# Database - Feature #6816

## improve PreselectQuery.assembleHQL

10/06/2022 02:26 AM - Constantin Asofiei

| | | | |
|---|---|---|---|
| **Status:** | Internal Test | **Start date:** | |
| **Priority:** | Normal | **Due date:** | |
| **Assignee:** | Radu Apetrii | **% Done:** | 100% |
| **Category:** | | **Estimated time:** | 0.00 hour |
| **Target version:** | | | |
| **billable:** | No | **version:** | |
| **vendor_id:** | GCD | | |

**Description**

### History

#### #1 - 10/06/2022 02:36 AM - Constantin Asofiei

The code in PreselectQuery.assembleHQL appears in the profiler when testing very complex application APIs, in 6129a. In that case, it is called ~20k times, and it creates lots of objects. We need to make this more efficient both in terms of CPU speed and allocated objects.

#### #2 - 11/17/2022 03:33 AM - Alexandru Lungu

*- Status changed from New to WIP*

*- Assignee set to Radu Apetrii*

#### #3 - 11/17/2022 05:40 AM - Alexandru Lungu

For an initial check, I recommend implementing a very small suite of performance tests (~6) for PreselectQuery to test different aspects of HQL composition. Use ~10k preselect queries with very low iteration count (~2). Use temp-tables in:

- 1 test with high number of selected fields (one big table, small where, no index)
- 1 test with several joins (multiple small tables, small where, no indexes)
- 2 tests with a large WHERE clause
  - one small table, large where clause, no index
  - one small table, where clause with placeholders (?), no index
- 1 test with a large ORDER BY clause (some small tables, small where, multiple large indexes)
- 1 overall test (multiple tables, large where clause, multiple large indexes)

For each of these tests, record the QUERY_ASSEMBLE JMX average time output. This should work as a baseline for your further improvements. Note that VisualVM can also hint the hotspots.

#### #4 - 12/05/2022 02:50 AM - Radu Apetrii

After I did some investigation upon the PreselectQuery.assembleHQL function, two types of optimization occurred to me. The first one consists of small changes (e.g. replacing unnecessary function calls with a variable that contains the result). **The second type is more of a "big deal" and I would like to know your opinion on this since it requires some time to implement.**
The plan is to introduce a (double-layered) cache for the assembly of the hql. Thus, the process will work as follows:

- Phase 1: first, the program will check if the hql can be retrieved immediately from the cache (all at once). If so, then there is no need to move to phase 2.

- Phase 2: the program will check every function/method of the 5 (assembleOrderByClause, assembleSelectClause, etc.) and see if their result can be retrieved from the cache. The ones that cannot get their result from cache will move to phase 3.
- Phase 3: call the function/method normally and store the result in the cache.

Let me know if you think this is a good idea and/or if you would change some aspects in the thought process. Thank you!

#### #5 - 12/14/2022 04:22 AM - Radu Apetrii

*- File 6816.patch added*

Update: I noticed that SortCriterion.toString gets called quite often so I had a look in that function to see if some improvements can be made. This patch does just that.
**I have introduced a cache for storing the result of the SortCriterion.toSortExpression function**, which is used in toString.

The key is formed out of the four parameters of the function. In addition, to make sure that no other components are required for the key, I've modified the constructor a little bit in order to make the SortCriterion.name, SortCriterion.subscript and SortCriterion.dialect final, since their values were not changing anywhere.

These changes were tested with a large customer application, with 3821c rev. 14437. Also, in terms of performance, the execution time of running all the SortCriterion.toString functions in a program (on a test I made) dropped from ~185ms to ~144ms for a single user, and from ~4086ms to ~3013ms for 10 users running at the same time. The cache hit ratio is about 80%, again, on the test I ran.

As a side note, I tried to implement the cache with String[] and Collections.synchronizedList, but the Vector approach provided the best results.

#### #6 - 12/15/2022 08:36 AM - Radu Apetrii

*- File 6816-2.patch added*

Another update: after the last post I started working on a Factory design pattern for SortCriterion. This patch includes that file and some other small changes that are required in order to make this work:

- **Instead of creating a new instance of SortCriterion, SortCriterionFactory now checks in its cache if there is an instance equal to the one desired.** This works because all of SortCriterion's components are final.
- I added to the Cache class a function called computeIfAbsent which does the following thing: if the cache contains a value related to the given key, then the value is returned. Otherwise, the given function tries to compute a value, and if it is not null, it is then put in the cache. (I believe the Cache.computeIfAbsent javadoc is more descriptive than this comment)
- The places in which a new SortCriterion was created were changed with a SortCriterionFactory.getInstance in order to fulfill the plan.

