Database - Bug #7138

Short circuit fake-update (update with a new value eqaul to the old value) in H2

02/22/2023 08:50 AM - Alexandru Lungu

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

History

#1 - 02/22/2023 08:57 AM - Alexandru Lungu

I encountered several fake-updates in large customer applications. A fake-update is something like UPDATE tt SET f1 = 2 WHERE f1 = 2. It is not necessarily that the WHERE clause should match the SET clause; only that the old row value should be equal to the new row value.

In the latest FWD-H2, we reconstruct indexes **only** if indexed fields are updated. However, if we fake-update an indexed field, we may end up reconstructing the index. H2 specifically removes the old row and adds a new row to emulate modification. This doesn't make sense if the old value is the same as the new value. The only side-effect I see is the corruption of delta tables (SELECT * FROM OLD TABLE (UPDATE TEST SET B = 3 WHERE ID = 1);) and triggers, but this won't affect FWD.

As part of the work in #7061, the lazy keyword has some issues with there fake-updates. In fact, lazy will skip all equal records from an index if such fake-update happens (in fact, this happens in way more specific case).

#2 - 02/22/2023 08:13 PM - Ovidiu Maxiniuc

These should not occur since we create the SET part of statement based only on changed properties (BaseRecord.dirtyProps).

However, changing back a value of a field before it had the chance to be flushed could possibly generate the fake-updates (by marking the property as dirty but saving the exact same value). This could be theoretical double checked in Persister.update() but would add a performance penalty (ignoring the dmo.getDirtyProps() and recomparing the properties again). It also looks a bit stupid, unless we rename it to getPossiblyDirtyProps. This should be done **only** if the performance gain in H2 big enough to overcome this extra work.

#3 - 02/23/2023 09:39 AM - Alexandru Lungu

I recall a scenario in which some records were updated in a for-loop and their new values were the same as the old values. I didn't dig it up too much; I will try to search for that example.

#4 - 03/14/2023 05:10 AM - Alexandru Lungu

```
define temp-table tt field f1 as character.

create tt.
tt.f1 = "abc".
```

```
for each tt:
    tt.f1 = "abc". // fake-update
    message tt.f1.
end.
```

I didn't expect to be such a simple case. I have a large customer application which has several such constructs: tt.path = replace(tt.path, "~\":U, "/":U)., in which tt.f1 doesn't actually have a ~\, so this is a fake-update. I am concerned because of:

- SQL parsing, cache pollution, index look-up, etc.: update tt3 set f1=? where recid=? and _multiplex=?
- Index reconstruct and logging: remove row, add row, register update to transaction undo log
- 4GL FOR-EACH behavior: the query is not "invalidated", because in 4GL, the update is not done. In FWD, the query is invalidated because of the index reconstruction. We have a weak solution for this, reconstructing the index cursor based on the old value. However, if this is a fake-update, the index cursor reconstruction may skip some records equal to the updated one (because it is unware of which record was stored before invalidation).

If we are to consider a solution at the FWD level, we can cover fake-update in persistent databases as well. If we want to target H2, we can recover some time from index and cursor reconstruction by checking if the updated indexed field is the same.

I agree that the equality check itself is time consuming (especially for large strings), but I think the string hashing is good enough. My point is that:

- if string hashes don't match, let the engine do the fake-update (this is the usual case and is not that time consuming - or is it?)
- if the string hashes match, do a full string equality check
    - if the strings match, avoid doing the update. This recovers a lot of time. Reconstructing an index means several string equality checks, doing only one to prevent it is ideal.
    - if the strings don't match, let the engine do the fake-update (even if this is an overhead, this is very rare)
- maybe we can do some prerequisite checks to avoid hash computation: do reference equality check, check sizes, check first 2 and last 2 characters (this can rule out some cases very fast)

This technique is classic and may fit our intention very well. My only concern is that the hash computation of strings is time consuming, but it is usually cached (internally in String).

#5 - 03/14/2023 08:46 AM - Ovidiu Maxiniuc

H2 database engine is built in Java. My first reaction to your hash suggestion was that it unneeded since Java has the hashing thing already built-in. More than that, since the String is an immutable class, the hash can be computed at first usage, stored and used when a equality test is needed, to quickly determine when two String objects are different.

