Database - Bug #7991

use ScrollingResults instead of ProgressiveResults when FORWARD-ONLY mode and database cursors are available.

11/01/2023 04:24 AM - Alexandru Lungu

Status:	WIP	**Start date:**	
Priority:	High	**Due date:**	
Assignee:	Dănuț Filimon	**% Done:**	30%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**case_num:**	
vendor_id:	GCD		

Description

Related issues:

Related to Database - Bug #7496: finish support for query:forward-only attribute	**WIP**

History

#1 - 11/01/2023 04:58 AM - Alexandru Lungu

- Subject changed from Presort Query generates a projection FQL to PresortQuery.next improvements

I've seen PreselectQuery.next a lot of times popping as no. 1 candidate in profiling tests (on several customer applications). For one of them, I've noticed that it is usually a PresortQuery. After some slim investigation, it looks like PresortQuery is always extracting "full-records" from the database so it can sort and break them properly. However, the final results stores only their ids (as a projection query does).

This is accurate as it forces the next to load records back from the database to honor cases where the DMO changes in the mean-time **or** is deleted (so it doesn't use stale data). This exact issue is already a problem in the PreselectQuery that doesn't load records back from the database. This issue is attempted to be solved #5768 (5768a). Anyway, for PresortQuery, as all records are hydrated in the mean-time, they will be found in the session cache, so there won't be a second database round-trip.

I don't have an exact idea how to improve the process here. It is important to notice that there are **tons** of queries which instead of FOR EACH tt, use FOR EACH tt BREAK BY f1, but don't use the break groups / accumulation. **I suspect this kind of code was an "optimization" in the legacy code to force iterating non-scrolling, type-forward-only queries with a simple FOR EACH block**.

Maybe we can store the whole DMO in the PresortQuery results to avoid doing a second cache look-up (?).

#2 - 12/06/2023 09:11 AM - Dănuț Filimon

While looking into improving the FQL building, I stumbled upon a similar code to the one from FQLPreprocessor.parse() in FQLPreprocessor(). The parse method makes use of a cache added in #7731 and this constructor can also benefit from it.

#3 - 12/11/2023 04:30 AM - Dănuț Filimon

After investigating #7991 for any other possible improvements, I found the following:

- PreselectQuery.execute should check for the results and components first instead of the sort, this allows a quick exit - **LOW**.
- TemplateResults is a nested class in PreselectQuery, but it is used by other queries with no direct access to it meaning that it has to be done via

the PreselectQuery.execute method. After running a POC test, i noticed that this method is not generating a time worth optimizing, but the fact that the getTemplateRowId mostly returns null and TemplateResults are not used is concerning. - **VERY LOW**.
- Using JMX counters I noticed a large number of OffEndExceptions being thrown when a PresortQuery calls next, around 1/6 ended up throwing it. This is already being worked on in #7045 - **HIGH**.
- During POC tests I noticed PreselectQuery.next() call AdaptiveQuery.next(LockType) for 72% of the time, the 28% is calling PreselectQuery.next(LockType). From the looks of it, a lot of time is spent in ProgressiveResults. This statistic was extracted during run 60/100 - **HIGH**.
- Other points of interest: assembleFQL spents the most time in assembleFromAndWhereClause mostly preparing the FQLPreprocessor, ProgressiveResults.next - **NORMAL**.
- Methods from the FQLPreprocessor.preprocess such as parse where optimized through #7731, but as mentioned in #7991-2 there is another method that can benefit from this cache - **NORMAL**.
- #7855 is also an issue with the aim of integrating the fixEmptyContains into mainWalk. In my opinion, this is a low improvement overall, most of the methods executed in preprocess take around **14 ms** - **LOW**.

#4 - 12/11/2023 05:10 AM - Alexandru Lungu

Danut, can you test the POC by replacing ProgressiveResults with ScrollingResults? Progressive is a pessimistic approach where we avoid fetching too many records fearing that they might get invalidated early. ScrollingResults is optimistic and retrieves all records. I don't think it is trivial to do the replacement while honoring the invalidation as well, but maybe you can explore how ScrollingResults works without invalidation. If the improvement is huge, we shall make a move here.

