

Introduction.

Some words about the method of testing on the local machine:

- I was using JMH Insert/Select benchmarks from our code base.
- Every run was firstly analyzed “rough” by the async profile with cpu and wall clock sampling to find the pieces, which can be found by this cheapest approach
- Potentially suspicious places wrapped around by the simple-one-monitor-one-class measurement tool. Each wrapped call saves the start and end in the array.
- On the get/put method end all measurements printed in the file/committed to JFR, it depends on the settings
- In the file output, it has additional features,
 - “Here is hidden N us” - which gives us an estimation which amount of time we lose between two logged actions. This feature is highly helpful, when I was trying to find the longest pauses between known pieces.
 - Whole time - the whole time of query in us
 - Found time - is the sum of all known measures from the list above in nanos
 - Not found time/Percentage of found - calculated according to the difference of whole time and found time (nanos).

After the collecting bottlenecks on local runs:

- Moreover, local runs are the place where we find the main bottlenecks, but then all our hypotheses are finally checked on the YCSB benchmark on teamcity with enabled JFR logging. So, all local bottlenecks must be real and local laptop issues must be eliminated. **All provided profile dumps received on the real YCSB runs on TeamCity**

How to recheck my results.

If you want to run the same workload locally:

- Get the <https://github.com/gridgain/apache-ignite-3/commit/a57d313cada91aa9464114c33bbfa149f51cef56>
- Change the file path here <https://github.com/gridgain/apache-ignite-3/blob/653bf33c6ac5476a9110fe29f3cd3bec032ea46c/modules/core/src/main/java/org/apache/ignite/Instrumentation.java#L38>
- Run 1 node, fsync=false <https://github.com/gridgain/apache-ignite-3/blob/a57d313cada91aa9464114c33bbfa149f51cef56/modules/runner/src/integrationTest/java/org/apache/ignite/internal/benchmark/SelectBenchmark.java#L131> . Look at the file with results from the previous point

KV get performance investigations.

So, typical get query on SelectBenchmark.java run with my Instrumentation looks like (Teamcity runs with the same JFR events confirms this picture):

```
kvGetMark 0.031 7162381 7162412
    Here is hidden 0.575 us
schemaVersionAt 0.918 7162987 7163905
    Here is hidden 0.297 us
keyMarshaller 0.326 7164202 7164528
    Here is hidden 0.062 us
valueMarshaller 0.042 7164590 7164632
    Here is hidden 0.058 us
buildRow 0.476 7164690 7165166
    Here is hidden 0.089 us
beginTransaction 1.491 7165255 7166746
    Here is hidden 0.288 us
awaitPrimaryReplica 0.8 7167034 7167834
    Here is hidden 0.323 us
responseFuture 3.992 7168157 7172149
    Here is hidden 1.243 us
getPrimaryReplica 0.293 7173392 7173685
    Here is hidden 1.092 us
indexGet 0.283 7174777 7175060
    Here is hidden 2.416 us
readPageMemory 3.196 7177476 7180672
    Here is hidden 2.221 us
whenAsyncExecutor 8.097 7182893 7190990
    Here is hidden 0.288 us
HybridClock#update 0.448 7191278 7191726
    Here is hidden 0.333 us
postEvaluate 0.577 7192059 7192636
    Here is hidden 0.257 us
unmarshalValue 0.254 7192893 7193147
Whole time: 36.5 us
Found time: 21224 Not found time: 15276.0 Percentage of found: 58.14794520547945
```

So, here we have some unknown (hidden) parts and some leaders in time consumption:

- [beginTransaction](#) 1.491us
- [responseFuture](#) 3.992us (surprise for me)
- [readPageMemory](#) 3.196us

- whenAsyncExecutor 8.097us (one another surprise)

responseFuture

The tricky part, that the call of [CompletableFuture.orTimeout\(\) here](#) is wasting the time, which is comparable with the main readPageMemory activities. If we create the simple **new CompletableFuture()** here - it will be ok. Moreover, this replacement change the performance of the whole YCSB runs also.