Then I open the String class (for OpenJDK 1.8) and I found:

```
    public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        if (anObject instanceof String) {
            String anotherString = (String)anObject;
            int n = value.length;
            if (n == anotherString.value.length) {
                char v1[] = value;
                char v2[] = anotherString.value;
                int i = 0;
                while (n-- != 0) {
                    if (v1[i] != v2[i])
                        return false;
                    i++;
                }
                return true;
            }
        }
        return false;
    }
```

Apparently, the hash short-circuit is not used. The length of the strings are compared, but not the hashes. I really did not expect this.

So the hash check is probably going to add a bit of performance improvement, unless the Strings were already intern -ed, in which case the equals will return with positive result from the first line.

OTOH, I think we are interested mostly on the case insensitive case, which is the feature of 4GL we are trying to emulate for basic character data. Unfortunately, the hash will not work here as expected. At least with standard hash of the String. To avoid this, we can use the hash of the lowercased instance of the string instead.

#6 - 03/14/2023 09:42 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> Apparently, the hash short-circuit is not used. The length of the strings are compared, but not the hashes. I really did not expect this.

I had the same expectation. String.compareTo is even more shocking, as there is no equality rule-out.

> OTOH, I think we are interested mostly on the case insensitive case, which is the feature of 4GL we are trying to emulate for basic character data. Unfortunately, the hash will not work here as expected. At least with standard hash of the String. To avoid this, we can use the hash of the lowercased instance of the string instead.

The case-sensitivity is important. Doing something like update tt set f1 = 'abC' where f1 = 'aBc' is not a "fake-update"; I expect to retrieve abC afterwards. I don't think we should suppress such update, even if f1 is case-insensitive and matches the old value. I agree however on the fact that its position inside the index doesn't change, so there is no need to reconstruct. But this is a fine tune IMO, as I expect most of the "fake-updates" to have the same case as before.

Right now, I am still dragged back by String.hashCode:

```
  public int hashCode() {
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;

            for (int i = 0; i < value.length; i++) {
                h = 31 * h + val[i];
            }
            hash = h;
        }
        return h;
    }
```

Worst case scenario, a string is not interned, so we have to compute its hash code. If we do such iteration anyways, we could have done the equality check instead at the same cost. My solution heavily relied on the fact that String hash code computation is faster and better cached. If we know that we already have computed the hash-code in a previous stage, we are good to go. But I don't think that this is the case. For an update, I don't think we ever compute the hash code in a previous step.

#7 - 03/14/2023 10:32 AM - Ovidiu Maxiniuc

Alexandru Lungu wrote:

> Right now, I am still dragged back by String.hashCode:
> [...]

Why is that? It seems like the usual lazy computation of hash private field. Maybe the function is not dispersing the string values the best across the full 32 bit key space but this is a quite standard, fast approach.

> Worst case scenario, a string is not interned, so we have to compute its hash code. If we do such iteration anyways, we could have done the equality check instead at the same cost. My solution heavily relied on the fact that String hash code computation is faster and better cached. If we know that we already have computed the hash-code in a previous stage, we are good to go. But I don't think that this is the case. For an update, I don't think we ever compute the hash code in a previous step.

At the moment the String value reaches the database level it is probably have been involved in some computations and the chances are that the has already been calculated. Usually, the String values are rarely equals (think of unique string indices) so the 'fake' updates are not common. Detecting the difference with a simple integer compare operation would be faster in those cases. But if the ABL code does this repeatedly or the values are long enough, the improvement might be visible.

#8 - 03/14/2023 10:50 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> Alexandru Lungu wrote:
>
> > Right now, I am still dragged back by String.hashCode:
> > [...]
>
> Why is that? It seems like the usual lazy computation of hash private field. Maybe the function is not dispersing the string values the best across the full 32 bit key space but this is a quite standard, fast approach.

Mostly because it does a full string traversal. I agree that there is no faster way to do it, but we are to compute a hash-code **for each** update we are going to make in the database. Also, a simple integer compare is fast, but to achieve that integer, you have to do a full string traversal (maybe tens or even hundreds of characters?). I also expect most of the strings to be encountered for the first time as they are not interned, so they don't have a precomputed hash-code. This whole effort just to check if we hit that slim number of cases of an fake-update.

