Database - Bug #7108

Simulate upper/rtrim directly in H2 using case specific columns

02/09/2023 10:42 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

Related issues:

Related to Database - Bug #6837: prevent unnecessary upper/rtrim injection in...	**WIP**
Related to Database - Bug #7220: Record not created when property is set to e...	**New**
Related to Database - Bug #7352: Mismatching results when comparing case-inse...	**WIP**

History

#1 - 02/09/2023 10:42 AM - Alexandru Lungu

- Related to Bug #6837: prevent unnecessary upper/rtrim injection in queries if the data type doesn't require it added

#2 - 02/09/2023 10:49 AM - Alexandru Lungu

- Status changed from New to WIP

- Assignee set to Dănuț Filimon

Consider supporting case specific operators directly in H2 for character fields. For case-insensitive character fields, FWD does upper(rtrim(field)) to match the 4GL behavior.

To emulate this in H2, we use a computed column (prefixed with __i) which holds the upper-case version of the legacy field. This field is used in indexes to match the upper(rtrim(field)) necessity. The same happens for case-sensitive character fields which are prefixed with __s.

Once we have access to FWD-H2, we can make use of the VARCHAR_IGNORECASE and VARCHAR_CASESENSITIVE H2 data-types, which should match our use-case.

Please consider if such data types are worth using inside indexes without any obvious regressions / performance drops. We should aim on having less SQL overhead (of generating upper and rtrim), less H2 overhead (of computing the upper-case version of each case-insensitive column) and better performance (of comparing strings (?)).

Please relate to #6837 for parallel work over other dialects.

#3 - 02/09/2023 12:53 PM - Ovidiu Maxiniuc

Please test how the right-padding spaces are handled. An important element in the expressions above is the rtrim function. Even if these data types meet the needs for case-sensitiveness the spaces at the end have to be ignored when the records are indexed.

A similar issue we have encountered for PostgreSQL, wehn we attempted to use for same reasons the citext ('case insensitive text') data type. The spaces at the end are not ignored. Wrapping the indexed columns in rtrim() was not useful because the return value of the function was case-sensitive.

So you may want to check whether the rtrim function keep the case-sensitivity of the argument. (I doubt this since H2 does not support overloaded functions). Alternatively, if we do changes at this lower level in H2, maybe we should make sure we are working with character data types which are a match for 4GL's counterparts (that is, to ignore the trailing spaces) an use these instead.

#4 - 02/14/2023 06:18 AM - Dănuț Filimon

- File ffCache-20230214.patch added

I've done some changes to how strings are compared in H2. I included a rtrim parameter that, if set to true (by default), will take the largest string out of the two and check if the rest of the string is strictly consisted of spaces, returning **0** because the strings should be equal in this case. I also encountered a bug in FWD related to this issue which prevented me from testing the changes made to H2. The problem is related with the value retrieval from ffCache, here is a minimal testcase:

```
DEFINE TEMP-TABLE tt2 FIELD key AS CHARACTER CASE-SENSITIVE INDEX idx2 AS UNIQUE PRIMARY KEY.
DEFINE VARIABLE myVar AS CHARACTER.

CREATE tt2.
tt2.key = "def ".
RELEASE tt2.

myVar = "def".
FIND FIRST tt2 WHERE tt2.key = myVar.
MESSAGE "6t" AVAILABLE(tt2).
RELEASE tt2.

myVar = "DeF ".
FIND FIRST tt2 WHERE tt2.key = myVar NO-ERROR.
MESSAGE "7f" AVAILABLE(tt2).
RELEASE tt2.
```

Output should be 6t yes and 7f no, but it's 6t yes and 7f yes. When retrieving the value from ffCache, the parameters **"def"** and **"DeF "** are not case sensitive when compared, which makes it return the same value from the cache. I fixed this issue by making all values that are an instance of character case sensitive. I attached the patch with the solution, please review.

#5 - 02/14/2023 06:29 PM - Ovidiu Maxiniuc

Review of ffCache-20230214.patch.

Nice detective work. Indeed, there seems to be a key collision which causes the incorrect result from cache to be fetched.