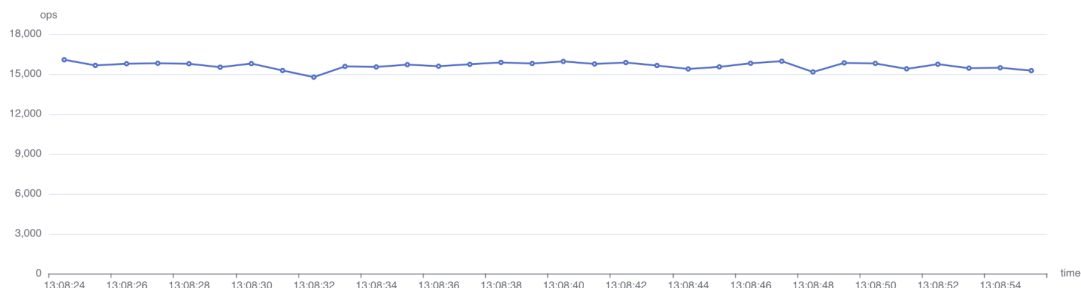
whenAsyncExecutor

The same as previous - simple change [here](#) from whenCompleteAsync to whenComplete - remove the performance penalty on this peace of code

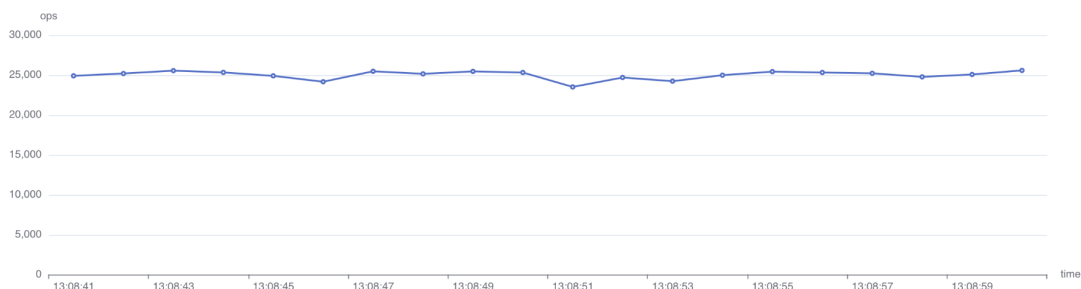
responseFuture + whenAsyncExecutor double check.

Without any measurements overhead on the main branch (831e59b6d48ab682c37710eca971ed73083536ff) the two runs with removed orTimeout and whenCompleteAsync looks like:

Main version (source <https://ward.gridgain.com/tests/654a0cd7165f80acb1d485d9>)



Hot fixed version (source <https://ward.gridgain.com/tests/654a0cdd165f80acb1d485de>)



So, these fixes give us up to 1.6x performance increase.

KV get queries after hot fixes

```
kvGetMark 0.025 3295366 3295391
    Here is hidden 0.125 us
schemaVersionAt 0.264 3295516 3295780
    Here is hidden 0.087 us
keyMarshaller 0.223 3295867 3296090
    Here is hidden 0.055 us
valueMarshaller 0.039 3296145 3296184
    Here is hidden 0.041 us
buildRow 0.355 3296225 3296580
    Here is hidden 0.057 us
beginTransaction 0.476 3296637 3297113
    Here is hidden 0.178 us
awaitPrimaryReplica 0.283 3297291 3297574
    Here is hidden 0.145 us
responseFuture 0.027 3297719 3297746
    Here is hidden 0.541 us
getPrimaryReplica 0.146 3298287 3298433
    Here is hidden 1.189 us
resolvePk 0.059 3299622 3299681
    Here is hidden 0.054 us
indexGet 0.197 3299735 3299932
    Here is hidden 0.057 us
hasNextPageMemoryIndexStorage 1.472 3299989 3301461
    Here is hidden 0.098 us
readPageMemory 1.481 3301559 3303040
    Here is hidden 0.077 us
hasNextPageMemoryIndexStorage 0.936 3303117 3304053
    Here is hidden 0.609 us
HybridClock#update 0.232 3304662 3304894
    Here is hidden 0.125 us
postEvaluate 0.325 3305019 3305344
    Here is hidden 0.079 us
unmarshalValue 0.157 3305423 3305580
    Here is hidden 0.175 us
kvGetEndMark 0.026 3305755 3305781
Whole time: 10.498 us
Found time: 6723 Not found time: 3775.0 Percentage of found: 64.04076967041341
```

KV get queries spikes

We have one more issue with readPageMemory part: from time to time [here](#) wastes significant amount of time: 20-30us instead of 2-4us (look at the attached file from the end, because it is already warmed up state). Issue <https://ggsystems.atlassian.net/browse/GG-37880>

KV put performance investigations.