#5 - 12/11/2023 07:23 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Danut, can you test the POC by replacing ProgressiveResults with ScrollingResults? Progressive is a pessimistic approach where we avoid fetching too many records fearing that they might get invalidated early. ScrollingResults is optimistic and retrieves all records. I don't think it is trivial to do the replacement while honoring the invalidation as well, but maybe you can explore how ScrollingResults works without invalidation. If the improvement is huge, we shall make a move here.

I saw multiple errors such as **Failed to create ... record**, **Record does not exist**, **Invalid item**, **Failed to update ... record** when replacing ProgressiveResults with ScrollingResults because the operation requires a scrollable ResultSet, but is uses FORWARD_ONLY. After replacing the ResultSet.TYPE_FORWARD_ONLY with ResultSet.TYPE_SCROLL_INSENSITIVE, the POC ran without problems. I only ran the POC once and the results were very good compared to the baseline, ~**13%** for the average of the last 20 runs and ~**8%** for the total average. I want to run the POC tests again (2-3 times) to get a correct measurement, is that ok or are the results good enough?

#6 - 12/11/2023 07:32 AM - Alexandru Lungu

Everything that is >1% is worth investigating. Everything >5% is a breakthrough. If your results are correct, it is something worth investigating. **However**, we shall not ignore the fact that ScrollingResults was not tested before with invalidation. You need to double check if these kind of results are actually invalidated or not. Do a separate test-case where you invalidate the results. The query should be re-executed from that reference record.

Invalidating a ScrollingResults is way worse than invalidating a ProgressiveResults. Can you provide an analysis with AdaptiveQuery invalidations tracker using LTW (with 7156b, not including the changes here) over POC, to see how many such invalidation actually happen. AFAIK, there are close to none. Post your results on #7026.

Also, do the tests on the full customer application regression tests (set-up according to Wiki).

#7 - 12/12/2023 03:06 AM - Alexandru Lungu

- Assignee changed from Alexandru Lungu to Radu Apetrii

- Priority changed from Normal to Urgent

Radu, please mark this as an Urgent matter, top most priority in the next 2 days - we should have it merged asap.

Danut will be caught up in #6649 pushing it over the line. The switch from ProgressiveResults to ScrollingResults shouldn't be that hard, but we need hard testing:

- Is the invalidation still honored?
- Is the LAZY mode still doing its job properly?
- Was the ProgressiveResults doing something extra that we stop doing with ScrollingResults.
- Is the performance improvement **that good**; are we missing something?

Please investigate:

```
 saw multiple errors such as Failed to create ... record, Record does not exist, Invalid item, Failed to updat
e ... record when replacing ProgressiveResults with ScrollingResults because the operation requires a scrollab
le ResultSet, but is uses FORWARD_ONLY. After replacing the ResultSet.TYPE_FORWARD_ONLY with ResultSet.TYPE_SC
ROLL_INSENSITIVE, the POC ran without problems
```

I think we still need FORWARD_ONLY in some cases, so hard-coding it back to TYPE_SCROLL_INSENSITIVE may not be that good? Can we detect upfront if we really need TYPE_SCROLL_INSENSITIVE or FORWARD_ONLY? Maybe it is correct to hard-code it to TYPE_SCROLL_INSENSITIVE in this case?

Please make sure you come up with a proper solution on 7791a. After, we can get a Review and start the testing process. Thank you!

#8 - 12/12/2023 03:38 AM - Radu Apetrii

Is there a branch already created for this task or shall I create my own?

#9 - 12/12/2023 03:38 AM - Alexandru Lungu

Just created 7991a.

#10 - 12/12/2023 03:45 AM - Radu Apetrii

Alexandru Lungu wrote:

> I think we still need FORWARD_ONLY in some cases, so hard-coding it back to TYPE_SCROLL_INSENSITIVE may not be that good? Can we detect upfront if we really need TYPE_SCROLL_INSENSITIVE or FORWARD_ONLY? Maybe it is correct to hard-code it to TYPE_SCROLL_INSENSITIVE in this case?

When I first read this I instantly recalled one of Eric's comments about the TYPE_SCROLL_INSENSITIVE vs FORWARD_ONLY topic. It can be found on #7047-5, and I believe it needs to be taken into consideration.