The problem is I do not think forcing the character substitutions to be case-sensitive is the solution. For example, if the field to compare with is case insensitive, the values the createKey receives are already uppercased (note that TextOps.toUpperCase() preserves the case-ing). This would lead to other extreme, where if the key was not found initially, it will also not be even if the key is changed. Ideally, the L2Key should be aware of the case-sensitiveness of the field the substitution is used with and perform the lookup based on that.

#6 - 02/15/2023 03:28 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> The problem is I do not think forcing the character substitutions to be case-sensitive is the solution. For example, if the field to compare with is case insensitive, the values the createKey receives are already uppercased (note that TextOps.toUpperCase() preserves the case-ing).

Do you mean the case where the field is case-insensitive and the field is case-sensitive, but upper-cased by the conversion? Like in the following example in 4GL:

```
define temp-table tt field key as char.
def var myVar as char case-sensitive.
create tt.
tt.key = "AbC". // case-insensitive: stored internally as "AbC", can be compared only with "ABC" case-sensitiv
e

myVar = "AbC".
find first tt where tt.key = myVar no-error. // fails because "AbC" != "ABC" as case-sensitive
message available(tt).
myVar = "ABC".
find first tt where tt.key = myVar no-error. // succeeds because "ABC" == "ABC" as case-sensitive
message available(tt).
```

Note that right now the answer is still yes, yes in FWD and no, yes in 4GL. **Unfortunately, this is still not fixed after the patch!**

> This would lead to other extreme, where if the key was not found initially, it will also not be even if the key is changed. Ideally, the L2Key should be aware of the case-sensitiveness of the field the substitution is used with and perform the lookup based on that.

Sorry, I don't understand your case here.

#7 - 02/15/2023 03:53 AM - Alexandru Lungu

Danut, please make a complete set of independent **4GL tests** for each of these cases - or **search the testcases project** for such:

- compare **case-insensitive** database field with **case-sensitive** variable
- compare **case-sensitive** database field with **case-sensitive** variable
- compare **case-insensitive** database field with **case-insensitive** variable
- compare **case-sensitive** database field with **case-insensitive** variable

For each of these scenarios, test:

- case-sensitivity:
  - compare database field "A" with variable "A"
  - compare database field "A" with variable "a"
  - compare database field "a" with variable "A"
  - compare database field "a" with variable "a"
- rtrim:
  - compare database field "a " with variable "a"
  - compare database field "a" with variable "a "
  - compare database field "a " with variable "a "
- other
  - no ltrim: compare " a" with "a", "a" with " a" and " a" with " a"
  - only space: compare 1 space with 2 spaces, 2 spaces with 1 space, empty string with 1 space, 1 space with empty string and empty string with empty string

All of these are for you to get your head around the 4GL case sensitivity and rtrim.
Please make a similar suite to adaptive-scrollling from the testcases project! That is a master procedure which calls of all your testcases (numbered 1 to ...). Group common scenarios into the same file. Make sure you output the results into a file to easily compare to the FWD results. Please report back to the mismatches we currently face in FWD. You can do the tests exclusively for H2 right now.

Right now, it is hard to get straight to H2 VARCHAR_IGNORECASE and VARCHAR_CASESENSITIVE without acknowledging the "4GL rules". You can start off by building this testcase, report the issues we have right now in FWD and eventually try to implement rtrim directly in H2.

#8 - 02/15/2023 09:04 AM - Dănuț Filimon

I found several mismatched between FWD and 4GL. All of the tests are done in the following manner:

```
DEFINE TEMP-TABLE tt1 FIELD key AS CHARACTER INDEX idx1 AS UNIQUE PRIMARY KEY.
DEFINE VARIABLE cVar AS CHARACTER CASE-SENSITIVE.

CREATE tt1.
tt1.key = "A".
RELEASE tt1.

cVar = "a".
```

```
FIND FIRST tt1 WHERE tt1.key = cVar NO-ERROR.
MESSAGE AVAILABLE(tt1).
```

The following test cases had different results from each other:

- compare case-insensitive database field with case-sensitive variable
  - case-sensitivity
    - compare database field 'A' with variable 'a': 4GL no, FWD yes
    - compare database field 'a' with variable 'a': 4GL no, FWD yes
  - rtrim
    - compare database field 'a ' with variable 'a': 4GL no, FWD yes
    - compare database field 'a' with variable 'a ': 4GL no, FWD yes
    - compare database field 'a ' with variable 'a ': 4GL no, FWD yes
  - other, no ltrim
    - compare database field ' a' with variable ' a': 4GL no FWD yes
- compare case-sensitive database field with case-insensitive variable
  - case-sensitivity (this happens only if the FastFindCache is enabled)
    - compare database field 'A' with variable 'a': 4GL no FWD yes
    - compare database field 'a' with variable 'a': 4GL yes, FWD no

#9 - 02/15/2023 03:01 PM - Ovidiu Maxiniuc

Alexandru Lungu wrote:

> Do you mean the case where the field is case-insensitive and the field is case-sensitive, but upper-cased by the conversion? Like in the following example in 4GL:
> [...]
> Note that right now the answer is still yes, yes in FWD and no, yes in 4GL. **Unfortunately, this is still not fixed after the patch!**

Yes. Exactly.

> This would lead to other extreme, where if the key was not found initially, it will also not be even if the key is changed. Ideally, the L2Key should be aware of the case-sensitiveness of the field the substitution is used with and perform the lookup based on that.

> Sorry, I don't understand your case here.

I was talking here about the FastFindCache. I think there is a flaw here: while the original query is generated by FWD conversion to honour the 4GL case-sensitiveness semantics, this may be lost if the FFC come into play. This is because the key (which contains the subst values) should be compared using the case-sensitiveness of field, not the actual values.
For example, if we have two queries with predicates like: where tt.key = my_CS-Var and where tt.key = my_CI-Var, they will use the same FFC key. The result depends on whether tt.key is CS or not.

Dănuț,
Could you please repeat the test using disabled FFC (keep ffCache == null in RandomAccessQuery)? You mentioned something in note-8 but only for CS field and CI subst (variable). Maybe it is worth checking the behaviour for same casing: in your patch you're setting the substitution as CS and I expect issues here.

#10 - 02/16/2023 07:48 AM - Dănuț Filimon

The tests from #7108-8 were done **without any patch applied**. I retested with the patch applied and there is no difference between using ffCache and not using it. The results of this comparison are almost the same as the ones mentioned in #7108-8, except the case that used a case-sensitive database field with case-insensitive variable, which executed correctly.

#11 - 02/20/2023 08:48 AM - Alexandru Lungu

Ovidiu, I recall that you had done some work with locale at some point. Please let me know if you can help me on this one:

I am facing a hot-spot in my profiling that may actual affect #7108 effort as well. It is related to org.h2.value.CompareModeDefault, more exactly I see lots of calls to java.text.Collator.getCollationKey. H2 is doing explicit text comparisons in regard to a specified collation. From CompareModeDefault:

```
   @Override
    public int compareString(String a, String b, boolean ignoreCase) {
        if (ignoreCase) {
            // this is locale sensitive
            a = a.toUpperCase();
            b = b.toUpperCase();
        }
        int comp;
        if (collationKeys != null) {
            CollationKey aKey = getKey(a);
            CollationKey bKey = getKey(b);
            comp = aKey.compareTo(bKey);
        } else {
            comp = collator.compare(a, b);
        }
        return comp;
    }
```

My concern here is:

- Do we actually need this, at least for the H2 embedded mode? I don't have the best understanding of how this is handled internally in FWD, (i.e. collation-specific text comparisons). I know there is a SPI generated to provide several char-sets we actually support. I suspect that we tell the JVM to use specific collators at run-time, so that String.compareTo will do its job under the hood. Please correct me if I am wrong. **My point** is that H2 is doing this kind of work explicitly, using CollationKey for each string to be compared, and maybe we can benefit from our SPI inside FWD-H2.
- Right now, H2 uses two traversals of each string before comparing it with ignoreCase. First traversal makes the string upper-case and the second compares using the Collator. There is no "compareIgnoreCase" for Collator. Do we have a workaround for this, integrating ignoreCase into the collator comparison?
- Doing rtrim means removing the right spaces before comparison. However, manually right trimming and comparing means two string traversals. This means that the text indexed columns in H2 will be right trimmed at each comparison and this is clearly worse than keeping a rtrimed version of each indexed char column (computed column).
- Can we ever consider a char-by-char comparison of legacy character data? Or are we always forced to make use of the Collator, so we can't integrate ignoreCase and rtrim in a single string traversal?

