Database - Feature #6582

implement multi-table AdaptiveQuery

07/05/2022 12:46 PM - Eric Faulhaber

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Alexandru Lungu	**% Done:**	40%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**vendor_id:**	GCD

Description

Related issues:

Related to Database - Bug #5307: AdaptiveQuery goes into dynamic mode too agg...	**WIP**
Related to Database - Feature #6695: Multi-table preselect query may underper...	**WIP**
Related to Database - Bug #4931: possible ProgressiveResults performance impr...	**New**
Related to Database - Support #7058: get the FWD fork of the H2 code base und...	**WIP**
Related to Database - Feature #7061: Enable the use of lazy result sets in H2	**Closed**
Related to Database - Feature #7066: Implement multi-table indexed query in H2	**WIP**

History

#2 - 07/05/2022 01:01 PM - Eric Faulhaber

AdaptiveQuery currently is limited to a single table query, due primarily to the complexity of the state machine which manages the switches between preselect and dynamic mode (dynamic mode in this context refers to the record-by-record fetching/iteration of query results, not to a dynamically defined, runtime converted query; the latter came later and is a different animal). Due to this single-table limit, we have many more FOR EACH, {EACH|FIRST|LAST}, ... constructs converting to CompoundQuery than we would like. We want to eliminate as many cases of CompoundQuery use as possible, as this is the slowest possible way to manage a multi-table join.

The purpose of this task is to:

- define test cases which currently convert to CompoundQuery, but which could/should convert to AdaptiveQuery, if AdaptiveQuery supported multi-table joins;
- modify conversion to convert these test cases to AdaptiveQuery, instead of CompoundQuery;
- modify the runtime to expand (and hopefully refactor and simplify the overall design of) the state machine which manages the switches between preselect and dynamic mode, to handle these switches seamlessly for multi-table join cases.

#3 - 07/12/2022 09:24 AM - Eric Faulhaber

- Assignee set to Alexandru Lungu

Alexandru, please review the existing AdaptiveQuery implementation and consider how we might go about extending/reworking this for multiple tables.

#4 - 07/12/2022 09:26 AM - Alexandru Lungu

- Status changed from New to WIP

Eric Faulhaber wrote:

> Alexandru, please review the existing AdaptiveQuery implementation and consider how we might go about extending/reworking this for multiple tables.

Sure!

#5 - 07/22/2022 10:07 AM - Eric Faulhaber

Alexandru Lungu wrote:

> Eric Faulhaber wrote:
>
> > Alexandru, please review the existing AdaptiveQuery implementation and consider how we might go about extending/reworking this for multiple tables.
>
> Sure!

Any thoughts so far?

#6 - 07/22/2022 10:41 AM - Alexandru Lungu

- % Done changed from 0 to 20

I got the time to familiarize myself with AdaptiveQuery. I understood the motivation for the "adaptive" part, but now I have a clear view of what the low-level implications are (invalidComponent, breakValue, AdaptiveQuery$DynamicResults, invalidate, validate, etc.). I also debugged out some examples to understand when does the invalidation/re-validation occur in the Java code.

Further, I started some basic investigation into how the multi-table scenario can be integrated. I see that there is already some trivial multi-table support for AdaptiveQuery (if there are multiple tables, use a CompoudQuery instead of RAQ as dynamic back-end), but this is only for Preselect -> Dynamic transition. Also, this is not exposed, as multi-table queries do not convert to AdaptiveQuery. At this stage I am not even sure if AdaptiveQuery listens for the invalidation of components, so this existing support may be improved.

The problem of course is the backward transition (Dynamic -> Preselect). At this very start, I can tell that it is possible. It is correct to work around the first query in the query composition. If the first query got over the "sort band", so it reverts to preselect mode, then the parent AdaptiveQuery can revert to preselection as-well (as the underlying sub-queries should restart their search anyways). Also, if a component is in dynamic mode, moving to the next element in the first query will also cause a revalidation. Of course, this is a trivial strategy and can be further optimized. Also, I didn't consider yet other aspects as: FIRST/LAST or OUTER-JOIN.

I was planning to dedicate this weekend to some "drawing on paper" kind of work to have a solid plan of implementing an efficient multi-table AdaptiveQuery. From my point of view, there is no need of "reworking", but only extending this feature such that AdaptiveQuery is sensitive with its

components.

#7 - 07/22/2022 10:54 AM - Eric Faulhaber

- Related to Bug #5307: AdaptiveQuery goes into dynamic mode too aggressively added

#8 - 07/22/2022 11:46 AM - Eric Faulhaber

Alexandru Lungu wrote:

> I got the time to familiarize myself with AdaptiveQuery. I understood the motivation for the "adaptive" part, but now I have a clear view of what the low-level implications are (invalidComponent, breakValue, AdaptiveQuery$DynamicResults, invalidate, validate, etc.). I also debugged out some examples to understand when does the invalidation/re-validation occur in the Java code.

The ideal scenario is that we start in preselect (fast) mode and we only ever transition to dynamic (slow) mode when some change to the index we are walking (at a position we have not yet reached in our walk), forces that transition. However, this is not currently working correctly in all cases (see #5307). I only mention this so that as you dig into the code and debug, you are aware that the implementation may not perfectly match the intended design currently.

> Further, I started some basic investigation into how the multi-table scenario can be integrated. I see that there is already some trivial multi-table support for AdaptiveQuery (if there are multiple tables, use a CompoudQuery instead of RAQ as dynamic back-end), but this is only for Preselect -> Dynamic transition. Also, this is not exposed, as multi-table queries do not convert to AdaptiveQuery. At this stage I am not even sure if AdaptiveQuery listens for the invalidation of components, so this existing support may be improved.

Please bear in mind that the existing multi-table code in AdaptiveQuery was meant to be very limited in scope to solve a specific problem; it was not necessarily meant to be a model for proper, multi-table support. It is a bit of a hack that I bolted on to support the CompoundQuery optimizer, where we attempt at runtime to morph as many CompoundQuery component queries into server-side joins as possible, each running as AdaptiveQuery components which replace two or more original components in the pre-optimized CompoundQuery. These adaptive queries start in preselect mode, but I needed a way to fall back to dynamic mode, in the event it was necessary. IIRC, I didn't provide a way back to preselect mode in this limited use case, as it was too complicated at the time.

> The problem of course is the backward transition (Dynamic -> Preselect). At this very start, I can tell that it is possible. It is correct to work around the first query in the query composition. If the first query got over the "sort band", so it reverts to preselect mode, then the parent AdaptiveQuery can revert to preselection as-well (as the underlying sub-queries should restart their search anyways). Also, if a component is in dynamic mode, moving to the next element in the first query will also cause a revalidation. Of course, this is a trivial strategy and can be further optimized. Also, I didn't consider yet other aspects as: FIRST/LAST or OUTER-JOIN.
>
> I was planning to dedicate this weekend to some "drawing on paper" kind of work to have a solid plan of implementing an efficient multi-table AdaptiveQuery. From my point of view, there is no need of "reworking", but only extending this feature such that AdaptiveQuery is sensitive with its components.

Great! I'm excited to see what you come up with to evolve the adaptive query concept to support server-side joins.

#9 - 07/23/2022 07:44 AM - Igor Skornyakov

Alexandru Lungu wrote:

> Also, I didn't consider yet other aspects as: FIRST/LAST or OUTER-JOIN.

I can suggest the following trick for FIRST/LAST.
Consider the following 4GL query:

```
FOR EACH T FIELDS(<field list T>),
   LAST C FIELDS(<field list C>) BY <field list C0>
   WHERE <condition depending on <field list C1>>
```

The corresponding SQL query can look like this:

```
WITH cte AS (
   SELECT row_number() over(order by <field list C0>) as rn, C.*
   FROM C
)
SELECT <field list T>, <field list C>
FROM T
  JOIN (SELECT MAX(rn) AS maxrn,  <field list C1> from cte GROUP BY <field list C1>) ctemax ON <condition depe
nding on <field list C1>>
  JOIN cte ON cte.rn = ctemax.maxrn
```

In case of OUTER JOIN the second JOIN will be OUTER.

#10 - 07/23/2022 01:50 PM - Igor Skornyakov

As far as I understand the approach described in #6582-9 works with MariaDB 10.2+ as well.

#11 - 07/24/2022 06:06 AM - Alexandru Lungu

Eric Faulhaber wrote:

> The ideal scenario is that we start in preselect (fast) mode and we only ever transition to dynamic (slow) mode when some change to the index we are walking (at a position we have not yet reached in our walk), forces that transition. However, this is not currently working correctly in all

> cases (see #5307). I only mention this so that as you dig into the code and debug, you are aware that the implementation may not perfectly match the intended design currently.

Good to know; I will focus a bit more on debugging some test-cases which do not match the intended AdaptiveQuery design. I tried out #5307 and it looks like it is fixed (the converted AdaptiveQuery is no longer invalidated).

> Please bear in mind that the existing multi-table code in AdaptiveQuery was meant to be very limited in scope to solve a specific problem; it was not necessarily meant to be a model for proper, multi-table support. It is a bit of a hack that I bolted on to support the CompoundQuery optimizer, where we attempt at runtime to morph as many CompoundQuery component queries into server-side joins as possible, each running as AdaptiveQuery components which replace two or more original components in the pre-optimized CompoundQuery. These adaptive queries start in preselect mode, but I needed a way to fall back to dynamic mode, in the event it was necessary. IIRC, I didn't provide a way back to preselect mode in this limited use case, as it was too complicated at the time.