I worked with 3821c, rev. 14437. In terms of performance, there are two things I want to address:

- On some tests I have tried (with single user and multiple users at the same time), the cache hit ratio almost reached 100%, which provided a massive boost to the process of "creating" SortCriterion s. Long story short, **that process went down from ~163ms to ~66ms for a single user run and from ~2749ms to ~1487ms for 10 users running at the same time** (obviously, synchronization slowed the process a bit).
- **On a large customer application, the cache hit ratio reached 88.91%**, which seems really decent IMO.

I want to point out that I tried different capacities for the SortCriterionFactory cache, but the value 2048 provided the most balanced results.
And one more thing, these patch includes both the changes (this one and the last post one).
Is it OK to commit these changes?

#### #7 - 01/04/2023 11:30 AM - Constantin Asofiei

Radu, can you port this patch to 6129b? I'd like to test the impact on a customer's application.

#### #8 - 01/05/2023 06:26 AM - Radu Apetrii

*- File 6129b-6816.patch added*

Sure, here you go. I've updated the file so the command patch now works on 6129b.

#### #9 - 01/05/2023 07:45 AM - Constantin Asofiei

Unfortunately the patch shows a slowdown in the performance testing. I'm not sure from where it is. I wonder if is not related to many cache misses (too small cache size), I'll check a heap dump and see what the cache reports.

This is a review (mostly formatting) for the patch:

- in cache.Cache.computeIfAbsent, parameters need to be on the same line, if they fit in the max line limit
- in ExpiryCache.computeIfAbsent, you have lines like if ((v = get(key)) == null) - v is never even really used, and also including the assignment in the IF test just makes the code less readable. Same with Objects.requireNonNull(mappingFunction);, I don't think is needed.
- in SortCriterion, you have a Vector toStringCache - why not use instead directly a Java array? As the array from the look of it doesn't need to grow, is fixed size.
- in SortCriterionFactory:
  - the JPRM column from the header needs to be removed.
  - the cache field can be private

#### #10 - 01/05/2023 07:59 AM - Constantin Asofiei

I have a mod count for the LRUCache.cache map (used by SortCrtierionFactory) of 11k for just 10 runs of the performance tests. So this just adds overhead, there are lots of cache misses - and it might not be solved just by increasing the cache size.

Please check how the cache behaves when you have (the same) temp-table created dynamically (with different temp-table names, just the fields are the same) and after that used in dynamic queries. Also, does SortCrtierionFactory$Key.text have the columns prefixed with the DMO alias?

#### #11 - 01/05/2023 09:15 AM - Alexandru Lungu

Is the profiling done with multiple users? The cache is global across all users, so it is a synchronized data structure. Maybe this can lead to some overhead?

Also, there are two "somehow independent" optimizations: sort criterion factory and SortCriterion toString caching. Maybe we can integrate them separately. Clearly the "toString" optimization is more powerful along the factory, but anyways.

#### #12 - 01/05/2023 10:06 AM - Constantin Asofiei

Alexandru Lungu wrote:

> Is the profiling done with multiple users? The cache is global across all users, so it is a synchronized data structure. Maybe this can lead to some overhead?

No, the profiling is done with a single user, which executes the tests sequentially.

#### #13 - 01/05/2023 11:06 AM - Constantin Asofiei

I've increased the cache limit to something big (2048000) and now for 100 runs the average is only a little less worse (11513ms vs 11435ms without this patch, when previously the patch had 11621ms). The actual number of entries in the cache is 12590.

Radu: please think of a way to stress-test this, and find what can be improved. Maybe doing some profiling when populating the cache with 10k entries (and using some 100s of queries which call assembleHQL 20k times) can help.

Also, the cache Key doesn't need the dialect - the DmoMeta already is distinct for temp-table and permanent DMOs. And instead of DmoMeta, the DMO class can be used safely at the key. From the heap dump, the Key.text does have the buffer alias as prefix, but I don't think this can be easily removed.