#11 - 12/12/2023 04:06 AM - Alexandru Lungu

Radu Apetrii wrote:

> Alexandru Lungu wrote:
>
> > I think we still need FORWARD_ONLY in some cases, so hard-coding it back to TYPE_SCROLL_INSENSITIVE may not be that good? Can we detect upfront if we really need TYPE_SCROLL_INSENSITIVE or FORWARD_ONLY? Maybe it is correct to hard-code it to TYPE_SCROLL_INSENSITIVE in this case?
>
> When I first read this I instantly recalled one of Eric's comments about the TYPE_SCROLL_INSENSITIVE vs FORWARD_ONLY topic. It can be found on #7047-5, and I believe it needs to be taken into consideration.

Right :/ Changing to ScrollingResults will also break #7241. Dumping data out of the application was fixed using more granular batches of ProgressiveResults, as ScrollingResults was not doing the job properly, loading too much data in the memory. The database cursor should be working only with #7496 (in case the query is set to be forward-only).

Maybe we should get #7496 in trunk before approaching this (?). I still think it is quite risky now, as we don't know how database cursors work on MariaDB. The implementation there is mainly focused on keeping TYPE_SCROLL_INSENSITIVE to determine PG to use cursors. What about MariaDB?

If the switch (TYPE_SCROLL_INSENSITIVE instead of FORWARD_ONLY) works way better on the POC and doesn't break the large customer application on testing, I think we can go ahead with TYPE_SCROLL_INSENSITIVE and ScrollingResults on 7156b, and see how we can deliver it further in trunk.

If this is truly ~8%, we should do our best to get it to 7156b asap and ensure it won't break POC and the large app. First and foremost - ensure that the solution of using TYPE_SCROLL_INSENSITIVE and ScrollingResults **is correct** (a.k.a handles invalidation right). We will worry about memory consumption later.

#12 - 12/12/2023 10:43 AM - Eric Faulhaber

Alexandru Lungu wrote:

> Radu Apetrii wrote:
>
> > Alexandru Lungu wrote:
> >
> > > I think we still need FORWARD_ONLY in some cases, so hard-coding it back to TYPE_SCROLL_INSENSITIVE may not be that good? Can we detect upfront if we really need TYPE_SCROLL_INSENSITIVE or FORWARD_ONLY? Maybe it is correct to hard-code it to TYPE_SCROLL_INSENSITIVE in this case?
> >
> > When I first read this I instantly recalled one of Eric's comments about the TYPE_SCROLL_INSENSITIVE vs FORWARD_ONLY topic. It can be found on #7047-5, and I believe it needs to be taken into consideration.
>
> Right :/ Changing to ScrollingResults will also break #7241. Dumping data out of the application was fixed using more granular batches of ProgressiveResults, as ScrollingResults was not doing the job properly, loading too much data in the memory. The database cursor should be working only with #7496 (in case the query is set to be forward-only).
>
> Maybe we should get #7496 in trunk before approaching this (?). I still think it is quite risky now, as we don't know how database cursors work on MariaDB. The implementation there is mainly focused on keeping TYPE_SCROLL_INSENSITIVE to determine PG to use cursors. What about MariaDB?
>
> If the switch (TYPE_SCROLL_INSENSITIVE instead of FORWARD_ONLY) works way better on the POC and doesn't break the large customer application on testing, I think we can go ahead with TYPE_SCROLL_INSENSITIVE and ScrollingResults on 7156b, and see how we can deliver it further in trunk.
>
> If this is truly ~8%, we should do our best to get it to 7156b asap and ensure it won't break POC and the large app. First and foremost - ensure that the solution of using TYPE_SCROLL_INSENSITIVE and ScrollingResults **is correct** (a.k.a handles invalidation right). We will worry about memory consumption later.

I am very encouraged by the performance findings and I agree that the correctness of behavior has to be the key consideration. That being said, we can't have this change causing OOME instability and regressing #7241 and #7496. This change will need to be regression tested carefully, not just on the POC and large app: note that it was with an early ChUI app on PostgreSQL where the OOME problems originated.

If the behavior/performance/resources are different across different databases, we can always use the dialect infrastructure to tailor the solution to the database in use. I agree that we should test all solutions (for this and #7496) on MariaDB as well.