A very "optimized" compound query (such that all components are joined into a single fat one) will have a single underlying query and that is an AdaptiveQuery. This means that converting to an AdaptiveQuery directly or enhancing the current CompoundQuery to better combine its components (and ultimately resolving to one single AdaptiveQuery) should have the same semantics. For performance, converting to AdaptiveQuery is cleaner, more flexible and avoids the overhead of run-time optimization stage. In any case, the fallback to preselect mode is needed.

Igor Skornyakov wrote:

> I can suggest the following trick for FIRST/LAST.
> Consider the following 4GL query:

I was rather thinking if there are implications of FIRST/LAST and OUTER-JOIN when doing invalidation/revalidation of multi-table AdaptiveQuery. The issue provided by Eric (#5407) seems to be a starting point for investigating FIRST and LAST in the context of AdaptiveQuery (in this case, the multi-table scenarios). OUTER-JOIN in the context of AdaptiveQuery is still to be investigated.

The HQL assembling itself (or even SQL generation) is rather related to the PreselectQuery mode of the AdaptiveQuery. AFAIK, FIRST and LAST already have support for preselect queries. OUTER JOIN is work in progress. Please consider #6196-15 where there are multiple ideas for implementing OUTER-JOIN for preselection (the first solution there seems the most suitable).

The provided query is reliable, but there are some aspects to consider. I also faced them when searching for SQL solutions too in #6196-15.

- field referencing becomes complex. <condition depending on <field list C1>> should be rewritten to "requantify" the C fields as ctemax fields. This should also be done for other WHERE clauses of further JOIN clauses which depend upon C fields. For example, <field list C> is in fact a rewritten list of C quantified with ctemax.
- the implementation of PreselectQuery doesn't use ON keyword right now; all joins are cross-joins which are filtered out by a single WHERE clause at the end of the SQL statement. This is not that concerning I guess, as semantically the 4GL WHERE should be the same as the SQL ON (please correct me if I am wrong).
- in MAX(rn) AS maxrn [...] from C the rn field can't be found (is it part of cte which is not yet joined?). Should the sub-select use cte instead of C? If yes, AFAIK there is no proper way to reference the cte fields selected with * (for instance C.field1 is valid, but cte.field1 is invalid as CTE doesn't store the field names of *).

#12 - 07/24/2022 01:28 PM - Igor Skornyakov

Alexandru Lungu wrote:

> I was rather thinking if there are implications of FIRST/LAST and OUTER-JOIN when doing invalidation/revalidation of multi-table AdaptiveQuery. The issue provided by Eric (#5407) seems to be a starting point for investigating FIRST and LAST in the context of AdaptiveQuery (in this case, the multi-table scenarios). OUTER-JOIN in the context of AdaptiveQuery is still to be investigated.
>
> The HQL assembling itself (or even SQL generation) is rather related to the PreselectQuery mode of the AdaptiveQuery. AFAIK, FIRST and LAST already have support for preselect queries. OUTER JOIN is work in progress. Please consider #6196-15 where there are multiple ideas for implementing OUTER-JOIN for preselection (the first solution there seems the most suitable).

I was thinking about optimizing CompoundQuery which now implements multi-table join in the Java code. I've not noticed at least LAST support in the SQL generation (the FIRST support is trivial for a single table query).

> The provided query is reliable, but there are some aspects to consider. I also faced them when searching for SQL solutions too in #6196-15.
>
> - field referencing becomes complex. <condition depending on <field list C1>> should be rewritten to "requantify" the C fields as ctemax fields. This should also be done for other WHERE clauses of further JOIN clauses which depend upon C fields. For example, <field list C> is in fact a rewritten list of C quantified with ctemax.

Sure, but, based on my experience it no a big deal. I've implemented a similar logic for the word tables' support.

- the implementation of PreselectQuery doesn't use ON keyword right now; all joins are cross-joins which are filtered out by a single WHERE clause at the end of the SQL statement. This is not that concerning I guess, as semantically the 4GL WHERE should be the same as the SQL ON (please correct me if I am wrong).

Logically it is possible of course not to use ON clause for JOIN (but **not** for LEFT JOIN) and move the corresponding predicate(s) (after possible 'requantifying') to the WHERE part. However, I think that it makes the query structure less comprehandable and (possibly) makes the job of the query analyzer more complicated.

- in MAX(rn) AS maxrn [...] from C the rn field can't be found (is it part of cte which is not yet joined?). Should the sub-select use cte instead of C? If yes, AFAIK there is no proper way to reference the cte fields selected with * (for instance C.field1 is valid, but cte.field1 is invalid as CTE doesn't store the field names of *).

Yes, it was a typo, sorry. Fixed now.

#13 - 07/25/2022 03:38 AM - Alexandru Lungu

I've done a bit of query exploring. I converted the following example:

```
def temp-table tt field f1 as integer
                  index idx1 f1.
def temp-table tt2 field f2 as integer
                  index idx2 f2.

def query q for tt, tt2.
open query q for each tt, last tt2.
/* get next q. -> the retrieval mode is not relevant */
```

The generated CompoundQuery has two components: AdaptiveQuery and RandomAccessQuery.

```
query0.assign(new CompoundQuery().initialize(true, false));
query0.addComponent(new AdaptiveQuery().initialize(tt, ((String) null), null, "tt.f1 asc", "WHOLE-INDEX,idx1")
);
query0.addComponent(new RandomAccessQuery().initialize(tt2, ((String) null), null, "tt2.f2 asc", "WHOLE-INDEX,
idx2"), QueryConstants.LAST);
```

When retrieving a record, CompoundQuery$Optimizer.isServerJoinPossible is used and tells that a server join is in fact possible. The back-end generates an AdaptiveQuery with two components. Initially, it is preselect mode, so it is able to generate a multi-table FQL which looks like:

```
from Tt_1_1__Impl__ as tt,
     Tt2_1_1__Impl__ as tt2
where (tt._multiplex = ?0) and
     ((tt2._multiplex = ?1) and
      (tt2.recid = (select tt2_1.recid from Tt2_1_1__Impl__ as tt2_1 order by tt2_1.f2 desc, tt2_1.recid desc
single)))
order by tt._multiplex asc, tt.f1 asc, tt.recid asc
```

The LAST retrieval is done similar to FIRST, but with inverted sort clause (tt2_1.f2 desc, tt2_1.recid desc). If I use first instead of last, the FQL ends with tt2_1.f2 asc, tt2_1.recid asc single.

In this example, the run-time generates a "query chain": CompoundQuery delegates an AdaptiveQuery in preselect mode (using PreselectQuery).

Finally, I surely understand your point of having database-join support for CompoundQuery as well. Even in my example, if I was to change any of the buffers, the AdaptiveQuery may invalidate and fallback to an un-optimizable dynamic retrieval (CompoundQuery). I agree this should be the next topic to handle after the current thread, regarding multi-table AdaptiveQuery.

#14 - 07/25/2022 03:56 AM - Igor Skornyakov

Alexandru Lungu wrote:

> I've done a bit of query exploring. I converted the following example:
>
> [...]
>
> The generated CompoundQuery has two components: AdaptiveQuery and RandomAccessQuery.
>
> [...]
>
> When retrieving a record, CompoundQuery$Optimizer.isServerJoinPossible is used and tells that a server join is in fact possible. The back-end generates an AdaptiveQuery with two components. Initially, it is preselect mode, so it is able to generate a multi-table FQL which looks like:
>
> [...]
>
> The LAST retrieval is done similar to FIRST, but with inverted sort clause (tt2_1.f2 desc, tt2_1.recid desc). If I use first instead of last, the FQL ends with tt2_1.f2 asc, tt2_1.recid asc single.
>
> In this example, the run-time generates a "query chain": CompoundQuery delegates an AdaptiveQuery in preselect mode (using PreselectQuery).
>
> Finally, I surely understand your point of having database-join support for CompoundQuery as well. Even in my example, if I was to change any of the buffers, the AdaptiveQuery may invalidate and fallback to an un-optimizable dynamic retrieval (CompoundQuery). I agree this should be the next topic to handle after the current thread, regarding multi-table AdaptiveQuery.

Indeed, my trick is just a "shooting from a cannon to sparrows". It is sufficient (for PostgreSQL) to use LIMIT 1/FETCH FIRST 1 with a proper ORDER BY clause (inverted for LAST)
It is strange that I've not seen the results of the optimization in my debugging sessions.

#15 - 08/15/2022 10:09 AM - Alexandru Lungu

I created 6582a as a branch of 3821c. I committed (rev. 14159) some conversion related work and minimal runtime changes to support multi-table adaptive query.

Right now, I have some of my examples working (from conversion point of view), but I still have some not yet compiling properly - I will fix them ASAP. Afterwards, I will do some extra tests for can-find as I see that there is a really consistent logic in the rules and I am not yet sure if my changes are totally safe in regard to can-find.

Initially, I expect to see the AdaptiveQuery working out-of-the-box with the current implementation with multi-table (already in place from the compound query optimization). Afterwards, I will get into the optimization there.

Finally, my solution is mostly working around OPEN QUERY, FOR, REPEAT and DO database queries which are dynamic, multi-table, without outer-join and without presort. There are now converting to CompoundQuery and I will replace them with AdaptiveQuery.

#16 - 08/17/2022 03:45 AM - Alexandru Lungu

Committed 6582a/rev. 14160 including some fixes for order by clause and iteration type in multi-table AdaptiveQuery. can-find is working alright. Hotel GUI converts successfully.

- some of open query and for are no longer converted to CompoundQuery, but to AdaptiveQuery
- this happens in specific cases: more than 1 table and not in preselect mode, no outer-join, no presort and contiguous.