If we are tightly bound on using Collator for the H2 text comparisons, then I can't see how a H2 native rtrim may be faster than a computed column using a rtrim defined function.

#12 - 02/20/2023 02:24 PM - Ovidiu Maxiniuc

Dănuț Filimon wrote:

> The tests from #7108-8 were done **without any patch applied**. I retested with the patch applied and there is no difference between using ffCache and not using it.

OK, the idea was to understand whether the issue is caused by the FFC, that is, if the FFC keys work correctly.

> The results of this comparison are almost the same as the ones mentioned in #7108-8, except the case that used a case-sensitive database field with case-insensitive variable, which executed correctly.

This means the issue is at database level, so a hidden flaw. However, it is really strange that altering the keys in FFC fixes the result returned by the query.

I looked again at your results in #7108-8. They do not seem alright. There are several cases like:

- compare database field 'a' with variable 'a': 4GL **no**, FWD yes

Regardless of the case-sensitiveness of the field and variable, the result of this should be positive in 4GL! They are the same value: a single lower-case a character.

#13 - 02/20/2023 03:45 PM - Ovidiu Maxiniuc

Alexandru Lungu wrote:

> I am facing a hot-spot in my profiling that may actual affect #7108 effort as well. It is related to org.h2.value.CompareModeDefault, more exactly I see lots of calls to java.text.Collator.getCollationKey. H2 is doing explicit text comparisons in regard to a specified collation. From CompareModeDefault:
> [...]
> My concern here is:
>
> - Do we actually need this, at least for the H2 embedded mode? I don't have the best understanding of how this is handled internally in FWD, (i.e. collation-specific text comparisons). I know there is a SPI generated to provide several char-sets we actually support. I suspect that we tell the JVM to use specific collators at run-time, so that String.compareTo will do its job under the hood. Please correct me if I am wrong. **My point** is that H2 is doing this kind of work explicitly, using CollationKey for each string to be compared, and maybe we can benefit from our SPI inside FWD-H2.

Yes, I think we need that. I do not have an exact example (when I was working with these I could give an exact case) but the idea is that in different collations characters may sort different. For example, in a collation the we could have â > ă, in other collation they might be sorted so that ă < â. In FWD the characters always use the Java String as internal representation - UTF, and these bytes may be unrelated to the desired collation. You can see a glimpse of the order for various collations in locale/*. The order of the supported characters in that specific CP is given by the order of unicode in the order_start section. The very same order you can see in Collator classes from com.goldencode.p2j.spi.*. The lib/spi/fwdspi.jar contains the latter. When H2 attempts to sort two strings, it cannot do that directly, only looking at the String representation UNLESS the collation is UNICODE (aka characters sorted by their unicode encoding). For all other collations, the UTF representation must be converted into something (numbers) with sort as expected in respective collation. Here come into play the CollationKey s: each String is replaced by a CollationKey (that is an array of integers) and the compare operation is done on those numbers instead. And these CollationKey s are constructed using the classes which extend BaseFwdCollator found in fwdspi.jar. This jar was especially constructed for H2 as the scripts from locale/ were created for PostgreSQL, for the very same purpose.

- Right now, H2 uses two traversals of each string before comparing it with ignoreCase. First traversal makes the string upper-case and the second compares using the Collator. There is no "compareIgnoreCase" for Collator. Do we have a workaround for this, integrating ignoreCase into the collator comparison?

Looking at the code, I see they do not compute the CollationKey s every time, instead the associated values are kept in a map/cache, to avoid converting the strings each time. Maybe we can use two of these caches (one CS one CI) and, when they are being created to automatically convert the characters to uppercase (or lowercase) and trim the spaces at the end. So the answer to your question is: we need to reduce the double traversals to a single one.

- Doing rtrim means removing the right spaces before comparison. However, manually right trimming and comparing means two string traversals. This means that the text indexed columns in H2 will be right trimmed at each comparison and this is clearly worse than keeping a rtrimed version of each indexed char column (computed column).

As noted above, my idea is to let the strings as they are and only create the associated CollationKey auto-trimmed and case-sensitive aware. If that is possible.