#13 - 12/13/2023 03:14 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Invalidating a ScrollingResults is way worse than invalidating a ProgressiveResults. Can you provide an analysis with AdaptiveQuery invalidations tracker using LTW (with 7156b, not including the changes here) over POC, to see how many such invalidation actually happen. AFAIK, there are close to none. Post your results on #7026.
>
> Also, do the tests on the full customer application regression tests (set-up according to Wiki).

Alexandru, the LTW test is going to take a fair bit since the process is very slow so I'll put it aside until I finish testing #6649 and #5768. I ran the POC and it reached run 30/100 and had <1.5k invalidations, there were quite a lot of AdaptiveQuery instances.

I shared with Radu a snapshot created with VisualVM of the POC test for PreselectQuery, AdaptiveQuery and PresortQuery so he can take a look and investigate.

#14 - 12/13/2023 08:40 AM - Radu Apetrii

Good news: large application tests (a.k.a. functional tests) pass.

Bad news: while running the adaptive_scrolling testcases, I found the following mismatches:

- On test adaptive_use_same_buffer/adaptive-noscroll-set51.p: ShowHide

Progressive	Scrolling	Progress
2 4 1 2 2 1 2 2 2 2 2 3	2 4 1 2 1 2 2 1 3	2 4 1 2 1 2 2 1 3 2 2 1 2 2 2 2 2 3

> This isn't necessarily a regression since it wasn't working before. I just find it funny how Progressive generates half of the result and Scrolling the other half in order to have the full result as in Progress.

- On test adaptive_use_same_buffer/adaptive-noscroll-set54.p: ShowHide

Progressive	Scrolling	Progress
10 10 10 10 10 2 10 10 2 4 4 4 4 4 4 4 4 4 10 10 10	10 10 10 10 10 2 10 10 2 4 4 4 4 4 4 4 4 4	10 10 10 10 10 2 10 10 2 4 4 4 4 4 4 4 4 4 10 10 10

> This test has always been a problem and it seems like it continues to be. Here, Scrolling isn't sensitive to the data change and skips the last line (that appears in both Progressive and Progress).

I'll move on and test the performance, but I'll keep these results in mind and think of a solution.

#15 - 12/14/2023 07:35 AM - Radu Apetrii

I finished running the performance tests. The results are quite good:

- By setting TYPE_SCROLL_INSENSITIVE instead of FORWARD_ONLY (and nothing else), the improvement is ~**-1%**.
- By replacing ProgressiveResults with ScrollingResults, plus the above mentioned change, the improvement is ~**-4.8%**.

I guess the next objective would be to check the ScrollingResults implementation, maybe we can get those two failing tests fixed (#7991-14)?

#16 - 12/14/2023 08:01 AM - Alexandru Lungu

Exactly, please do.

EDIT: sometime ago I've seen that the invalidation on the query was done only if the record change event was generated by the queried buffer. I don't think this is right; there were multiple examples in which changing record from another buffer was also invalidating. ProgressiveResults has a second check (if the last record in the prev. batch doesn't match the record in the next batch, it will invalidate). This does not exist in ScrollingResults, so it is "exposed" to some new problems.

#17 - 12/14/2023 08:51 AM - Eric Faulhaber

Radu Apetrii wrote:

> I finished running the performance tests.
> [...]

Which dialect(s) did you test?

#18 - 12/14/2023 08:54 AM - Radu Apetrii

Eric Faulhaber wrote:

> Radu Apetrii wrote:
>
> > I finished running the performance tests.
> > [...]
>
> Which dialect(s) did you test?

Only PostgreSQL. Now that you mention, I'll add MariaDB performance/regression testing to the to-do list.

#19 - 12/14/2023 11:39 AM - Eric Faulhaber

Do any of the tests include a query which would generate a very large result set? I would think that there would be some that at least would trigger multiple ProgressiveResults brackets, since we are seeing a performance improvement by switching to ScrollingResults, which effectively is always one "bracket".