I will start the functional tests for a basic multi-table AdaptiveQuery test. I will do a very short initial profiling of the performance of the current implementation vs multi-table AdaptiveQuery. Afterwards, I will go ahead with optimizations of the AdaptiveQuery.

#17 - 08/19/2022 10:36 AM - Eric Faulhaber

Alexandru Lungu wrote:

> Initially, I expect to see the AdaptiveQuery working out-of-the-box with the current implementation with multi-table (already in place from the compound query optimization). Afterwards, I will get into the optimization there.

By this, I understand you to mean that the transition from preselect to dynamic mode initially uses a CompoundQuery for dynamic mode. Is that correct? That should already offer an improvement over what we have today, especially if a query can stay in preselect mode through the entire result set iteration.

Once you have the basics working well, I am interested to see your ideas about the more optimized transitions:

- what it looks like going from preselect to dynamic mode;
- how the query operates while in dynamic mode; and
- how/when we transition back to preselect mode.

...Or perhaps you have some completely different ideas that don't exactly follow this model.

#18 - 08/20/2022 07:45 AM - Alexandru Lungu

Eric Faulhaber wrote:

> Alexandru Lungu wrote:
>
> > Initially, I expect to see the AdaptiveQuery working out-of-the-box with the current implementation with multi-table (already in place from the compound query optimization). Afterwards, I will get into the optimization there.

> By this, I understand you to mean that the transition from preselect to dynamic mode initially uses a CompoundQuery for dynamic mode. Is that correct?

Exactly. The AdaptiveQuery is always starting as a PreselectQuery and unless it detects a change in the indexed fields, it will stay in preselect mode until the end. It uses CompoundQuery only when the dynamic mode is required. Right now there is no "going back" to the preselect mode implemented for multi-table AdaptiveQuery.

> That should already offer an improvement over what we have today, especially if a query can stay in preselect mode through the entire result set iteration.

I am facing a three times slower query in AdaptiveQuery comparing to CompoundQuery (a read-only example). This may be due to some implementation inside AdaptiveQuery which "wasn't ready" for multi-table scenarios. I am going to ensure that AdaptiveQuery meets our time expectation right now.

> Once you have the basics working well, I am interested to see your ideas about the more optimized transitions:
>
> what it looks like going from preselect to dynamic mode;
> how the query operates while in dynamic mode; and
> how/when we transition back to preselect mode.
> ...Or perhaps you have some completely different ideas that don't exactly follow this model.

Only the OPEN QUERY or FOR with implicit sort or explicit sort using first-table fields convert to AdaptiveQuery. The others use PreselectQuery. Therefore:

- in my mind, the "dynamic mode" is still initially represented by the CompundQuery back-end. To benefit most from a multi-component AdaptiveQuery, I am thinking of building a single SQL which can dynamically retrieve records from the database. This can be done using the activeBundle method from RandomAccessQuery. That is, we can do a smart joining of the conditions from activeBundle to get a single SQL. This however will work under the specified conditions: no outer-join, no cross-database, no client-side where. I may also delay the integration of FIRST and LAST for now in order to keep things simple at this stage. I need to come up with some concrete examples on this one. Enqueuing this - preparing some insightful examples!
- initially, the switch between back-ends (preselect, compound) is ideal as we benefit much from the existing implementaion. We can presume that some changes to the buffers trigger dynamic mode. The sortBand is only on the first table and once the sortBand is reached by the first joined component, preselection is reused. However, once the "dynamic mode" is implemented using a single "fat" joining SQL (previous bullet), there is no need to make a distinction between preselect and dynamic. In fact, the preselect mode means that we work in a case requiring a "slim" joining SQL very much alike the one generated by PreselectQuery. What are your thoughts on this?
- the only thing concerning here is the result container used. I think we can still use ProgressiveResults, but I need to put some thought on this.

#19 - 08/22/2022 10:38 AM - Alexandru Lungu

> I am facing a three times slower query in AdaptiveQuery comparing to CompoundQuery (a read-only example). This may be due to some implementation inside AdaptiveQuery which "wasn't ready" for multi-table scenarios. I am going to ensure that AdaptiveQuery meets our time expectation right now.

I've done a comparison on a very simple 4GL test with all "multi-table" queries we have available right now:

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

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

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

For our current PreselectQuery implementation, this takes around 61s seconds.
For our current CompoundQuery implementation, this takes around 20s seconds.
For our freshly added AdaptiveQuery implementation, this takes around 62s seconds. The query is in preselect mode for the entire program, that is why it is similar to PreselectQuery.

This is a concerning case in which CompoundQuery works faster than PreselectQuery. I can't tell by debugging if there is any precise bottlneck code in PreselectQuery comparing to CompoundQuery. As a matter of fact, Apache JMeter shows that the single joined query (from PreselectQuery) works much faster than the one-by-one query (from CompoundQuery).

However, there seems to be a massive overhead in the hydratation/fetching process itself. The preselect query happens to fetch a record way slower than a compound query (e.g. BaseRecord.readProperty works in total 26s-preselect and 10s-compound). This is most probably because the preselect query has to gather the same row fragment multiple times. In the example above:

- preselect: a total number of 250,000 of tt and 250,000 of tt2 are retrieved. One single select SQL statement is used.
- compound: a total number of 5,000 of tt and 250,000 of tt2 are retrieved. 5000 select SQL statements are used.

I guess that 245,000 extra tt retrieved makes the preselect query slower even if the number of SQL statements is lower. Note that this issue is not related to multi-table AdaptiveQuery, but to the fact that there are cases in which dynamic mode works faster.

#20 - 08/23/2022 11:10 AM - Eric Faulhaber

Alexandru Lungu wrote:

> I am facing a three times slower query in AdaptiveQuery comparing to CompoundQuery (a read-only example). This may be due to some implementation inside AdaptiveQuery which "wasn't ready" for multi-table scenarios. I am going to ensure that AdaptiveQuery meets our time expectation right now.

I've done a comparison on a very simple 4GL test with all "multi-table" queries we have available right now:

[...]

For our current PreselectQuery implementation, this takes around 61s seconds.
For our current CompoundQuery implementation, this takes around 20s seconds.
For our freshly added AdaptiveQuery implementation, this takes around 62s seconds. The query is in preselect mode for the entire program, that is why it is similar to PreselectQuery.

This is a concerning case in which CompoundQuery works faster than PreselectQuery. I can't tell by debugging if there is any precise bottlneck code in PreselectQuery comparing to CompoundQuery. As a matter of fact, Apache JMeter shows that the single joined query (from PreselectQuery) works much faster than the one-by-one query (from CompoundQuery).

However, there seems to be a massive overhead in the hydratation/fetching process itself. The preselect query happens to fetch a record way slower than a compound query (e.g. BaseRecord.readProperty works in total 26s-preselect and 10s-compound). This is most probably because the preselect query has to gather the same row fragment multiple times. In the example above:

- preselect: a total number of 250,000 of tt and 250,000 of tt2 are retrieved. One single select SQL statement is used.
- compound: a total number of 5,000 of tt and 250,000 of tt2 are retrieved. 5000 select SQL statements are used.

I guess that 245,000 extra tt retrieved makes the preselect query slower even if the number of SQL statements is lower. Note that this issue is not related to multi-table AdaptiveQuery, but to the fact that there are cases in which dynamic mode works faster.

Some very interesting findings!

I should note that in a real application, it would be very unusual to have a wide open cross join like this, which produces the cartesian product of all records across multiple tables.

We have this code in SQLQuery.hydrateRecord, before looking to the ResultSet for data:

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

This code should save us from re-hydrating the same record over and over for your test, even though the number of records you create (10,000) far exceeds the initial session cache size of 1,024. Nevertheless, there is a good deal of code leading up to this point, and calling it hundreds of thousands or millions of times (as in an open cross join) will add up. CompoundQuery will only invoke this code once per record in table tt, while the PreselectQuery walking the result set will encounter the same tt record 5,000 times per tt2 record in the cross joined result set, and will thus hit this code a lot more.

It would be interesting to see how the same test performs with a persistent table with a PostgreSQL back-end. It also would be interesting to have a more realistic test with a where clause that restricts the join somewhat (perhaps with multiple tt2 records per tt record, but not 5,000 each). I expect that the more times the tt record appears in the result set, the more times we hit the hydrate code, and the more performance we give up, even if we are short-circuiting the actual hydration with the session cache check above.

It should also be noted that, even though the session cache is by default only 1,024 in size, it will "stretch" temporarily to become larger than that, if we are tracking the DMOs for UNDO processing. That will be the case for all persistent table DMOs and for all DMOs for temp-tables which are NOT marked NO-UNDO.

#21 - 08/23/2022 12:19 PM - Alexandru Lungu

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

#22 - 08/23/2022 12:34 PM - Alexandru Lungu

> I should note that in a real application, it would be very unusual to have a wide open cross join like this, which produces the cartesian product of all records across multiple tables.
> It would be interesting to see how the same test performs with a persistent table with a PostgreSQL back-end. It also would be interesting to have a more realistic test with a where clause that restricts the join somewhat (perhaps with multiple tt2 records per tt record, but not 5,000 each). I expect that the more times the tt record appears in the result set, the more times we hit the hydrate code, and the more performance we give up, even if we are short-circuiting the actual hydration with the session cache check above.

