.072 (-0.17%)

These are the results only with FWD-H2 changes. It is a bit worrying now that rev. 14540 alone is slower then 14529 (by +0.43% without any FWD-H2 changes).
For this task's purpose, the FWD-H2 changes show a consistent improvement alone to the direct-access engine (less instances created, faster invisible column filtering, faster no-record resolution).
I will update this table with the 7185a changes on lazy hydration; from some prerequisite tests, they should provide a slight boost over all.

#31 - 04/25/2023 10:20 AM - Alexandru Lungu

I updated #7185-30 with the times regarding the FWD changes (lazy hydration). While the FWD-H2 changes provides a consistent improvement for direct-access, the FWD lazy hydration changes introduce a severe slow down (+0.5%~0.6%).
This was a bit harsh to test because of the 7156/rev. 14529 - 14540 time differences, so I had really different results between runs. Even for this +0.5%~0.6% I had runs which were only +0.1%.

My suggestion: as the FWD-H2 changes are robust and well-tested, we can move on with integrating them **now**. A -0.5% improvement is something we can take as it is for now. I would like to integrate the direct-access support for lazy hydration as-well, even though it won't be used in FWD yet. I am planning to redo the performance tests once we have a new stable baseline.

I will leave this task WIP until then. I am brushing up 7185a_h2 to merge in FWD-H2 trunk tomorrow morning. Danut, please make a quick review on 7185a_h2 before that time.

#32 - 04/26/2023 04:35 AM - Dănuț Filimon

Review 7185a_h2. The changes are good to go, you can proceed with the merge.

#33 - 04/26/2023 08:05 AM - Alexandru Lungu

Merged 7185a_h2 into FWD-H2 trunk as rev. 17.
This doesn't require any FWD changes; it just optimizes the direct-access.
The changes for lazy hydration were included, but are not "activated": the API is there, but it is not consumed.

I will do another profiling round next week after stabilizing FWD-H2 and get some customer applications reconverted with the recent RTRIM changes (#7108).

#34 - 06/30/2023 03:32 AM - Alexandru Lungu

- Status changed from WIP to Review

- % Done changed from 70 to 100

This will go in pending for a while. Eric is already doing some parallel work on lazy hydration so we may need to synchronize on this. I can retest the FWD changes here, but I am not quite aware of how we will integrate later, considering the current low performance. The FWD-H2 changes were good and got them integrated already.

Conclusion: I will do a last profiling on this with my latest testing environment. If this is still slow, we shall either do some tracking to check where the time is spent **or** reject this.

#35 - 07/20/2023 08:16 AM - Alexandru Lungu

Rebased 7185a to latest trunk. Branch is now at rev. 14671.
Regression testing passes successfully.
Started profiling test. This round will be decisive for 7185a.

#36 - 07/21/2023 02:39 AM - Alexandru Lungu

- Status changed from Review to Test

I've got a +0.36% performance decrease, so the changes in 7185a can be dropped. At this point, the H2 modifications from 7185a_h2 are inside trunk and they provided a consistent improvement overall, so this task can halt at this point. If ww see changes into the hydration at any point, we might want to reconsider the discussion here, but maybe with another approach. This can be rejected.