> At the moment the String value reaches the database level it is probably have been involved in some computations and the chances are that the has already been calculated. Usually, the String values are rarely equals (think of unique string indices) so the 'fake' updates are not common.

Right. I will do some tests to check how many such fake-updates are actually in some large customer applications. I will also check how much the String.hashCode call takes on average in a real scenario.

#9 - 03/28/2023 04:27 AM - Alexandru Lungu

- Status changed from New to WIP

- Assignee set to Dănuț Filimon

#10 - 04/11/2023 07:17 AM - Dănuț Filimon

Committed 7138a_h2/rev.16. Made changes to should-circuit fake-updates. TestCompatibility change is related to #7108-41. Made build.sh executable.

A fake-update is detected using the Value.compareTypeSafe method, which compares values of the same type. I also keep track of the number of updated non-indexed/indexed columns, this helped to remove the inMemUpdate usage and replace it with another condition related to the number of columns updated. We want to make an update in memory when we handle non-indexed columns

There are some cases were fake-update will not be handled:

- the old value is **null** and it is updated with **null**
- the old value is Value.Enum
- the value types of the old and new value are different

For the last mentioned case, Table.validateConvertUpdateSequence is called after checking the columns that need to be updated and eventually leads to the value conversion:

```
    CREATE TABLE test (f1 VARCHAR(8)); // field values will be ValueString
    INSERT INTO test (f1) VALUES('21');
    UPDATE test SET f1 = 21; // new value is ValueInt
```

There were a few cases where tests would fail because they would not work with fake-update:

- TestLinkedTable: A table link can be created for a table and any operation done on the table link is reflected in the original table. The test runs 3 iterations using different table links in different modes and each iteration executes a delete, insert and update **using the same values**. While updating the value for a **read only** table link, the fake-update is discovered and the method returns early without throwing an error for a read only table. The test was changed to use different values at each iteration.
- TestStatement: Related to the MERGE statement. It would try to merge the same value twice and would fail since a fake-update is discovered.

#11 - 04/12/2023 10:17 AM - Alexandru Lungu

- % Done changed from 0 to 70

#12 - 04/19/2023 07:09 AM - Alexandru Lungu

Please address this review on **7138a_h2/rev.16** asap. Some of the points here are crucial for the performance tests to make sense.

- You have oldValue.getValueType() != Value.NULL twice in the fake-update conditional.
- You can extract getValueType answer in separate variable to avoid multiple calls to the same function.
- You use compareTypeSafe to check if the values are not in fact equal. Maybe it is faster to simply use equals instead, we already know that they have the same type?
- table.hasIndexedColumn is way faster than table.getIndexForColumn. Refactor to use hasIndexedColumn.
- I think it better to move the inMemRows and notInMemRows resets in the finally block. If an error occurs, these won't be reset properly.

We will take in consideration the hashing tactic discussed in #7138-6 afterwards, if still needed.

#13 - 04/19/2023 09:43 AM - Dănuț Filimon

Alexandru Lungu wrote:

Please address this review on **7138a_h2/rev.16** asap. Some of the points here are crucial for the performance tests to make sense.

- You have oldValue.getValueType() != Value.NULL twice in the fake-update conditional.

The last one should've been Value.ENUM, I didn't notice it.

- You can extract getValueType answer in separate variable to avoid multiple calls to the same function.
- You use compareTypeSafe to check if the values are not in fact equal. Maybe it is faster to simply use equals instead, we already know that they have the same type?

Using equals should be faster, there are some instances where compareTypeSafe is used by equals, but on the lower side (e.g. ValueDouble, ValueFloat). In this case, the additional check for oldValue.getValueType() != Value.ENUM will not be needed since ENUM comparison is easier when using equals.

- table.hasIndexedColumn is way faster than table.getIndexForColumn. Refactor to use hasIndexedColumn.
- I think it better to move the inMemRows and notInMemRows resets in the finally block. If an error occurs, these won't be reset properly.

Indeed.

Committed 7138a_h2/rev.17. I applied the changes mentioned in #7138-12. Please go ahead with the performance tests.

#14 - 04/24/2023 06:46 AM - Alexandru Lungu

Review to 7138a_h2/rev.17 : I am OK with the changes.