#### #14 - 01/05/2023 02:20 PM - Eric Faulhaber

Constantin Asofiei wrote:

> Also, the cache Key doesn't need the dialect - the DmoMeta already is distinct for temp-table and permanent DMOs.

The possible distinction of dialects is not only between temp-table and permanent table. It may be a rare or even just theoretical case, but an application could be configured with multiple, permanent databases of different dialects, using the same schema. So, I think the dialect is needed for the cache key, since the same generic sort criterion input taken from a converted query may be modified for a particular dialect when generating the final FQL sort expression (e.g., using a computed column property name instead of an upper/rtrim expression).

In retrospect, this processing seems better placed in the SQL generation step, instead of producing dialect-specific FQL. However, we used to rely on Hibernate for the HQL -> SQL generation, so we had less flexibility at that time and pushed this processing back to SortCriterion.

#### #15 - 01/05/2023 02:25 PM - Constantin Asofiei

Eric, I was assuming that the SortCriterion is being used at the FQL level, before the FQL translation (dependent on the dialect) is performed. Am I mistaken?

#### #16 - 01/05/2023 02:52 PM - Eric Faulhaber

Constantin Asofiei wrote:

> Alexandru Lungu wrote:
>
> > Is the profiling done with multiple users? The cache is global across all users, so it is a synchronized data structure. Maybe this can lead to some overhead?
>
> No, the profiling is done with a single user, which executes the tests sequentially.

Executing a synchronized block still can incur significant overhead, even for a single user. I noticed this while profiling SourceNameMapper APIs some time ago. Are we seeing any bottleneck around the cache use? We can temporarily disable the synchronization for comparison, since this testing is single-user.

In looking over the SortCriterion code, it seems to me that there are two things that are calling out for improvement in this code, but we should only take on this effort if we are seeing a clear bottleneck in these places:

- Do a much better job with the sort criterion/criteria parsing. That is, implement single-pass parsing in the methods/c'tors which parse. The aim would be to avoid all the extra string scanning we do by using redundant invocations of indexOf, substring, equalsIgnoreCase, etc. I'm not suggesting a full-blown grammar-based parser; hopefully, a hand-written character scanning loop for these relatively simple strings will suffice. BTW, I'm guilty of most of this, as I originally wrote this class.
- Assuming the above does not obviate the need for the LRU caches, we should consider using better synchronization for the caching than the simple synchronized block, leveraging the java.util.concurrent.locks package.

Considering that the overhead of the LRUCache is not insignificant, I wonder whether improving the parsing makes the LRU caching less attractive.

Again, these ideas currently are based on gut instinct, not profiling. Let's confirm that these are real bottlenecks with profiling before considering these improvements.

#### #17 - 01/05/2023 03:05 PM - Eric Faulhaber

Constantin Asofiei wrote:

> Eric, I was assuming that the SortCriterion is being used at the FQL level, before the FQL translation (dependent on the dialect) is performed. Am I mistaken?

That would be better, and I was (wishfully) remembering it this way, too. But the dialect does get stored in the SortCriterion instances and it is used to make decisions about how the FQL order by clause is generated, *before* the translation to SQL. Note the use of dialect in the SortCriterion c'tor and toSortExpression method.

This was the bit I mentioned at the end of my #6816-14 post. IIRC, I wrote it this way originally because we were handing off HQL to Hibernate that already referenced DMO properties either as computed columns or wrapped by upper[/rtrim], to allow Hibernate to generate the SQL correctly. This technical debt was not resolved when we pulled out Hibernate, so it still works that way now.

#### #18 - 01/13/2023 06:25 AM - Radu Apetrii

*- File 6129b-6816-2.patch added*

I've attached a patch in which I've done some refactoring to the SortCriterion constructor. Mainly, instead of calling all those functions that iterate through the text variable, **the program now does one iteration**. This was done using 6129b, rev. 14351.

But, there are some problems. Initially, when I tested these changes with a version of 3821c, the results were good (the time of execution of the constructor decreased by ~20%). After I moved the changes to 6129b, the results seemed problematic. By that I mean the execution time was worse than the current implementation by quite a lot. So I started to add more changes and I've managed to narrow down the performance time (the constructor is now 15% slower than the current implementation).