- Can we ever consider a char-by-char comparison of legacy character data? Or are we always forced to make use of the Collator, so we can't integrate ignoreCase and rtrim in a single string traversal?

We must use the Collator, regardless whether we compare one character at a time or the whole string at once. Again, individual character in a string are not compared by their U code, but using the order given by the collation.

> If we are tightly bound on using Collator for the H2 text comparisons, then I can't see how a H2 native rtrim may be faster than a computed column using a rtrim defined function.

If the CollationKey is constructed as the final spaces were removed, I think we do not need to inject the rtrim s at all, for both simple compare operations and indices.

#14 - 02/21/2023 05:04 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> Yes, I think we need that. I do not have an exact example (when I was working with these I could give an exact case) but the idea is that in different collations characters may sort different. For example, in a collation the we could have â > ă, in other collation they might be sorted so that ă < â. In FWD the characters always use the Java String as internal representation - UTF, and these bytes may be unrelated to the desired collation. You can see a glimpse of the order for various collations in locale/*. The order of the supported characters in that specific CP is given by the order of unicode in the order_start section. The very same order you can see in Collator classes from com.goldencode.p2j.spi.*. The lib/spi/fwdspi.jar contains the latter. When H2 attempts to sort two strings, it cannot do that directly, only looking at the String representation UNLESS the collation is UNICODE (aka characters sorted by their unicode encoding). For all other collations, the UTF representation must be converted into something (numbers) with sort as expected in respective collation. Here come into play the CollationKey s: each String is replaced by a CollationKey (that is an array of integers) and the compare operation is done on those numbers instead. And these CollationKey s are constructed using the classes which extend BaseFwdCollator found in fwdspi.jar. This jar was especially constructed for H2 as the scripts from locale/ were created for PostgreSQL, for the very same purpose.

This is very insightful. So the only difference between collations is that they provide a different character order. I am concerned about the high amount of CollationKey instances generated and the complexity of "normalizing" the strings. A simple Java String.compareTo does one simple traversal comparing each character code and can be altered in any way:

- we need caseInsensitive - we just compare the characters in upper case
- we need rtrim - we just stop at the first space and do some extra checks at the end.
- we need other String operations - we just inline them in the string comparing procedure

Because we rely on the collation key, we can't do any of the above "embedded customizations". **However**, for caseInsensitive, I've seen that there is a strength option which can help and it is locale dependent:

- "A" != "a" if strength Collator.TERTIARY
- "A" == "a" if strength Collator.SECONDARY or Collator.PRIMARY

The default in H2 is PRIMARY, so Collator.compare is case sensitive. We can generate a second Collator with TERNIARY to simulate that compareIgnoreCase I was thinking about. This way we avoid the upper-case of the string in that extra traversal, although we still do it to right trim :(I can't manage to make the most of this strength feature, so I think we should drop it, unless we can see its value somewhere else.

> Looking at the code, I see they do not compute the CollationKey s every time, instead the associated values are kept in a map/cache, to avoid converting the strings each time. Maybe we can use two of these caches (one CS one CI) and, when they are being created to automatically convert the characters to uppercase (or lowercase) and trim the spaces at the end. So the answer to your question is: we need to reduce the double traversals to a single one.

The existence of this cache is a life-saver. I also think that the cache can avoid double-traversals, if we customize the keys to our needs and use 2 caches: CS and CI with all values right trimmed. The only issue here is that we may end up with very similar keys pointing to the same value: aa, aA, Aa, AA, aa , Aa , etc. Although, I don't think this is a common case in large customer applications.

> As noted above, my idea is to let the strings as they are and only create the associated CollationKey auto-trimmed and case-sensitive aware. If that is possible.

This is possible. I can't think of a scenario in which we need to compare the varchar values without right trimming.

> We must use the Collator, regardless whether we compare one character at a time or the whole string at once. Again, individual character in a string are not compared by their U code, but using the order given by the collation.

I agree that we still need to use the Collator, but maybe we could have done something like:

```
int l1 = a.length;
int l2 = b.length;
for (int i = 0; i < Math.min(l1, l2); i++)
{
   char c1 = a.charAt(i);
   char c2 = b.charAt(i);
   // do upper-case for ignoreCase / honor the rtrim
   int r = <comparison of characters using the collator>;
   // return the r if not 0
}
```

Now that I have seen RuleBasedCollator.compare, things may not be that simple :) I thought only about a flexible way of comparing the strings in which we could have integrated ignoreCase and rtrim.