I agree, the persistent database may be the one which faces this situation more often. Also, a more appropriate testcase would be one with some where clauses indeed. To make a summary, the more records are found in the second table for each tt, the more extra fetches are done for tt (unless cached). Further, I am curious to see if there are joins in large 4GL applications that are closer to "many-to-many" than to "many-to-one". The latter is more concerning.

I created #6695 to discuss this issue further.

By now, I moved forward with the multi-table AdaptiveQuery stability. I've done some extra tests for: where clause, client where clause, locks and outer join. Some minor fixed were added to 6582a rev. 14161 and rev. 14162. I am facing right now some incorrect results from multi-table AdaptiveQuery in dynamic mode (a joined table is reiterated when updating a record). Working on this right now. Note that by default, each AdaptiveQuery component becomes to a AdaptiveQuery when delegating to CompoundQuery. This can/should be improved such that adaptive components can also become PreselectQuery or RandomAccessQuery.

#23 - 08/25/2022 03:49 AM - Alexandru Lungu

- File queries.zip added

I attached here the query conversion tests I used for reference. These are procedures which have structures involving queries.

#24 - 09/06/2022 11:26 AM - Alexandru Lungu

Alexandru Lungu wrote:

```
in my mind, the "dynamic mode" is still initially represented by the CompundQuery back-end.
```

This seemed to be the easy part back then, but now it seems that having a CompoundQuery as back-end is not that straight-forward when dealing with multi-table AdaptiveQuery. If the query goes into dynamic mode at some point (lets say X), a CompoundQuery should be built - a compound component should be generated for each adaptive component used. It wasn't obvious to me at the beginning that these adaptive components are stateful, so the compound components should be aware of the state (X) from which the AdaptiveQuery switched to dynamic mode. The old implementation naively resets the components and basically restarts the iteration.

The basic idea I adopted was to convert each adaptive component to a RandomAccessQuery when switching to dynamic mode.
What if we can use AdaptiveQuery component with breakValue - to be investigated.
RandomAccessQuery is able to start off from where AdaptiveQuery left in preselect mode. I committed this to 6582a/rev. 14163. My test is now providing the right results (it switches to dynamic and iterates correctly).

The status right now: we have a working prototype of multi-table AdaptiveQuery which doesn't switch back to preselect (yet). I will stress the current implementation a bit with various preselect-dynamic switches. My next move is to implement a mean of switching back to preselect (using breakValue on the outer-most component).

#25 - 09/09/2022 10:17 AM - Alexandru Lungu

I started using the tests from uast/adaptive_scrolling to test the changes for multi-table AdaptiveQuery. Almost all use multiple tables (2 or 3) and stress the preselect-dynamic switch very well (multiple buffers, buffer changes, where clauses, etc.).

There were many regressions initially which I fixed including:

- incorrect use of buffer snapshots when switching to dynamic mode.
- AdaptiveQuery re-fetching records which were already fetched by the delegated dynamic query.
- RandomAccessQuery not using the proper parameter filtering; field references were resolved to buffer current value instead of snapshot.
- Whole multi-table query invalidation was done on the last component instead of the first component.
- Multi-table AdaptiveQuery was not inlining the field references into the where clause when in preselect mode.

The following is the summary on 68 non-scrolling tests, out of which only 63 are compiling in 4GL. ~~Also, there are still 14 in the test-set left to investigate~~ Tested all non-scrolling tests which compile in 4GL:

- 51 tests are working OK both on 6582a and 3821c
- 10 tests are not working neither in 6582 nor in 3821c (latent issues)
- ~~03 tests are returning more or less records in 6582a (regressions)~~ fixed invalidation of AdaptiveQuery such that multiple components can be invalidated while doing the transition into dynamic mode + force invalidation when table field is used in inner where clause as snapshots should be used (available only in dynamic mode)
- ~~03 are throwing an error in 6582a (regressions)~~ fixed bug where the CompoundQuery delegate was doing next instead of iterate
- 02 are not working in 3821c, but are fixed in 6582a (fixes)

The issues right now are related to ProgressiveResults and I am not sure if they are latent or new regressions. Planning to finish this test-set and move on with the scrolling tests.

#26 - 09/13/2022 01:16 PM - Alexandru Lungu

- % Done changed from 20 to 30

The non-scrolling query examples that were working on 3821c are now working properly on 6582a as well. Note that I modified (stroke out some issues) #6582-25 to reflect the progress.
I am moving on with the scrolling query examples.

#27 - 09/14/2022 11:12 AM - Alexandru Lungu

- % Done changed from 30 to 20

I committed 6582a/rev. 14164 including all the latest changes done to increase stability of the multi-table AdaptiveQuery especially in regard to non-scrolling queries. I started running the test set from uast/adaptive_scrolling with scrolling queries, but only some seem to properly work. All 65 tests compile. ~~I tested only 25~~ I tested all of them:

- 55 tests are working OK both on 6582a and 3821c
- ~~36 tests are providing an incorrect list of results in 6582a, but work on 3821c (regressions)~~ only 05 failed tests are left
- ~~03 tests are throwing an error in 6582a (regressions)~~ only 02 failed tests are left
- 03 tests are not working neither in 6582a nor in 3821c (latent issues)

I need to take an in-depth look at the scrolling queries as they seem to work under different rules of invalidation. Planning to integrate the other tests and fix them.

#28 - 09/20/2022 11:29 AM - Eric Faulhaber

What can you see in terms of the impact of this refactoring on performance so far, if anything?

#29 - 09/21/2022 04:25 AM - Alexandru Lungu

I am close to an end with #6582-27. Only some use cases are left to clear off. Note that I updated #6582-27 to reflect the progress. The tests I use are very small and are not relevant for performance testing.

The only performance test I've done is the one from #6695. Of course, there can be similar test-cases (with WHERE, BY, etc.). However, I expect that the common bottleneck would be the **slow fetching**.
I will do a set of performance tests to check the performance improvement (I hope) by the end of the day.

#30 - 09/21/2022 11:10 AM - Alexandru Lungu

I've prepared a consistent set of tests, but it seems only some are actually finished in a reasonable time.

Test		4GL temporary		Temporary tables (H2)			Persistent tables (PostgreSQL)		
Scenario	Records	FOR	PRESELECT	Compound Query	PreselectQuery	AdaptiveQuery	Compound Query	PreselectQuery	AdaptiveQuery
2-table cross join	1000 * 1000	0 / 2258	977 / 1477	17 / 1115	497 / 562	14 / 4661	17 / 3347	1627 / 1293	19 / 4581
2-table cross join with by	1000 * 1000	1596 / 1665	1460 / 1544	2 / 759	374 / 372	388 / 420	3 / 3006	1399 / 1257	589 / 2531
2-table join first	1000 * 1	0 / 4	3 / 2	9 / 18	8 / 1	1 / 26	6 / 21	12 / 2	1 / 16
2-table join last with by	1000 * 1	4 / 2	4 / 1	2 / 8	5 / 1	7 / 1	2 / 10	4 / 1	4 / 3
2-table "inner" join	10 * 100	0 / 503	523 / 1	3 / 165	141 / 1	7 / 558	3 / 251	85 / 2	9 / 230
3-table "inner" join	10 * 100 * 10	0 / 1000	1070 / 16	4 / 165	178 / 10	1 / 284605	2 / 805	165 / 16	4 / 461

The tests were done with non-scrolling no-undo tables using etime. The cells use the format query open time / all records iteration time.
Right now, the times for multi-table AdaptiveQuery are not acceptable. I expected to see results rather similar to PreselectQuery, as the tests are not

switching to dynamic mode. The SQL generated are quite similar (between AdaptiveQuery and PreselectQuery), but there are some obvious differences:

- When using AdaptiveQuery, the order by clause is built differently: concatenate the sort clause of each adaptive component (to which we append the _multiplex). In PreselectQuery, the sort clause is inferred at the conversion time. The sort clause seems to match the indexes used by each table. I need to investigate further.
- When inlining referenceSubs (TableField is embedded into the FQL), some extra conditions are added: ? = tt2.f22 {tt.f1} is translated as tt.f1 = tt2.f22 or tt.f1 is null and tt2.f22 is null. These are not added into PreselectQuery where clause. Maybe we can restrict these only when outer-join is used? Further, I need to check if the or and and keywords are prioritized correctly when having more complex where clauses.
- AdaptiveQuery uses ProgressiveResults. The goal is to make the first retrievals as fast as possible (by limiting the result set). For example, 200 * 200 records are retrieved by executing 5 SQL queries with LIMIT/OFFSET. Side note: I encountered some more specific testcases in which each of these SQL queries were having the same execution time. This may be a false alarm, but I should clarify.

#31 - 09/22/2022 10:11 AM - Alexandru Lungu

I've done the same tests for persistent tables and updated #6582-30.
For persistent tables, multi-table AdaptiveQuery seems to do a better job as the backend (PostgreSQL) finds better plans and optimizes the joins very well. However, these tests does not have the _multiplex overhead. Sorting or filtering using _multiplex may be an issue when computing join plans in H2.

I focused on the temporary 3-table "inner" and all of the remarks from #6582-30 contribute to the slow-down:

> When using AdaptiveQuery, the order by clause is built differently: concatenate the sort clause of each adaptive component (to which we append the _multiplex). In PreselectQuery, the sort clause is inferred at the conversion time. The sort clause seems to match the indexes used by each table. I need to investigate further.