Danut, please re-test 7138a_h2 (own test-cases, large customer application, FWD-H2 regression testing) and let me know if everything is OK. Please fix/remove in 7138a_h2 the regression tests which fail due to your latest changes; AFAIK they address very specific cases not beneficial to FWD.

Once finished, let me know. I will start the testing/profiling routine.

#15 - 04/25/2023 09:44 AM - Dănuț Filimon

I tested Hotel_GUI and found no problems, but I found a regression in a customer application which I didn't manage to point out yet. I am still looking into it.

#16 - 04/26/2023 05:57 AM - Alexandru Lungu

- Status changed from WIP to Review

- % Done changed from 70 to 100

Fixed a regression in 7138a_h2 regarding updating indexed computed columns in inMemUpdate mode. If you have (f1, __if1) fields, where f1 is not-indexed and __if1 is computed (upper(f1)) and indexed, and update f1, than this should be a non-in-mem-update, because changing f1 alters __if1 (so changing a non-indexed field causes non-in-mem-update). This was handled previously, but I guess you removed it in the meantime.

Danut, please retest your application. Also, please use the performance test from #6928-3 and others you can make. I want to see a performance improvement in a dummy test. Also, please run the regression tests against FWD-H2.
I will also run some performance tests.

#17 - 04/26/2023 09:16 AM - Dănuț Filimon

I've tested #6928-3 and also made my own set of performance tests (results are measured in milliseconds). The results from the #6928-3 performance test were:

Version	71381_h2.rev15	7138a_h2.rev18	Difference
indexed	369	334.2	-9.43%
non-indexed	85.8	96	+11.888%

I also made a few specific fake-update tests (10000 initial creates for each one):

Version	7138a_h2.rev 15	7138a_h2.rev 18	Difference
update both fields	1571.4	1608.8	+2.38%
fake-update indexed field	865.4	801	-7.441%
fake-update non-indexed field	712.6	823.8	+15.604%
fake-update both fields	445.4	400.8	-10.013%

I obtained good and bad results. I will continue to expand the tests and share my findings. Please let me know of any specific tests that should be done.

#18 - 05/02/2023 07:26 AM - Dănuț Filimon

I've tested fake-update performance for larger tables and extent fields, results of each test are measured in milliseconds and consist of the average obtained after running it 5 times.

Test 1: Temporary tables with different DMOs, 10 non-indexed fields and 10 indexed.

Version	7138a_h2.rev15	7138a_h2.rev18	Difference
update all fields (cold test)	4134.4	4037.2	-2.351%

fake-update non-indexed field	3022.2	3146	+4.096%
fake-update indexed field	2546.2	2227.4	-12.52%
fake-update all fields	2478.8	1958.4	-20.994%

Test 2: Temporary tables with the same DMO (different multiplex), 10 non-indexed fields and 10 indexed.

Version	7138a_h2.rev15	7138a_h2.rev18	Difference
update all fields (cold test)	4099.2	4131.4	+0.785%
fake-update non-indexed field	2617	2540.8	-2.911%
fake-update indexed field	2451.8	1773.2	-27.677%
fake-update all fields	2480.6	1731.2	-30.21%

Test 3: Temporary tables with the different DMOs, 10 indexed fields and 1 field with EXTENT 10.

Version	7138a_h2.rev15	7138a_h2.rev18	Difference
update all fields (cold test)	4590	4598.4	+0.183%
fake-update non-indexed field	3633	3686.8	+1.48%
fake-update indexed field	3167.8	2598.8	-17.961%
fake-update all fields	2974.6	2209.8	-25.711%

Test 4: Temporary tables with the same DMO (different multiplex), 10 indexed fields and 1 field with EXTENT 10.

Version	7138a_h2.rev15	7138a_h2.rev18	Difference
update all fields (cold test)	4482.2	4707.4	+5.024%
fake-update non-indexed field	3034	3038	+0.131%
fake-update indexed field	2828	2201	-22.171%
fake-update all fields	2784	2120.8	-23.821%

In all cases, fake-update of indexed fields or of all fields shows an improvement, while updating all fields or fake-update of non-indexed fields show little to no change (<= 5%). Should I do similar tests for persistent tables?