> If the CollationKey is constructed as the final spaces were removed, I think we do not need to inject the rtrim s at all, for both simple compare operations and indices.

The goal is to avoid injecting rtrim at all, so we should work around generating and caching the CollationKey faster and better. The cache has a capacity of 32_000, but I am still thinking of testing its hit/miss rate.

#15 - 02/21/2023 05:17 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> I looked again at your results in #7108-8. They do not seem alright. There are several cases like:
>
> - compare database field 'a' with variable 'a': 4GL **no**, FWD yes
>
> Regardless of the case-sensitiveness of the field and variable, the result of this should be positive in 4GL! They are the same value: a single lower-case a character.

Unless there is a testing issue, I have a theory! The case-insensitive database fields are also storing an upper-case version (much like we do with a computed column). All checks are done with that upper-case version from the database. Because the parameter is case sensitive, the comparison is sensitive so it ends up comparing A (computed column) with a (query parameter) and the answer is no in 4GL. This explains the following result: "compare database field CI 'a' with variable CS 'A': 4GL **yes**", because there is a case sensitive comparison between the computed column "A" and the query parameter "A".

This looks like a quirk, as 4GL is always using the upper-case version of the database field if CI. If the theory is right and we would need to address it, I think we can disregard the __s prefixed columns. Also, I think we can also simplify some of our H2 or FFC work to handle this quirk.

Danut, please extract this into a separate 4GL test: CI database field with CS query parameter. Re-check all the 4 combinations in a single procedure. Do the same test for persistent database. If the tests are right, please check the documentation for such information; we should really know if this is a quirk of a documented feature, so we can move on with the fix.

#16 - 02/21/2023 02:36 PM - Ovidiu Maxiniuc

Alexandru Lungu wrote:

> I agree that we still need to use the Collator, but maybe we could have done something like:
> [...]

Yes, this could be an improvement if the strings are really different at their beginning. This code will use the quick-out approach so the CollationKey is computed partially, only for the compared characters (up to first difference). OTOH, if the string values are similar at the beginning and accessed multiple time, the H2 implementation is better since thy compute the CollationKey only once (well, until the pair is dropped from cache, if that happens). Your code has the disadvantage that it is computed only partially so it cannot be cached, so in the long run, for a set of data with good entropy, I think the current H2 solution has better performance.

> Now that I have seen RuleBasedCollator.compare, things may not be that simple :) I thought only about a flexible way of comparing the strings in which we could have integrated ignoreCase and rtrim.

Life is never simple :-).

> > If the CollationKey is constructed as the final spaces were removed, I think we do not need to inject the rtrim s at all, for both simple compare operations and indices.
>
> The goal is to avoid injecting rtrim at all, so we should work around generating and caching the CollationKey faster and better.

That would be nice. Dropping the rtrim and upper methods means the precomputed columns __ci and __cs will not be necessary so we will also use less space, and less CPU to compute/update them.

> The cache has a capacity of 32_000, but I am still thinking of testing its hit/miss rate.

I think this can be configured externally, but I think it worths an investigation to see when the cache 'overflows'. But this is project-related, depends on the number of String instances are stored in (temporary-) DB.

#17 - 02/22/2023 03:33 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> Yes, this could be an improvement if the strings are really different at their beginning. This code will use the quick-out approach so the CollationKey is computed partially, only for the compared characters (up to first difference). OTOH, if the string values are similar at the beginning and accessed multiple time, the H2 implementation is better since thy compute the CollationKey only once (well, until the pair is dropped from cache, if that happens). Your code has the disadvantage that it is computed only partially so it cannot be cached, so in the long run, for a set of data with good entropy, I think the current H2 solution has better performance.

You are right. CollationKey.compareTo still does a full string traversal (like I do), but maybe that "<comparison of characters using the collator>" overhead may hurt the performance. Anyways, RuleBasedCollator.compareTo is too complex to bother rewriting in the hope to optimize some milliseconds (at best).