This is the only difference between AdaptiveQuery and PreselectQuery generated SQL it seems. Below is an example where the difference is clear:

```
order by tt._multiplex asc, tt.f1 asc nulls last, tt.recid asc,
         tt2._multiplex asc, tt2.f2 asc nulls last, tt2.recid asc,
     tt3._multiplex asc, tt3.f3 asc nulls last, tt3.recid asc -- AdaptiveQuery

order by tt._multiplex asc, tt.f1 asc nulls last, tt2.f2 asc nulls last, tt3.f3 asc nulls last -- PreselectQue
ry
```

There are several things to point out here:

- AdaptiveQuery generates a multiplex and recid for each component. The multiplex is generated to encourage the use of a pre-sorted index (by matching the order by clause with the index) and avoid sorting all records at the end. The recid field is mostly added for correctness as two similar records should be fetched base on the recid (is the current PreselectQuery incorrect from this point of view?)
- H2 allows the selection of an index to avoid sorting at the end. However, H2 supports only one single "sort index" and this should match the outer-most table selected by the plan. Unfortunately, our previous assumptions are not relevant in multi-table scenarios, neither for AdaptiveQuery nor for PreselectQuery
- It was hard o believe that solely the fact that there are 9 sort clauses instead of 4 makes the query so slow. After some investigation, the sort clause is taken into consideration for the query cost. Unfortunately, all 9 sort clauses from AdaptiveQuery contribute to a inefficient plan with reordered joins.

  > When inlining referenceSubs (TableField is embedded into the FQL), some extra conditions are added: ? = tt2.f22 {tt.f1} is translated as tt.f1 = tt2.f22 or tt.f1 is null and tt2.f22 is null. These are not added into PreselectQuery where clause. Maybe we can restrict these only when outer-join is used? Further, I need to check if the or and and keywords are prioritized correctly when having more complex where clauses.

Inlining referenceSubs is not revelant here as the pure PreselectQuery also generates tt.f1 = tt2.f22 or tt.f1 is null and tt2.f22 is null. It was nice to see that after removing tt.f1 is null and tt2.f22 is null, H2 selects a better plan by inferring some join keys from the where clause and boosts up the performance. Maybe we should think of a mean of rewriting such clause (eventually using IS NOT DISTINCT FROM). Note that even PostgreSQL prefers to sort the records at the end just because solving the or earlier seems a more appealing plan (no no sort-index is used).

> AdaptiveQuery uses ProgressiveResults. The goal is to make the first retrievals as fast as possible (by limiting the result set). For example, 200 *

200 records are retrieved by executing 5 SQL queries with LIMIT/OFFSET. Side note: I encountered some more specific testcases in which each of these SQL queries were having the same execution time. This may be a false alarm, but I should clarify.

If the query is sorted at the end (so no "sort index" is found), all records are retrieved anyways. Therefore, ProgressiveResults may determine a full-table scan even if it has limit 1 just because of order-by clause. Maybe we can detect such case before planning and avoid using ProgressiveResults and stick to ScrollableResults.

#32 - 09/23/2022 10:52 AM - Eric Faulhaber

- Related to Bug #4931: possible ProgressiveResults performance improvement added

#33 - 09/24/2022 05:30 AM - Alexandru Lungu

Iterate all records with NEXT												
Test		4GL temporary		Temporary tables (H2 fwd-h2/rev. 6)			Persistent tables (PostgreSQL 10)			Persistent tables (MariaDB 10.3.34)		
Scenario	Records	FOR	PRESELECT	Compound	Preselect	Adaptive	Compound	Preselect	Adaptive	Compound	Preselect	Adaptive
2-table cross join	1000 * 1000	0 / 2258	977 / 1477	19 / 1265	721 / 707	28 / 5358	16 / 3794	1332 / 1503	19 / 4628	15 / 5082	2665 / 1328	15 / 13132
2-table cross join with by	1000 * 1000	1596 / 1665	1460 / 1544	1 / 742	510 / 485	503 / 637	2 / 3569	1207 / 1367	632 / 2612	1 / 4523	2111 / 1131	2334 / 1232
2-table join first	1000 * 1	0 / 4	3 / 2	9 / 32	11 / 2	2 / 57	7 / 25	11 / 3	2 / 22	9 / 43	4 / 2	2 / 12
2-table join last with by	1000 * 1	4 / 2	4 / 1	2 / 9	10 / 1	12 / 1	2 / 12	5 / 2	5 / 5	1 / 9	4 / 2	5 / 2
2-table "inner" join	10 * 100	0 / 503	523 / 1	4 / 174	17 / 3	7 / 23	4 / 391	97 / 3	10 / 231	4 / 531	159 / 3	8 / 487
3-table "inner" join	10 * 100 * 10	0 / 1000	1070 / 16	5 / 200	148 / 16	4 / 494	3 / 861	113 / 19	4 / 435	2 / 1424	330 / 16	3 / 1208
range cross join	600 * 1000	0 / 1279	587 / 878	2 / 447	349 / 318	3 / 2870	2 / 1956	579 / 717	3 / 2243	1 / 2876	981 / 715	2 / 4842
fitler cross join	200 * 1000	0 / 476	196 / 296	1 / 153	121 / 93	1 / 880	1 / 710	302 / 225	1 / 665	1 / 863	207 / 224	1 / 749
range + fitler cross join	161000	0 / 370	157 / 234	1 / 111	104 / 78	2 / 835	1 / 570	200 / 311	1 / 572	2 / 768	167 / 213	1 / 604
3-table "inner" join first	1000	0 / 491	491 / 2	3 / 79	11 / 3	2 / 35	2 / 301	64 / 2	3 / 195	2 / 379	177 / 3	1 / 694
inner join with range	801	0 / 412	418 / 1	2 / 214	4 / 1	2 / 13	2 / 426	58 / 2	1 / 152	2 / 1028	98 / 1	1 / 391
inner join with filter	900	0 / 500	505 / 1	2 / 94	4 / 1	2 / 13	3 / 302	45 / 2	1 / 176	2 / 598	121 / 1	2 / 482
inner join	1000	0 / 6	5 / 2	2 / 30	4 / 2	1 / 15	2 / 442	47 / 3	2 / 185	2 / 908	121 / 2	3 / 477

with unique												
range inner join with unique	601	0 / 4	3 / 1	1 / 14	3 / 2	2 / 14	1 / 261	29 / 1	1 / 116	3 / 599	73 / 1	2 / 292

I committed 6582a/rev. 14165 including the new support for IS (NOT) DISTINCT FROM which allows the simplification of clauses like tt.f1 = tt2.f22 or tt.f1 is null and tt2.f22 is null. I repeated the tests and I've got the results from the table above. There is a clear improvement for PreselectQuery and AdaptiveQuery both for temporary and persistent tables. This is mainly because:

- H2 is considering tt.f1 IS NOT DISTINCT FROM tt2.f22 as a filter when joining tt with tt2 (even if the scan index is still used). This way, it eliminates a lot of records before moving one with the next operations like joining or sorting.
- PostgreSQL generates a new plan for tt.f1 IS NOT DISTINCT FROM tt2.f22. It doesn't prioritize the OR clause and it materializes faster. It also identifies the join as an "inner join" using this new clause.

The remaining bottlenecks are related to the sort clauses and ProgressiveQuery. The query can be discussed and optimized in #4931. For the sorting part, there is no clear solution yet. Even for simple queries, the scan index is used and the sorting is very slow. The example below is using 1000 tt and 1000 tt2 and one index per table (_multiplex, f1, recid and _multiplex, f2, recid). **The tests below are simulated using PostgreSQL 10; H2 is not providing such an in-depth analysis**

Query	Total time	Join time	Sort time	Description
`select * from tt cross join tt2 order by tt.multiplex asc, tt.f1 asc nulls last, tt.recid asc, tt2.multiplex asc, tt2.f2 asc nulls last, tt2.recid asc`	1.16s	0.14s	1.02s	This is the FWD generated query which confirms the limitations of query planning based on ORDER BY described in #6582-31. Neither tt nor tt2 are traversed using their index.
`select * from tt cross join tt2 order by tt.multiplex asc, tt.f1 asc nulls last, tt.recid asc`	0.13s	0.13s	0.00s	This is a hypothetical query which doesn't order by the second table. tt is traversed using an index; tt2 is scanned.
`select * from tt cross join (select * from tt2 order by tt2._multiplex asc, tt2.f2 asc nulls last, tt2.recid asc) t2 order by tt._multiplex asc, tt.f1 asc nulls last, tt.recid asc`	0.12s	0.12s	0.00s	This is a partial solution which retrieves the records in the correct order. tt and tt2 are traversed using an index. The down-side is that complex queries may produce unpredictable orders, as the planning may permute the join order.

I will do some tests with the latest supported PostgreSQL (version 14) and H2 (version 2.1.214) to check if the planning technique is considering the ORDER BY clause when choosing an index. Anyways, we should find a way to cut out order by clauses and use ProgressiveResults only if we end up sorting based exclusively on the outer-most table. Otherwise, the backing database will sort all records at the end, so LIMIT is not relevant.

#34 - 09/27/2022 09:45 AM - Alexandru Lungu

- % Done changed from 20 to 40

I updated the table from #6582-33 with 8 new multi-table queries which are often used in large applications:

- **range**: only a contiguous sub-sequence of records are joined. The range query is done on an indexed field (100 <= f1 and f1 <= 900).
- **filter**: a custom single-table constraint is used. The constrained uses a non-indexed non-unique field (f22 = 1 or f22 = 6).
- **unique**: a unique index is defined on the join key. This makes FWD to omit certain sort clauses (f1 = f2 and f2 is unique).

I've also redone all tests for MariaDB using 3821c and a manually rebased 6582a (also in #6582-33). The results from MariaDB are relatively similar to PostgreSQL in regard to the query type, but absolutely slower in general. Note that IS (NOT) DISTINCT FROM is not available in MariaDB so it is emitted conditionally based on the dialect (6582a/rev. 14167).