According to InsertBenchmark.kvInsert (These numbers rechecked on TC runs also with JFR mode. In general, in the most cases numbers changes proportionally from run to run on TC and on local runs.

```
kvPutMark 0.025 3021894 3021919
    Here is hidden 0.272 us
schemaVersionAt 0.259 3022191 3022450
    Here is hidden 0.074 us
keyMarshaller 0.339 3022524 3022863
    Here is hidden 0.043 us
valueMarshaller 2.454 3022906 3025360
    Here is hidden 0.069 us
buildRow 5.551 3025429 3030980
    Here is hidden 0.067 us
beginTransaction 0.653 3031047 3031700
    Here is hidden 0.253 us
awaitPrimaryReplica 0.438 3031953 3032391
    Here is hidden 0.502 us
responseFuture 0.027 3032893 3032920
    Here is hidden 0.462 us
HybridClock#update 0.426 3033382 3033808
    Here is hidden 0.154 us
getPrimaryReplica 0.172 3033962 3034134
    Here is hidden 0.924 us
awaitIndexes 0.211 3035058 3035269
    Here is hidden 1.074 us
indexGet 0.289 3036343 3036632
    Here is hidden 0.068 us
hasNextPageMemoryIndexStorage 2.403 3036700 3039103
    Here is hidden 1.013 us
awaitIndexes 0.097 3040116 3040213
    Here is hidden 0.964 us
validateWriteAgainstSchemaAfterTakingLocks 0.306 3041177 3041483
```

Here is hidden 0.078 us
awaitCleanup 0.07 3041561 3041631
Here is hidden 0.375 us
marshall 0.775 3042006 3042781
Here is hidden 0.394 us
responseFuture 0.032 3043175 3043207
Here is hidden 0.296 us
executorDelay 9.174 3043503 3052677
Here is hidden 0.534 us
intercept 0.134 3053211 3053345
Here is hidden 1.795 us
appendEntries 0.718 3055140 3055858
Here is hidden 0.713 us
StripeAwareAppendBatcher#appendToStorage 1.274 3056571 3057845
Here is hidden 0.155 us
StripeAwareAppendBatcher#commitWriteBatch 4.389 3058000 3062389
Here is hidden 0.119 us
StripeAwareAppendBatcher#notifyClosures 0.864 3062508 3063372
Here is hidden 2.066 us
awaitIndexes 0.279 3065438 3065717
Here is hidden 1.245 us
performStorageCleanupIfNeeded 0.029 3066962 3066991
Here is hidden 0.128 us
addWriteCommitted 7.698 3067119 3074817
Here is hidden 0.066 us
awaitIndexes 0.129 3074883 3075012
Here is hidden 0.383 us
extractColumns 0.268 3075395 3075663
Here is hidden 0.049 us
putIndex 3.913 3075712 3079625
Here is hidden 0.074 us
lastApplied 0.332 3079699 3080031
Here is hidden 0.295 us
executeBatchGc 0.048 3080326 3080374
Here is hidden 0.066 us
updateTrackerIgnoringTrackerClosedException 0.178 3080440 3080618
Here is hidden 1.855 us
awaitIndexes 0.128 3082473 3082601
Here is hidden 0.635 us
performStorageCleanupIfNeeded 0.031 3083236 3083267
Here is hidden 0.058 us
addWriteCommitted 5.965 3083325 3089290
Here is hidden 0.083 us
awaitIndexes 0.194 3089373 3089567

Here is hidden 0.33 us
extractColumns 0.134 3089897 3090031
Here is hidden 0.074 us
putIndex 1.373 3090105 3091478
Here is hidden 0.329 us
executeBatchGc 0.046 3091807 3091853
Here is hidden 2.946 us
HybridClock#update 0.519 3094799 3095318
Here is hidden 0.433 us
finishFull 0.327 3095751 3096078
Here is hidden 5.421 us
updateTrackerIgnoringTrackerClosedException 0.118 3101499 3101617
Here is hidden 0.067 us
updateTrackerIgnoringTrackerClosedException 0.214 3101684 3101898
Here is hidden 2.625 us
kvPutEndMark 0.027 3104523 3104550
Whole time: 83.023 us
Found time: 53030 Not found time: 29993.0 Percentage of found: 63.87386627801934

Known leaders in timeline:

- [buildRow](#) 5.551us
- [hasNextPageMemoryIndexStorage](#) 2.403us
- [executorDelay](#) 9.174us
- [StripeAwareAppendBatcher#commitWriteBatch](#) 4.389us
- [addWriteCommitted](#) 7.698us
- [addWriteCommitted](#) 5.965us
- [putIndex](#) 3.913us