> I think this can be configured externally, but I think it worths an investigation to see when the cache 'overflows'. But this is project-related, depends on the number of String instances are stored in (temporary-) DB.

I agree this is project related. I was curios of the magnitude order of the number of strings used in a large project. For a large customer application I've got:

- number of calls to org.h2.value.CompareModeDefault.compareString: 2,485,979
- number of calls to org.h2.value.CompareModeDefault.getKey: 4,971,958
- number of cache hits: 4,911,679 (**98.7%**)
- number of cache misses: 60,279 (**1.3%**)

- java.text.CollationKey.compareTo: 2,485,979 (times), 122ms (total time), 0.0004ms (avg)
- java.text.Collator.getCollationKey: **60,279 (times), 530ms (total time), 8.7ms (avg)**

#18 - 02/22/2023 07:00 AM - Dănuț Filimon

- File results.txt added

- File temp-table-test.p added

- File persistent-table-test.p added

Alexandru Lungu wrote:

> Danut, please extract this into a separate 4GL test: CI database field with CS query parameter. Re-check all the 4 combinations in a single

> procedure. Do the same test for persistent database. If the tests are right, please check the documentation for such information; we should really know if this is a quirk of a documented feature, so we can move on with the fix.

I made two separate tests for temporary and persistent tables. The results were the same in both cases and can be checked in the attached **results.txt** file. I also found a related article that presents the same situation we are in: https://community.progress.com/s/article/P166063. As stated in the article, this is an expected behavior and it also refers to a note from the ABL Reference documentation which states that we should **not compare case-sensitive data with case-insensitive data in a WHERE expression**.

#19 - 02/22/2023 04:34 PM - Ovidiu Maxiniuc

I am really puzzled about this fact. I found the article in a parallel investigation. It really shocked my understanding of how comparing CI and CS instances work when one of them is an indexed field.

I understand that both permanent and temp-tables work the same so we can focus on only one of them, probably the temp-tables because they are easier to alter.

I altered your tests and remove the index for both tables: in this case we have the more logical result ('A'='A' and 'a'='a' but 'a'!='A' and 'A'!='a'). I did not test this is FWD (please, you do it), but FWD seems also wrong here. ATM, the FWD conversion injects upper(<field>) and toUpperCase(<subst>) (or just make the string in uppercase if it is a string literal) when and only when the compared field in current query component is case-insensitive, REGARDLESS the field being a component in an index. So we need to update the conversion algorithm to include these new findings.

However, in the article, they recommend **NOT** to *"compare case-sensitive data with case-insensitive data in a WHERE expression"*. The *"AVM cannot determine the results"* because **"case sensitivity in selection criteria is handled differently by different DataServers"**. In other words, the resultas are not deterministic and probably differ from installation to installation. I remember we had access to a Linux installation of 4GL and, indeed, there were various differences from Windows when I executed certain procedures (not necessarily related to LF / CRLF and / / \).

#20 - 02/23/2023 04:53 AM - Dănuț Filimon

Ovidiu Maxiniuc wrote:

> I altered your tests and remove the index for both tables: in this case we have the more logical result ('A'='A' and 'a'='a' but 'a'!='A' and 'A'!='a'). I did not test this is FWD (please, you do it), but FWD seems also wrong here.

I tested FWD and the results are: 'A'='A', 'a'='a', 'a'='A', 'A'='a'.

I am currently experimenting with VARCHAR_IGNORECASE and VARCHAR_CASESENSITIVE as an alternative to VARCHAR, eliminating usage of __s and __i. One of the problems I have right now is that the rows are missing parameters when comparing in H2 and I am looking into it.

#21 - 02/28/2023 07:23 AM - Dănuț Filimon

> One of the problems I have right now is that the rows are missing parameters when comparing in H2 and I am looking into it.

The parameters weren't missing, since __s and __i columns were eliminated, the numbers of parameters was less when comparing rows.

I made the necessary changes to include upper and rtrim into H2, at the same time I used VARCHAR_IGNORECASE and VARCHAR_CASESENSITIVE instead of VARCHAR, eliminated the need for computed columns. After comparing the results from the old version (no changes to FWD or H2) and the current one I managed to keep obtain the same results when executing my test cases, with no difference between them.