From the performance testing point of view, there are still some wider scenarios to explore:

- queries which are not completely iterated, but only some results are retrieved (first / 3% / 10%)
- queries which restart their iteration by calling first (based on a growing threshold)
- queries which use both next and prev
- scrolling queries which use reposition on cached records / on new records (these are not stable as part of the regressions in #6582-27)
- queries which are invalidated (after the first / 10% / 25% / 50% records) and are iterated until the end
- queries which are invalidated (after the first / 10% / 25% / 50% records) and switch back to preselect right after / at some moment before reaching the end (this is not yet implemented)

However, the biggest impact of the multi-table AdaptiveQuery was expected to be in regard to the preselect mode (tested here). Therefore, I will focus on improving AdaptiveQuery for the current tests before moving on. Most probably #4931 and #6695 will also help on this matter.

#35 - 09/28/2022 07:11 AM - Alexandru Lungu

Iterate all records with NEXT									
Test		4GL temporary		Persistent tables (PostgreSQL 10)			Persistent tables (PostgreSQL 14)		
Scenario	Records	FOR	PRESELECT	Compound	Preselect	Adaptive	Compound	Preselect	Adaptive
2-table cross join	1000 * 1000	0 / 2258	977 / 1477	16 / 3794	1332 / 1503	19 / 4628	15 / 3931	778 / 1219	17 / 2319
2-table cross join with by	1000 * 1000	1596 / 1665	1460 / 1544	2 / 3569	1207 / 1367	632 / 2612	2 / 3611	1087 / 1074	11 / 2841
2-table join first	1000 * 1	0 / 4	3 / 2	7 / 25	11 / 3	2 / 22	7 / 24	12 / 2	1 / 12
2-table join last with by	1000 * 1	4 / 2	4 / 1	2 / 12	5 / 2	5 / 5	2 / 8	5 / 2	4 / 4
2-table "inner" join	10 * 100	0 / 503	523 / 1	4 / 391	97 / 3	10 / 231	3 / 846	70 / 3	8 / 66

3-table "inner" join	10 * 100 * 10	0 / 1000	1070 / 16	3 / 861	113 / 19	4 / 435	2 / 1452	119 / 17	4 / 182
range cross join	600 * 1000	0 / 1279	587 / 878	2 / 1956	579 / 717	3 / 2243	2 / 2069	622 / 617	2 / 1288
fitler cross join	200 * 1000	0 / 476	196 / 296	1 / 710	302 / 225	1 / 665	1 / 725	155 / 226	1 / 397
range + fitler cross join	161000	0 / 370	157 / 234	1 / 570	200 / 311	2 / 572	2 / 553	116 / 178	1 / 383
3-table "inner" join first	1000	0 / 491	491 / 2	3 / 301	64 / 2	3 / 195	2 / 675	59 / 2	2 / 64
inner join with range	801	0 / 412	418 / 1	2 / 426	58 / 2	1 / 152	1 / 731	39 / 2	1 / 41
inner join with filter	900	0 / 500	505 / 1	3 / 302	45 / 2	1 / 176	2 / 683	45 / 2	1 / 47
inner join with unique	1000	0 / 6	5 / 2	2 / 442	47 / 3	2 / 185	3 / 159	47 / 2	2 / 74
range inner join with unique	601	0 / 4	3 / 1	1 / 261	29 / 1	1 / 116	1 / 113	46 / 1	1 / 59

> I will do some tests with the latest supported PostgreSQL (version 14) and H2 (version 2.1.214) to check if the planning technique is considering the ORDER BY clause when choosing an index.

I used the same test-set for PostgreSQL 14 and I got the following remarks:

- The cross join is improved for PreselectQuery and implicitly for AdaptiveQuery.
- ProgressiveResults performs better in PostgreSQL 14. Therefore, AdaptiveQuery is always better with PostgreSQL 14.
- CompoundQuery is slower with inner joins, but faster with unique. I can't explain right now why this behavior occurs.

I can state that the AdaptiveQuery in preselect mode is significantly faster in PostgreSQL 14 than CompoundQuery in either PostgreSQL 10 or 14 (at least for the the tests I've done). Also, the PreselectQuery is totally better in PostgreSQL 14.

I've done some research and the improvement is due to the following feature introduced with PostgreSQL 13: **Incremental Sorting**. This doesn't match our expectation of joining tables by their index without a final sort phase, but it greatly improves the performance of a final sort. This feature kicks in when the first joined table is iterated using an index, so the records are sorted. At the end, there is no need of a whole result-set sort, but of multiple bucket sorts. In other words, the records containing the same row from the first table are part of the same bucket. Sorting the buckets will result in a sorted result-set. **This also greatly improves the cases with LIMIT, as only some buckets needs to be sorted to satisfy the limit.** This is why ProgressiveResults is working faster and thus the AdaptiveQuery is improved. ScrollableResults is faster just because the final sort phase is faster and thus the PreselectQuery is also improved.

I need to investigate the performance degradation of CompoundQuery. This is important to understand for 6582a, as CompoundQuery is still used as dynamic fallback for AdaptiveQuery and for some particular queries (using outer-join).

#36 - 09/29/2022 06:07 AM - Alexandru Lungu

I've done some tests with H2 2.1.214 an there is no improvement with the query plan (especially for cross joins).

I've also done tests with ScrollableResults instead of ProgressiveResults for our current H2. As expected, the new times are better as they are similar to the ones generated by PreselectQuery. Even so, the cross joins are still slower than in CompoundQuery. Note that these numbers are for the "Iterate all records with NEXT" scenario; I expect worse results for scenarios retrieving less records or which invalidate (even for inner joins).

At this stage, I achieved better results with multi-table AdaptiveQuery only for persistent tables using PostgreSQL 14. While MariaDB may look promising as most of queries are faster with AdaptiveQuery, the temporary database is far from acceptable. This makes 6582a integration more specific:

- either the conversion should generate multi-table AdaptiveQuery only for persistent tables
- the AdaptiveQuery should be invalidated from the start if it works with temporary tables
- find means of optimizing: SQL generation (the current tests are really simple, can we really improve further?), H2 engine (not ideal as we are some versions behind the latest H2 and changes may be hard to maintain in the future) or FWD queries/results (maybe).

#37 - 09/29/2022 08:20 AM - Greg Shah

Are all of the tests written for this in 4GL code? Can the testing be easily replicated by others in the future? I'd prefer for both of these to be the case. The tests should be in the new testcases project.

#38 - 09/29/2022 08:24 AM - Greg Shah

> either the conversion should generate multi-table AdaptiveQuery only for persistent tables

I think we should be emitting an single "abstract" multi-table approach in converted code. The idea is that at runtime this can be mapped to the optimal query approach and we can change that by making runtime changes in FWD rather than requiring the code to be reconverted. Hard coding this decision at conversion time seems very limited.

#39 - 09/30/2022 04:42 AM - Alexandru Lungu

Greg Shah wrote:

> Are all of the tests written for this in 4GL code? Can the testing be easily replicated by others in the future? I'd prefer for both of these to be the case. The tests should be in the new testcases project.

Yes, they are profiled using etime. The tests are already checked in testcases uast/adaptive_scrolling/performance_tests. Anyways, some of these will be integrated with Danut effort in #6679 to integrate H2 profiler into the Hotel application.

> I think we should be emitting an single "abstract" multi-table approach in converted code. The idea is that at runtime this can be mapped to the optimal query approach and we can change that by making runtime changes in FWD rather than requiring the code to be reconverted. Hard coding this decision at conversion time seems very limited.

I agree, this is the way to go. I will try to check now if there are any further optimizations for temporary tables in the context of multi-table queries.

#40 - 10/05/2022 07:43 AM - Alexandru Lungu

In my effort to seek optimizing methods for our in-memory H2, I took some time to analyze some in-memory (eventually embedded) databases. HSQLDB is a predecessor of H2 (latest release on 18 July 2022). Derby is a common alternative for H2 developed by Apache (latest Java 8 compatible version is from 3 May 2018; I also particularly took a look into the latest version from 15 June 2022). Surprisingly, SQLite is allowing in-memory embedded databases (the JDBC is a wrapped over the C engine). After executing some SQL and analyzing/explaining for queries, I also computed some performance tests.