Further, does any test generate a very large result set, but then only use one or a small number of those results? Something like:

```
FOR EACH some_large_table WHERE <some wide open condition> NO-LOCK:
   <do something...>
   LEAVE.
END.
```

In this case, we leave after only one iteration, but a more realistic case would do some work until some separate condition from that of the WHERE clause was reached, then leave. In such a case, it could be less clear that only one or a handful of records might actually be used, though thousands might match the WHERE clause.

This type of code was the reason ProgressiveResults was created. It represents a case where ProgressiveResults would be expected to perform much better (at least on PostgreSQL) than ScrollingResults.

As noted earlier, large result sets (regardless of whether only a handful of records are used) are also where we could get into memory trouble by switching away from TYPE_FORWARD_ONLY. See https://jdbc.postgresql.org/documentation/query/#getting-results-based-on-a-cursor. This documentation was written a long time ago, but the site overall is current, so I don't think this aspect (i.e., the driver collecting all results at once and the case for server-side cursors) has changed.

#20 - 12/15/2023 03:43 AM - Alexandru Lungu

- Priority changed from Urgent to High

I think we should take an incremental approach on this. Lets focus on #7496 and merge it first, before moving on this one. That ensure that we actually have FORWARD-ONLY run-time (including the save of the remaining results if the transaction ends and the portal closes - test if this is dialect specific). Danut, mind that ForwardResults are already in trunk and are used for PresortQuery quite well already.

I will move this to High as there are several aspects to take care of first - having it on emergency is not feasible now.

We need **lots** of testing (with large customer applications, regression tests, custom tests) before moving on with #7496. Danut, please refer to 4GL_Unit_Testing and ask Serban for advice first as he has some knowledge on testing with it already. Please update #7496 with the points left to solve and eventually move it to Internal Testing if there are none left - will continue discussion there.

This task is pending #7496.

#21 - 12/15/2023 03:57 AM - Dănuț Filimon

I actually tested the changes from #7496 previously for #7767 as the changes could be a small performance boost and ended up with a few exceptions which is not good. I'll need to take a look at the implementation again and check if everything is working well before moving forward.

#22 - 12/19/2023 03:57 PM - Greg Shah

Do we have any chance of finishing this task by Thursday EOD?

#23 - 12/20/2023 08:25 AM - Alexandru Lungu

- Subject changed from PresortQuery.next improvements to use ScrollingResults instead of ProgressiveResults when FORWARD-ONLY mode and database cursors are available.

Greg, the whole task won't be finished by Thursday - most of the work moved to #7496. There is a change there with -0.7% that will get merged asap. After, the whole 7496a will be completed, tested and merged. At that point, we can actually return to the work here (#7991). Once we have support for FORWARD-ONLY, we can safely move most of the queries (used in FOR EACH loops) to ForwardResutls and benefit from database cursors (at least for PG). This should be largely tested. Ultimately, MariaDB should be tested as well.

I will change the name here to: use ScrollingResults instead of ProgressiveResults when FORWARD-ONLY mode and database cursors are available.

#24 - 01/31/2024 04:40 AM - Alexandru Lungu

- Assignee changed from Radu Apetrii to Dănuț Filimon

Danut is mostly into #7496 which is very related to this.

#25 - 01/31/2024 04:41 AM - Alexandru Lungu

- Related to Bug #7496: finish support for query:forward-only attribute added

#26 - 01/31/2024 04:45 AM - Alexandru Lungu

Eric, please mark this as a top candidate for performance improvement. Once we have FORWARD-ONLY fully implemented with #7496, we can:

- Adjust the FOR-EACH queries to use ForwardOnlyResults with db-side cursors. This will reduce the database queries and the total work done by AdaptiveQuery.
- Adjust the OPEN QUERY queries that use FORWARD-ONLY with db-side cursors. This will (again) reduce the database queries.

The reduction happens because we don't use ProgressiveResults anymore for such queries.

#27 - 02/14/2024 03:16 AM - Dănuț Filimon

Rebased **7991a** to trunk revision **14985**.

#28 - 02/15/2024 03:46 AM - Dănuț Filimon

- % Done changed from 0 to 30

Committed 7991a/rev.14986. Adjusted FOR-EACH queries to use ForwardResults.