UPDATE: Corrected results that were accidentally swapped. Tests need to be redone due to update statements being executed for each field.

#19 - 05/03/2023 02:33 PM - Eric Faulhaber

Dănuț Filimon wrote:

> Should I do similar tests for persistent tables?

As a lower priority, yes, just to make sure there isn't a regression when used in that mode.

We don't use H2 for persistent tables in production, so performance of persistent tables in H2 is not as important for us as embedded, in-memory use. However, we do use it for incremental conversion support, and occasionally for development/testing purposes. So, we will want to know if there is a regression. However, this applies to all H2 changes, not just this specific one.

#20 - 05/04/2023 03:24 AM - Alexandru Lungu

From the testing POV, we already covered several large applications for in-memory H2.
Danut:

- you can do a quick reconversion of Hotel GUI and use H2 persistent database just to check for regressions. AFAIK FWD-H2 regression testing is OK.
- please compute a ratio of how many fake-updates are happening in a large customer application: this is for reference only.

I am doing a final profiling round over a POC.

#21 - 05/04/2023 04:15 AM - Dănuț Filimon

Committed 7138a_h2/rev.21. Fixed a concurrency problem. inMemUpdate and notInMemUpdate need to be set to **null** when database is in persistent mode.

#22 - 05/05/2023 05:30 AM - Dănuț Filimon

Alexandru Lungu wrote:

> - you can do a quick reconversion of Hotel GUI and use H2 persistent database just to check for regressions. AFAIK FWD-H2 regression testing is OK.

Hotel GUI has no regressions with H2 persistent database.

> - please compute a ratio of how many fake-updates are happening in a large customer application: this is for reference only.

I updated and tested a customer application (~35 minutes) and from a total of **1352516** update statements I calculated the following results:

Update Type	Count	Percentage from total	Real Columns	Fake Columns	Percentage of fake columns from total

Updates made on both indexed and non-indexed columns	3236	0.2392%	11538	15365	57.1125%
Updates made only on indexed columns	36200	2.6764%	37695	953	2.4658%
Updates made only on non-indexed columns	1279157	94.5761%	1697223	198844	10.4871%
Fake updates	33923	2.5081%	0	58264	100.0%

Even thought updates made on both types of columns represent ~0.24%, more than half of the field updates are fake-updates, while updates made only on non-indexed columns have ~10% fake-update cases.

I ran separate tests and used **VisualVM** to track the total time of certain operations. The tests consist of a temporary table with 10 indexed fields and 10 non-indexed fields (all 20 are integers), it starts with an initial create of 1000 items (for 10000 it takes around ~9 minutes to create a snapshot).

Fake-update both type of fields:

Test 1	7138a_h2/rev.15 (ms)	Invocation count (1)	7138a_h2/rev.18 (ms)	Invocation count (2)	Difference (ms%)
Update.update()	7150.0	20000	7323.0	20000	+2.419%
PageStoreTable.addRow()	3388.0	11987	3347.0	10986	-1.21%
PageStoreTable.removeRow()	3135.0	11053	3043.0	10053	-2.934%
ValueInt.equals()	0	0	5.88	39000	+

Fake-update indexed fields, update non-indexed fields:

Test 2	7138a_h2/rev.15 (ms)	Invocation count (1)	7138a_h2/rev.18 (ms)	Invocation count (2)	Difference (ms%)
Update.update()	6431.0	20000	6282.0	20000	-2.316%
PageStoreTable.addRow()	3032.0	11987	2879.0	10986	-5.046%
PageStoreTable.removeRow()	2774.0	11053	2595.0	10053	-6.452%
ValueInt.equals()	0	0	5.4	39000	+

Only fake-update indexed fields:

Test 3	7138a_h2/rev.15 (ms)	Invocation count (1)	7138a_h2/rev.18 (ms)	Invocation count (2)	Difference (ms%)
Update.update()	7603.0	20000	5789.0	20000	-23.859%
PageStoreTable.addRow()	3592.0	11986	2682.0	10987	-25.334%
PageStoreTable.removeRow()	3385.0	11053	2386.0	10050	-29.512%
ValueInt.equals()	0	0	3.54	29000	+

Update both types of fields:

Test 4	7138a_h2/rev.15 (ms)	Invocation count (1)	7138a_h2/rev.18 (ms)	Invocation count (2)	Difference (ms%)
Update.update()	6655.0	20000	6213.0	20000	-6.641%
PageStoreTable.addRow()	3143.0	11986	2839.0	10987	-9.672%
PageStoreTable.removeRow()	2948.0	11053	2604.0	10053	-11.668%
ValueInt.equals()	0	0	4.97	39000	+

The test originally used integer fields. If necessary, I will also do the same tests using string fields and 2500-5000 initial creates.