Compare in-memory databases																				
Test		4GL temporary	H2 PAGESTORE (fwd-h2/rev. 6)			H2 MVSTORE (2.1.214)			HSQLDB (2.7.0)			Derby (10.14.2)			SQLite (3.39.3)			DuckDB (0.5.1)		
Scenario	Records	FOR	Prepare	Iterate	Total	Prepare	Iterate	Total	Prepare	Iterate	Total	Prepare	Iterate	Total	Prepare	Iterate	Total	Prepare	Iterate	Total
2-table cross join	1000 * 1000	0 / 2258	3,091	468	3,559	3,637	476	4,113	5,600	301	5,901	17,895	3,433	21,328	655	2,109	2,774	2,902	440	3,342
2-table cross join with by	1000 * 1000	1596 / 1665	433	482	915	540	492	1,032	811	325	1,136	2,718	1,968	4,686	621	2,437	3,058	671	481	1,152
2-table join first	1000 * 1	0 / 4	10	2	12	10	1	11	14	0	14	17	7	24	0	4	4	43	0	43
2-table join last with by	1000 * 1	4 / 2	4	2	6	3	2	5	4	1	5	10	2	12	2	3	5	11	0	11
2-table "inner" join	10 * 100	0 / 503	11	1	12	11	1	12	531	0	531	34	2	36	2	121	123	18	0	18
3-table "inner" join	10 * 100 * 10	0 / 1000	429	7	436	1,158	7	1,165	1,065	6	1,065	121	16	137	49	256	305	62	6	68
range cross join	600 * 1000	0 / 1279	1,633	287	1,920	2,117	294	2,411	3,183	176	3,183	7,798	1,224	9,022	237	1,298	1,535	1,159	344	1,503
fitler cross join	200 * 1000	0 / 476	453	93	546	612	85	697	886	63	886	1,809	130	1,939	12	485	497	276	87	363
range + fitler	161000	0 / 370	389	73	462	505	76	581	754	55	754	1,487	112	1,599	12	395	407	234	77	311

cross join																			
3-table "inner" join first	1000	0 / 491	20	3	23	18	1	19	532	0	532	51	2	53	3	117	120	94	1	95
inner join with range	801	0 / 412	8	1	9	11	1	12	533	2	533	33	2	35	2	100	102	11	0	11
inner join with filter	900	0 / 500	11	2	13	11	1	12	544	0	544	36	1	37	3	127	130	12	0	12
inner join with unique	1000	0 / 6	11	1	12	12	1	13	566	0	566	12	11	23	1	4	5	9	0	9
range inner join with unique	601	0 / 4	8	1	9	7	1	8	347	0	347	12	15	27	35	35	70	9	0	9

For the tests, I extracted the SQLs used by each testcase from FWD and executed them through the JDBC of each database. The times are in milliseconds and do not include the creation, population and dropping of the tables. Moreover, they are computed as an average of 10 individual runs. The queries measured are generated by a ProgressiveResults of a multi-table AdaptiveQuery, so there are around 5 statements per test with LIMIT and OFFSET. The "Prepare" time is the total time of all Statement.executeQuery(). The "Iterate" time is computed by iterating all results from the ResultSet and accessing all columns from each row. Finally, I came up with the following insights:

- H2 looks like the "leader" in the performance tests (even in MVSTORE), although it still under-performs in certain situations.
  - H2 has a good balance between "Prepare" and "Iterate".
  - H2 doesn't use index for solving the ORDER BY in multi-table contexts. The LIMIT and OFFSET keyword are not optimizing the query.
  - H2 is fetching fast enough, considering that it is an embedded Java database.
- HSQLDB is very slow, especially for the last 5 tests (which are maybe the most important).
  - HSQLDB is the best when it comes to record fetching. This may be due to its nature of pre-fetching all results and thus being slow on the "Prepare" phase.
  - HSQLDB doesn't use index for solving the ORDER BY in multi-table contexts. The LIMIT and OFFSET keyword are not optimizing the query.
- Derby is providing a competitive performance, but it really lacks performance on CROSS joins.
  - Derby has a good balance between "Prepare" and "Iterate".
  - Derby doesn't use index for solving the ORDER BY in multi-table contexts. The LIMIT and OFFSET keyword are not optimizing the query.
  - There are a lot of dialect differences (FETCH FIRST x ROWS ONLY instead of LIMIT x, OFFSET x ROWS instead of OFFSET x, no support for IS NOT DISTINCT FROM).
  - Derby may show better performance with newer versions running on Java 17 or higher. However, analyzing queries with the latest version from console, Derby still doesn't use index for solving ORDER BY
- SQLite is providing decent times, although it is still drawn-back by its "wrapper" nature
  - SQLite is really powerful in terms of "Prepare" (consistently better than H2). However, SQLite is really slow in terms of records fetching.
  - SQLite is using an index for the first joined table as it is part of the ORDER BY. FWD greatly benefits from this behavior as LIMIT and OFFSET became relevant.
  - SQLite is fetching slowly because it should "move" the values from the C back-end to the Java connector. I didn't find a way yet to fine tune this.

This feature of "use index for ORDER BY" which I have found in PostgreSQL 14 seems to be in SQLite as well. PostgreSQL itself takes this even further and does an incremental sort at the end. But even so, it is clear that such feature may consistently increase "Prepare" even for H2. I will need to check further if such feature can be smoothly integrated into H2.

#41 - 10/10/2022 09:01 AM - Alexandru Lungu

I replaced H2 (fwd-h2/rev.6) with H2 PAGESTORE (fwd-h2/rev. 6) and H2 MVSTORE (2.1.214) in #6582-40. The previous results (which are now replaced) were for fwd-h2/rev.6 with MVSTORE (irrelevant for our testing). Other tables in this thread are still correct as they were computed with etime using the internal connection string of FWD which uses PAGESTORE.

- H2 MVSTORE (2.1.214) is working 10% faster than H2 MVSTORE (fwd-h2/rev.6). This is encouraging as we can expect to see further performance improvements into the MVSTORE.
- The difference between PAGESTORE and MVSTORE is consistent, in the sense that every testcase is executed faster with PAGESTORE (excluding the negligible differences).
- PAGESTORE and MVSTORE are iterating the results similarly fast. The main difference between the engines is in the "Prepare" phase.

I added DuckDB (v0.5.1) to #6582-40. This is a MIT database written in C++ and represents "an in-process SQL OLAP Database Management System". This can be very easily added as in-memory embedded database through jdbc:duckdb. I find it relatively popular and fast, but it is drawn back by its lack of experience (released in 2019) and usability (only hundreds of users according to GitHub).

- DuckDB has the same model as SQLite, being a wrapped database written in C++. However, DuckDB has a significantly better iteration time.
- DuckDB doesn't use index for solving ORDER BY. **However**, the planner combines ORDER BY and LIMIT into TOP, which partially "quickly" sorts the result set until first N are in place.
- DuckDB has a weak prepare time (comparing to SQLite), but it is really competitive with H2.
- DuckDB has a duck mascot because it is a very versatile animal that can fly, walk and swim :)

DuckDB proposes a straight-forward mean of improving LIMIT: by resolving it together with the ORDER BY. However, H2 seems to have the same mechanism implemented, but the following tests prove that it is not that powerful:

Compare progressive results						
Ofset = 0, Limit	H2 PAGESTORE (fwd-h2/rev. 6)	H2 MVSTORE (2.1.214)	HSQLDB	Derby	SQLite	DuckDB
1	442	521	829	2,776	0	46
25	512	518	809	2,731	0	53
577	404	510	827	3,590	0	52
13825	458	528	813	3,674	0	122
331777	466	550	780	3,745	10	758

It looks like Derby, SQLite and DuckDB show different times for each value of limit. In general, limit 1/25 are processed consistently faster than bigger limits. HSQLDB in particular has no sign of optimizing the LIMIT.
If this matter can be optimized to reach times similar to DuckDB for lower limits (without using index), H2 can achieve really good results for multi-table queries. Looking into it!

#42 - 10/26/2022 08:38 AM - Alexandru Lungu

This is the latest comparison of the scenarios in the context of the new H2 feature: incremental indexed sorting (6582a / rev. 14170). I uncommitted the progress done with H2 2.1.210 (locally backup) and added the feature as rev. 7.

Profile incremental indexed sorting				
Scenario	H2 rev. 6		H2 rev. 7	
	Prepare	Iterate	Prepare	Iterate
2-table cross join	18	4,759	15	4,780
2-table cross join with by	417	462	476	482
2-table join first	1	28	2	38
2-table join last with by	12	1	9	1
2-table "inner" join	7	18	7	13
3-table "inner" join	2	464	3	326
range cross join	2	2,212	3	926
filter cross join	2	630	0	235
range + filter cross join	0	535	1	200
3-table "inner" join first	1	25	1	18
inner join with range	1	11	2	12
inner join with filter	1	7	1	8
inner join with unique	2	7	1	9
range inner join with unique	1	6	2	6

I am planning to add these tests as part of the Hotel GUI performance tests.
Only some times improved due to the "non-deterministic" order of the joins selected by the H2 engine. The optimization kicks in only when the planned join order matches the original join order. A great improvement was for range cross join which cut its time in half.
Very important to note that retrieving only some records (10%) may prove very fast with H2 rev. 7 as it can resolve LIMIT after fetching some rows, not the whole result-set.

#43 - 11/24/2022 10:56 AM - Alexandru Lungu

CHECKPOINT

I am pending the work here as I can't find any more consistent optimizations to motivate that multi-table AdaptiveQuery is faster than CompoundQuery. Even if logically it should, H2 doesn't handle very well the multi-table ORDER BY + LIMIT combination. Single-table AdaptiveQuery is a big deal as H2 is very good with one-table query planning (index selection to solve ORDER BY + LIMIT).

The status right now:

- multi-table AdaptiveQuery is only "decently competitive" with CompoundQuery for iterating all results.
- multi-table AdaptiveQuery is "disastrously slow" when iterating only on some results. Note that the tests were done only with AdaptiveQuery in preselect mode.

Therefore:

- there are side optimizations in #6582 which are really good for the performance (the use of IS (NOT) DISTINCT FROM) and may be integrated in our active branches any time. Upgrading to PostgreSQL 14 is also a "free meal" in terms of performance.
- moving forward with multi-table AdaptiveQuery means cutting down some query cases: this basically means reaching the exact same cases the CompoundQuery optimizer already handles. Having an AdaptiveQuery only for X cases or optimizing the CompoundQuery for the X cases sounds the same (I guess the second is even better being production tested).
- consider way stronger optimizations directly in the H2 engine: this involves some extended work into the H2 optimizer.