I've looked into how we can make use of ForwardResults when FOR EACH queries are present and a previous update added the query to the forEach call in the converted code. This was not available in **7991a**, so I rebased the branch when I noticed this. The changes include the addition of

the forwardOnly flag to AbstractQuery (by default false) and setting it to true in the forEachWorker when the query is an AdaptiveQuery.

Currently the ForwardResults replace the usage of ProgressiveResults for the mentioned case and I've tested these changes with POC performance and regression tests. I also wanted to run ChUI tests but 6667e does not have the additional query parameter change.

In case of the performance tests I did not find a noticeable improvement and created a few JMX counter to monitor how often the changes are used:

For the POC performance tests (warmup run + 5 runs), around **4.76%** AdaptiveQuery instances will create ForwardResults and **68.8%** of the forEachWorker calls will result into an AdaptiveQuery being used. The number are great, but the performance results of the 100 runs are not as expected. I suspect that the number of brackets is very low in the case of ProgressiveResults, this would explain the performance results since the number of times the results are retrieved closely match the number of times ForwardResults would be created. The solution for this would be to reduce the base of the ProgressiveResults to something like 5 or 8 and check the performance for this cases.

#29 - 02/15/2024 04:00 AM - Alexandru Lungu

> The solution for this would be to reduce the base of the ProgressiveResults to something like 5 or 8 and check the performance for this cases.

This will make the current solution in trunk slower to prove that ForwardResults is faster. This is not an option.

> For the POC performance tests (warmup run + 5 runs), around 4.76% AdaptiveQuery instances will create ForwardResults

This is not as expected 4.75% is quite low. Does this mean that there are way more OPEN QUERY queries than FOR EACH queries; so this leads to ~95% AdaptiveQuery being generated by OPEN QUERY? Or maybe we should look into AdaptiveQuery used as components for CompoundQuery?

Please clarify which is the 4GL construct that generates the most AdaptiveQuery instances: OPEN QUERY, FOR EACH or CompoundQuery component.

#30 - 02/15/2024 07:38 AM - Dănuț Filimon

Alexandru Lungu wrote:

> The solution for this would be to reduce the base of the ProgressiveResults to something like 5 or 8 and check the performance for this cases.

> This will make the current solution in trunk slower to prove that ForwardResults is faster. This is not an option.

Sorry for the misunderstanding, the solution I suggested was for **testing** to see if we have a performance improvement, to use 5 and 8 and see if the number of brackets was still close to the number of ForwardResults being created. If we use 5 and 8 we will have more brackets and we will be able to notice a change in performance. This is not a "solution", but something that could help with investigating the performance in detail.

> For the POC performance tests (warmup run + 5 runs), around 4.76% AdaptiveQuery instances will create ForwardResults

> This is not as expected 4.75% is quite low. Does this mean that there are way more OPEN QUERY queries than FOR EACH queries; so this leads to ~95% AdaptiveQuery being generated by OPEN QUERY? Or maybe we should look into AdaptiveQuery used as components for CompoundQuery?

Please note that executeQuery also has a isLazyMode() call that precedes isForwardOnly() and creates ScrollingResults, this also influences the ForwardResults since they may not be created. I will make sure to test how many ScrollingResults and ProgressiveResults are created next time.

> Please clarify which is the 4GL construct that generates the most AdaptiveQuery instances: OPEN QUERY, FOR EACH or CompoundQuery component.

Working on it.

#31 - 02/15/2024 11:25 AM - Dănuț Filimon

I've made more JMX counters and obtained new results (POC performance warmup + 5 runs):

Counter	Count 1	Count 2
Number of AdaptiveQuery components of a forward-only CompoundQuery	2340	2340
Total number of AdaptiveQuery instances created	133811	133811
Number of times where Adaptive.executeQuery is called	127423	127423
Number of times ScrollingResults are created (lazy clause)	65377	65377
Number of times ForwardResults are created	6069	7507
Number of times ProgressiveResults are created	55977	54539
Number of times FOR EACH worker is called	87505	87505
Number of times the query of the FOR EACH worker is an AdaptiveQuery	60197	60197
Number of times the query of the FOR EACH worker is a CompoundQuery	1478	1478
Number of times the query of the FOR EACH worker is another type of query	25830	25830

The **Count 1** is an update to the previous results and the **Count 2** will set the foward-only flag to **true** for any AdaptiveQuery component of a CompoundQuery. In the second set of results, the percentage of AdaptiveQuery instances that create ForwardResults went from **4.76%** to **5.9%** (this percentage excludes AdaptiveQuery components that are added through QueryWrapper and by PresortCompoundQuery - which I could not find any component being used but would like if this could be confirmed).