#23 - 05/05/2023 09:47 AM - Alexandru Lungu

Tested with a large customer application and I get similar results: between -0.1% and +0.1%. I will retest on a newer binaries set.

Danut, please let me know if you have any further ideas to optimize this. I am pleased that there are a considerable number of fake-updates we actually handle (in your example, at least 10% of the columns).
Also, can you take a look into the "null" cases? You rule out the nulls entirely when checking for fake update. I wonder if we can increase the ratio if we also take nulls into account. For that matter, please re-run your test checking how many updates work with null:

- update a non-null value with null
- update a null value with non-null
- fake-update with null

Note that by null, I mean NullValue.

#24 - 05/08/2023 04:31 AM - Dănuț Filimon

I retested the application and counted updates that use null, from a total of **1352885** I obtained the following:

Type	Count	% from total
update a non-null value with ValueNull	24254	1.7927%
update ValueNull with non-null	60934	4.504%
fake-update with ValueNull	13650	1.0089%

After looking at the code, I remember that I excluded Value.NULL because I was using compareTypeSafe which throws an exception when comparing null values, but this does not happen now because the comparison is done using equals. It is safe to remove the type check for Value.NULL and let it be handled as a fake-update when necessary.

#25 - 05/08/2023 04:32 AM - Alexandru Lungu

Dănuț Filimon wrote:

> After looking at the code, I remember that I excluded Value.NULL because I was using compareTypeSafe which throws an exception when comparing null values, but this does not happen now because the comparison is done using equals. It is safe to remove the type check for

> Value.NULL and let it be handled as a fake-update when necessary.

Please go ahead with this change. Let the null checks: a != null & b != null, but remove the NullValue checks.

#26 - 05/08/2023 04:51 AM - Dănuț Filimon

Committed 7138a_h2/rev.22. Allowed ValueNull values to be compared.

#27 - 06/30/2023 04:23 AM - Alexandru Lungu

- Status changed from Review to WIP

- % Done changed from 100 to 80

I've double checked how many fake-updates are in another customer application and indeed I've got only to 1%. The performance check is unstable: the improvement from avoiding index updates are counter-balanced from the overhead of value comparison for all the updated fields.

It is quite an overhead to compare the values for **all** updated fields just to end up with 1% fake updates. I guess we shall restrict this fake-update checking only for indexed fields, as fake-update on non-indexed fields are not that costly.

Danut, I rebased 7138a_h2 from FWD-H2 trunk (rev. 23). Please commit a patch to avoid fake-updates only on indexed fields. I will retest with your changes and #7454 (which will make the comparison faster eliminating the value casts).

At that point we can decide if this mechanism of avoiding fake-updates is good to go or not.

#28 - 07/03/2023 03:17 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Danut, I rebased 7138a_h2 from FWD-H2 trunk (rev. 23). Please commit a patch to avoid fake-updates only on indexed fields. I will retest with your changes and #7454 (which will make the comparison faster eliminating the value casts).

Committed 7138a_h2/rev.29. Only indexed columns will be checked for fake-updates.

#29 - 11/28/2023 08:35 AM - Dănuț Filimon

- Status changed from WIP to Review

Branch 7454a was previously merged into trunk as rev. 14833. Alexandru, should I run the POC tests for this task before continuing?

#30 - 11/28/2023 10:19 AM - Alexandru Lungu

Yes, please go ahead.

#31 - 11/29/2023 05:37 AM - Dănuț Filimon

After running a large POC with **7156b/rev.14846** (which uses **FWD-H2/rev.33**) and the changes from **7138a_h2/rev.24 - 29**. From 5 runs each (baseline and 7156b with 7138a_h2) I did not obtain an improvement for the **average of the last 20 runs** (the result was **+0.065%**) and ~**1%** improvement for the **total average of the 100 runs**.