#44 - 11/24/2022 10:57 AM - Greg Shah

> consider way stronger optimizations directly in the H2 engine: this involves some extended work into the H2 optimizer

Let's move ahead with this option. We need to make a significant improvement here.

I'll let Eric discuss the other ideas.

#45 - 12/13/2022 10:37 AM - Eric Faulhaber

Greg Shah wrote:

> > consider way stronger optimizations directly in the H2 engine: this involves some extended work into the H2 optimizer
>
> Let's move ahead with this option. We need to make a significant improvement here.

> I'll let Eric discuss the other ideas.

Alexandru, is your comment here regarding improvements to the H2 optimizer primarily about support for multi-table adaptive query, or do you see opportunities that generally would help the current implementation of ProgressiveResults (i.e., using ORDER BY + LIMIT) in single table cases?

In your opinion and based on the testing you and your team have done so far, does it make sense to use ProgressiveResults at all with H2 in its current implementation, or should we be making a runtime decision to instead use ScrollingResults, based on dialect?

What I am trying to figure out is whether it makes more sense to improve H2 to work better with a progressive fetching approach (even in single table cases), or to decide at runtime not use a progressive fetching approach, when the database dialect is H2?

Finally, have you done any tests to determine whether progressive fetching helps or hurts performance with MariaDB?

Thank you.

#46 - 12/14/2022 06:27 AM - Alexandru Lungu

I've wrapped my head around this topic (ProgressiveResults vs ScrollingResults) for a while now, so I can provide some valuable feedback.

> Alexandru, is your comment here regarding improvements to the H2 optimizer primarily about support for multi-table adaptive query, or do you see opportunities that generally would help the current implementation of ProgressiveResults (i.e., using ORDER BY + LIMIT) in single table cases?

While designing my optimization from fwd-h2/rev. 7, I found some solid topics to be addressed in H2:

- In theory, it is an heuristic of stopping the fetching earlier if we know that there are enough records to match both ORDER BY and LIMIT (so the sorting is done on a sub-set of records).
- In practice, the **single-table** queries are using an index anyways (if ORDER BY is made on an indexed set of fields), so there are almost no cases in which my optimization really kicks in. Such cases are: using a query with a non-indexed ORDER BY, but has a prefix of indexed fields.
- In practice, the **multi-table** queries are mostly using this optimization because H2 is not capable of using indexes on each joined table to match ORDER BY. Unfortunately, in FWD all multi-table queries have a more or less complex ORDER BY (multiplex, recid, fields from table 1, fields from table 2, etc.). **This means that FWD multi-table queries are always scanned in H2 because they have a really complex ORDER BY**. In other words, because H2 needs to fetch ALL records in order to sort them at the end, using LIMIT is just an over-kill, unless my optimization is used such that H2 can retrieve only a part of the records (usually this cuts down the time to LIMIT % on indexes with high variance). There is already a mechanism in FWD (AdaptiveQuery.probablyRequiresResort) which takes into consideration if ScrollingResults or ProgressiveResults should be used; for multi-table queries this is still false to encourage the use of ProgressiveResults together with the rev.7 optimization.

Note that the use of the words "all" and "always" means > 95% of the cases.

Regarding my previous comment, I feel like having access to H2 can help us implement exactly the use-case we need here:

- On one hand, we can improve the H2 planning engine. For instance:
  - the optimization I mentioned above is basically part of this list.
  - implement a mechanism of allowing the choosing of an index for each joined table: basically identify that the sort criteria resemble an index from each joined table.
  - rework the H2 planning score to discourage table join permutations: doing a join permutation will always require a resort at the end.
  - implement non-synchronized **database-side table/index/view cursor**: we can iterate single tables (or views) with no LIMIT overhead. This is something I am really intrigued of.
- On the other hand, we can be more specific with the changes to mimic FWD use-case in H2 (#6995 is a place to start):
  - we always iterate tables by an index; maybe we can be more specific and tell H2 that we really want a specific index to be used.
  - move the invalidation logic into H2. In 4GL the "invalidation" seems to be a database-level quirk: changing an indexed field "automatically" moves that record at its right position inside that index. This process is already done under the hood in H2 at some extent. However, FWD is not sensitive to such changes through ResultSet, but may be if we implement a **database-level cursor** which behaves exactly like in 4GL (which, in my opinion, is not that specific for non-scrolling queries).

From my point of view, we should have a middle-ground: SensitiveResults (using live-cursor instead of result-set), which can boost a lot the dynamic single-table queries (or even multi-table queries). This way, we stop bothering about LIMIT, ORDER-BY, OFFSET etc. However, if we focus on using only plain SQL to interact with H2, I guess that only multi-table queries (preselect due to final resort, adaptive due to final resort and limit) are the ones to be optimized. Apart from avoiding table permutation and using index for each table, I can't think of any other solution. Even so, OFFSET is something we can't really optimize and may prove a real burden at some point. Therefore, we can't admit that ScrollingResults is the way to go (due to invalidation and low-number of record scenarios), neither ProgressiveResults (due to database limitations).

From H2 official website: Server side cursors are not supported currently.

> In your opinion and based on the testing you and your team have done so far, does it make sense to use ProgressiveResults at all with H2 in its current implementation, or should we be making a runtime decision to instead use ScrollingResults, based on dialect?

> What I am trying to figure out is whether it makes more sense to improve H2 to work better with a progressive fetching approach (even in single table cases), or to decide at runtime not use a progressive fetching approach, when the database dialect is H2?

> Finally, have you done any tests to determine whether progressive fetching helps or hurts performance with MariaDB?

Most of these remarks are covered in my previous comments. You mention making AdaptiveQuery.probablyRequiresResort always return true for multi-table queries and force ScrollingResults. In this specific moment, both H2 (>=rev.7) and PostgreSQL (>= 13) have "Incremental Sorting" and, thus, both can handle ProgressiveResults in a fairly optimized way. However, there are common "worst-case" scenarios in which ScrollingResults works faster (unfortunately we can't detect this at run-time before execution because it depends on the data). Dialect-wise, MariaDB is not doing that great with ProgressiveResults: it is always faster to use ScrollingResults (consider #6582-33, Preselect vs Adaptive).

If you ask me, multi-table queries in general are not something we really want in our current version of H2. This is because, CompoundQuery is no extremely far away from PreselectQuery for full-result-set fetching. However, if we want only 5% of the records, CompoundQuery is the way to go. For MariaDB and PostgreSQL, ScrollingResults are usually better, but again, if we need only 5% of the records, CompoundQuery is there for us (or even optimized ProgressiveResults for PostgreSQL).

In conclusion, I think our approach should be split between H2, PostgreSQL and MariaDB:

- For H2, we usually work with one-user NO-UNDO tables. This means we can go ahead with very specific changes inside H2 to critically boost the performance: no useless fetched records, no final resort, no LIMIT and OFFSET over-kill, no synchronization or transaction overhead etc. This of course may disregard the switch to other in-memory database that easily.
- For PostgreSQL and MariaDB, the interaction is mostly transactional across several users and we can't easily change any behavior here:
  - I think here is the real discussion over ScrollingResults vs ProgressiveResults.
  - The only decisive statement I have is that multi-table ProgressiveResults are not beneficial for MariaDB. However, this should be a red flag for us, as MariaDB won't be capable of iterating only 5% records efficiently or handle invalidation optimally.
  - For PostgreSQL we can continue discussion, but I would go with ProgressiveResults due to the existing "Incremental sorting". I feel like PostgreSQL is going the right way ("FWD's way") of optimizing multi-table queries with LIMIT.
  - As a final mention, even PostgreSQL allows database-level cursors, but I don't know exactly how we can integrate them.

Sorry for the long post, but I think this can be used as a reference for a decision regarding the H2 optimization path we are going to take.

#47 - 01/20/2023 12:45 PM - Eric Faulhaber

- Related to Support #7058: get the FWD fork of the H2 code base under version control added

#48 - 01/23/2023 05:36 AM - Alexandru Lungu

- Related to Feature #7061: Enable the use of lazy result sets in H2 added

#49 - 01/24/2023 06:13 AM - Alexandru Lungu

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

#50 - 02/27/2023 03:56 AM - Alexandru Lungu

#7061 is mostly finished and now we have a lazy keyword in FWD-H2 which works as a H2 database-cursor allowing us to retrieve one record at a time with no overhead. This lazy approach is faster than both ProgressiveResults and ScrollingResults in all use-cases and doesn't require server-side invalidation.

I think it is time to move on with multi-table AdaptiveQuery, at least for temp database in H2. dirty, meta and persistent H2 databases are concurrent and can't use lazy cursors queries. So right now we address a limited use-case.

- First of all, I want to see a comparison between multi-table CompoundQuery and AdaptiveQuery, considering that:
  - The CompoundQuery uses AdaptiveQuery for each component, so a lazy iterator for each component.
  - The multi-table AdaptiveQuery should allow using lazy for joined queries. This is not supported yet and we may need to implement this first. However, there is a very complex invalidation logic behind, so we will need to tackle this carefully.

Radu, can you allow lazy for multi-table queries in FWD-H2 and add lazy to PreselectQuery just as a test? I am curious of how lazy will behave in such scenario: will it keep a state of the query? will it keep a cursor in each joined table? will it keep a reference to nodes inside table indexes? Is the multi-table lazy query conceptually faster than the CompoundQuery?

My first worry right now is that having a lazy multi-table query may be slower than a server-side CompoundQuery just because the CompoundQuery will iterate each component in an indexed manner! At this point, we need to know if we should tackle #7066 first.

Files

queries.zip	3.85 KB	08/25/2022	Alexandru Lungu