I also confirmed that **9750** CompoundQuery instances are created, and there are an additional **20907** AdaptiveQuery components which could be made forward-only and this would bring the percentage up to ~**16.8%**. But those are the queries that are not generated by FOR EACH and need to be investigated.

Alexandru Lungu wrote:

> Please clarify which is the 4GL construct that generates the most AdaptiveQuery instances: OPEN QUERY, FOR EACH or CompoundQuery component.

From a total of **133811** AdaptiveQuery instances created, **60197** are directly used in FOR EACH, **23247** are components of a CompoundQuery and the rest of **50367** should be generated by OPEN QUERY.

From my findings, the OPEN QUERY has the most potential here to raise the percentage of AdaptiveQuery instances that are forward-only.

#32 - 02/16/2024 10:30 AM - Greg Shah

> I also wanted to run ChUI tests but 6667e does not have the additional query parameter change.

Make sure you are using 6667i which is being rebased and is kept closer to trunk.

#33 - 02/19/2024 06:58 AM - Dănuț Filimon

Greg Shah wrote:

> > I also wanted to run ChUI tests but 6667e does not have the additional query parameter change.
>
> Make sure you are using 6667i which is being rebased and is kept closer to trunk.

Thank you for the information, I was planning to shift to 6667f. Before anything else, is **6667i** a stable branch that passes all regression tests? The wikis only mentions **6667e** / **6667f**, will the documentation be updated?

#34 - 02/19/2024 07:35 AM - Greg Shah

Dănuț Filimon wrote:

> Greg Shah wrote:
>
> > I also wanted to run ChUI tests but 6667e does not have the additional query parameter change.
>
> Make sure you are using 6667i which is being rebased and is kept closer to trunk.

> Thank you for the information, I was planning to shift to 6667f. Before anything else, is **6667i** a stable branch that passes all regression tests? The wikis only mentions **6667e** / **6667f**, will the documentation be updated?

Tomasz: Please advise.

#35 - 02/19/2024 07:37 AM - Tomasz Domin

Greg Shah wrote:

> Dănuț Filimon wrote:
>
> > Greg Shah wrote:
> >
> > > I also wanted to run ChUI tests but 6667e does not have the additional query parameter change.
> >
> > Make sure you are using 6667i which is being rebased and is kept closer to trunk.
>
> Thank you for the information, I was planning to shift to 6667f. Before anything else, is **6667i** a stable branch that passes all regression tests? The wikis only mentions **6667e** / **6667f**, will the documentation be updated?
>
> Tomasz: Please advise.

Please use 6667i, I'll update wiki pages.

#36 - 02/19/2024 08:52 AM - Dănuț Filimon

I've gathered more information for #7991-31.

From a total of **133811** AdaptiveQuery instances created, **60197** are directly used in FOR EACH and **44687** are created in QueryWrapper.prepare() (this is resulted from a CREATE QUERY qh. ... qh:QUERY-PREPARE("FOR EACH pt1") type of statement). This leaves **28927** which should be from generated from OPEN QUERY and use QueryWrapper.assign() but the method is not called at all. AdaptiveFind instances are not generated, but CompoundQuery$Optimizer.serverJoinAdaptive() generates **5680** AdaptiveQuery instances so we actually have **23247** AdaptiveQuery that I don't know yet how they are instantiated.

#37 - 02/19/2024 09:21 AM - Tomasz Domin

Greg Shah wrote:

> Thank you for the information, I was planning to shift to 6667f. Before anything else, is **6667i** a stable branch that passes all regression tests? The wikis only mentions **6667e** / **6667f**, will the documentation be updated?

Tomasz: Please advise.

Once more. I've updated wiki page.
Just to note - please use branch 6667i. You also need to apply patch taken from #6667-823
If you would need a newer 6667i I will rebase this week to the latest trunk.