The for-loop itself (from the patch) is faster than the processing part of the initial implementation, but, **the part where the variables get assigned takes some time** and this is hurting the whole process. Can you let me know, please, if there is something that I've missed or if you get the same results as me? Thank you!

Note: I've tried multiple implementations of the constructor, but this one is the fastest on 6129b.

#### #19 - 01/20/2023 03:11 AM - Constantin Asofiei

I've tried the patch with a customer application and there is an abend:

```
Caused by: java.lang.StringIndexOutOfBoundsException: String index out of range: -1
        at java.lang.String.<init>(String.java:196)
        at com.goldencode.p2j.persist.SortCriterion.<init>(SortCriterion.java:391)
        at com.goldencode.p2j.persist.SortCriterion.<init>(SortCriterion.java:319)
        at com.goldencode.p2j.persist.SortCriterion.parse(SortCriterion.java:568)
        at com.goldencode.p2j.persist.SortCriterion.parse(SortCriterion.java:676)
        at com.goldencode.p2j.persist.SortCriterion.parse(SortCriterion.java:652)
        at com.goldencode.p2j.persist.AdaptiveComponent.initialize(AdaptiveComponent.java:327)
        at com.goldencode.p2j.persist.AdaptiveQuery.execute(AdaptiveQuery.java:2870)
        at com.goldencode.p2j.persist.PreselectQuery.getResults(PreselectQuery.java:6034)
        at com.goldencode.p2j.persist.AdaptiveQuery.next(AdaptiveQuery.java:1660)
        at com.goldencode.p2j.persist.PreselectQuery.next(PreselectQuery.java:2617)
```

#### #20 - 01/20/2023 06:52 AM - Radu Apetrii

*- File 6129b-6816-2-fixed.patch added*

With Alex's help, I've managed to find an example that results in that error. The problem was my misinterpretation of the text field given as a parameter in the SortCriterion constructor.
From all my gatherings of this parameter, at the end of each one, there would always be an asc or desc keyword, so I just assumed that this would always be the case. **I did not take into consideration that asc or desc might be missing from the text**, which meant that, implicitly, SortCriterion.ascending should be true. Because I was computing some fields based on the position of the space (but there was no space in the given text field), the program gave an error.

I hope this solves the problem. If you find any more, please let me know and I will deal with them ASAP.
I navigated through a customer application, tried some other tests and I could not find any errors. These changes were tested with 6129b, rev. 14367.
Thank you and sorry for the trouble!

#### #21 - 01/20/2023 09:33 AM - Constantin Asofiei

I think this can be committed  to 6129b.  Alexandru, any concerns?

#### #22 - 01/20/2023 10:38 AM - Alexandru Lungu

Looks good, no concerns.

#### #23 - 01/20/2023 10:39 AM - Alexandru Lungu

*- % Done changed from 0 to 40*

#### #24 - 01/23/2023 02:53 AM - Radu Apetrii

Alexandru Lungu wrote:

> Looks good, no concerns.

Done. Committed to 6129b, rev. 14379.

#### #25 - 01/19/2024 01:37 AM - Eric Faulhaber

The changes implemented for this task have long since made their way into trunk. It is still marked 40% done. Is there an expectation that further improvements can be made or is this task essentially finished?

#### #26 - 01/19/2024 05:05 AM - Radu Apetrii

Give me a bit of time to check PreselectQuery.assembleFQL, maybe I can notice something else. I'm not sure if Alex had in mind other optimizations, though.

#### #27 - 01/19/2024 05:15 AM - Alexandru Lungu

I am dumping some statistics with LTW right now. I will let you know if I continue seeing something obvious here.

#### #28 - 01/19/2024 05:15 AM - Alexandru Lungu

*- Status changed from WIP to Internal Test*

*- % Done changed from 40 to 100*

### Files

| | | | |
|---|---|---|---|
| 6816.patch | 5.17 KB | 12/14/2022 | Radu Apetrii |
| 6816-2.patch | 24.4 KB | 12/15/2022 | Radu Apetrii |
| 6129b-6816.patch | 24.5 KB | 01/05/2023 | Radu Apetrii |
| 6129b-6816-2.patch | 5.32 KB | 01/13/2023 | Radu Apetrii |
| 6129b-6816-2-fixed.patch | 5.67 KB | 01/20/2023 | Radu Apetrii |