#32 - 12/22/2023 10:42 AM - Greg Shah

What is the status of 7138a_h2? Does it need more review? More testing? It seems to provide a decent improvement on the total runs.

#33 - 01/05/2024 04:01 AM - Alexandru Lungu

- Status changed from Review to Internal Test

- % Done changed from 80 to 100

Greg Shah wrote:

> What is the status of 7138a_h2? Does it need more review? More testing? It seems to provide a decent improvement on the total runs.

I missed Danut's profiling results in #7138-31. It seems indeed that the improvement here looks good. Danut, please rebase 7138a_h2 to the latest FWD-H2 trunk. Run against:

- large customer regression tests
- profile POC (you already did it; I just want to ensure that your results are consistent)
- run smoke tests on another customer application
- run ChUI regression tests if possible.

#34 - 01/08/2024 08:45 AM - Dănuț Filimon

I updated the sources for the customer applications but there are some regressions/files missing which I'll need to look into. The POC needs to be reconverted with the latest version of 7156b. I am currently running ChUI regression tests since those are currently the only possible ones that can be done immediately.

#35 - 01/08/2024 09:10 AM - Alexandru Lungu

Please let the application convert over night. After, we can go ahead with proper testing.

#36 - 01/08/2024 09:13 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Please let the application convert over night. After, we can go ahead with proper testing.

That's my intention, confirming that the POC results are close to the ones from #7138-31 is a priority.

#37 - 01/09/2024 04:20 AM - Dănuț Filimon

An update on the current test status:

- Rebased **7138a_h2** with **fwd-h2/rev.38**. **7138a_h2** is now at revision **45**;
- ChUI regression tests passed;
- Reconverted POC project with **7156b/rev.14932** and will start redoing the baseline and testing 7138a_h2;
- A few issues in a large customer application were solved in the latest version of trunk, so I'll go ahead and do a quick smoke test before continuing with POC.

#38 - 01/11/2024 03:52 AM - Dănuț Filimon

There are currently problems with the POC project and the performance tests will be put on **hold** until further notice. I also successfully smoke tested two customer applications and found no problems so the POC is the only one remaining on the list.

#39 - 01/12/2024 09:49 AM - Dănuț Filimon

I tested POC with 7156b/rev.14933 (which uses FWD-H2/rev.38) and 7138a_h2/rev.45. From 5 runs each (baseline and 7156b with 7138a_h2) I obtained a decrease in performance in both cases, **1.087%** for the average of the last 20 runs and **0.786%** for the total average.

I don't think the results are spoiled, the only difference is that I had to redo the baseline since the last test from #7138-31 and that I am testing on another device. I want to make sure that we can move on with the changes, so I'll also run the tests on the original device.

#40 - 01/15/2024 02:29 AM - Alexandru Lungu

Danut, this is a common pattern: in different days and on different platforms to see different results. The only way to ensure you have a proper conclusion is to continue testing until the times actually converge to a fixed point. Ensure you are always on the same device (first and foremost), ensure there are no background tasks and *re-import" database when changing branches (it works faster on fresh database).

#41 - 01/15/2024 08:48 AM - Dănuț Filimon

I retested POC on the original device and obtained a performance improvement for both the average of the last 20 runs (**4.77%**) and the total average (**2.28%**). Regression tests were done (#7138-37) and there were no issues, can we go ahead and merge 7138a_h2?

#42 - 01/15/2024 08:54 AM - Greg Shah

Alexandru: If you are OK with this, it can merge now.

#43 - 01/15/2024 09:27 AM - Alexandru Lungu

Danut, please go ahead and merge to FWD-H2 trunk.
Greg, this will reach FWD on the next H2 upgrade.

#44 - 01/16/2024 08:26 AM - Dănuț Filimon

- Status changed from Internal Test to Test

Committed 7138a_h2 to FWD_H2 trunk as rev.39. Archived 7138a_h2.

#45 - 01/19/2024 01:43 AM - Eric Faulhaber

- Status changed from Test to Closed

I am closing this. Let me know if there is a reason it should remain open. We have done our internal testing and there is not really a separate customer testing step. If any regression is found, it will be treated as a bug.