But there is another problem. Creating two temporary tables with the same name and structure, in table A the character field is case insensitive and in table B the character field is case sensitive, generates the same DMO and the character field has the property caseSensitive = true. This isn't right, it means that temp-tables that share the same DMO can force the character field to have this property and it can result in different outputs. In H2, the comparison is either case insensitive or case sensitive, and since a database field is forced to be case sensitive means that a comparison that should be done in a CI manner is actually done using CS values.

Example:
set1.p
DEFINE TEMP-TABLE tt1 FIELD key AS CHARACTER INDEX idx1 AS UNIQUE PRIMARY KEY.

set2.p
DEFINE TEMP-TABLE tt1 FIELD key AS CHARACTER CASE-SENSITIVE INDEX idx1 AS UNIQUE PRIMARY KEY.

Key getter property after conversion:
@Property(id = 1, name = "key", column = "key", legacy = "key", format = "x(8)", order = 0, caseSensitive = true)

Defining a third temporary-table removes the caseSensitive property:
set3.p
DEFINE TEMP-TABLE tt1 FIELD key AS CHARACTER INDEX idx1 AS UNIQUE PRIMARY KEY.

@Property(id = 1, name = "key", column = "key", legacy = "key", format = "x(8)", order = 0)

It sets the properties based on the last table converted.

As a workaround for the current issue, I avoid using the same temp-table name when testing.

#22 - 03/03/2023 09:18 AM - Alexandru Lungu

Regarding CompareModeDefault.compareString, I found some neat means of short-circuiting. Note that neither RuleBasedCollationKey.compareTo nor String.compareTo have such optimization. Therefore, it is safe to assume that following optimization is not happening at a later stage anyway. My results are independent, so applying one short-circuit may lower the hit rate of another.

- Due to the fact that we are caching the collation keys, there are some chances that we may end up with the same cache value. In fact, doing something like if (aKey = = bKey) return 0; is safe enough, as having equal keys means that the characters are the same and compareTo is going to return 0 anyway. In my tests, 10% - 17% of the compareString executions would hit such short-circuit.
- Some of the String arguments are "interned", so adding an if (a = = b) return 0; right at the beginning would save us some time: no cache look-up, no eventual collation key generation and no key comparison. This has a lower optimization hit, but provides a better improvement. In my tests, 1% - 6% of the compareString executions have equal references. Note that this is tested before any case-sensitivity is honored; better results may be achieved after honoring ignoreCase.
- H2 uses internal structures for its data types, including ValueString. These are cached, so adding if (this = = o) return 0; at the beginning of compareTypeSafe avoids any comparison attempt. This has a really low hit rate, but has a good improvement rate. However, this optimization is dominated by the previous one (as both operands will be the same String - the one from the single ValueString instance). This may be moved somewhere earlier (Comparison.compare or Value.compareWithNull. In my tests, 0.03% - 2.5% comparisons are between the same instance of ValueString.

We should integrate the first two for now to benefit from the "free meal". Danut, please consider these two short-circuits in 7108a_h2.

#23 - 03/06/2023 06:52 AM - Dănuț Filimon

Alexandru Lungu wrote:

> We should integrate the first two for now to benefit from the "free meal". Danut, please consider these two short-circuits in 7108a_h2.

The two short-circuits mentioned are very useful and there will be no problem if they are included. I retested my own testcases and got the expected results.

Committed 7108a_h2/rev.9. Added the changes did for including rtrim/upper, the mainly focus is CompareModeDefault that uses two caches for CI/CS values.
Committed 7108a/rev.14485. Changes that follow up 7108a_h2/rev.9 in which computed columns are no longer used by the P2JH2Dialect (creating tables, indexes, ddls).
Committed testcases/rev.2374. Added tests that use case-sensitive/case-insensitive variables/database fields.

I have a question related to FQLPreprocessor.simplifyCiProperties which I modified to remove both **upper** and **rtrim** function nodes. MariaDbLenientDialect uses also has isAutoRtrimAndCi() set to **true**. Is there any case where a **rtrim** is used but should not be removed when using this specific dialect?

#24 - 03/08/2023 07:43 AM - Alexandru Lungu

- % Done changed from 0 to 100

- Status changed from WIP to Review

First thing to notice: the application always requires reconversion due to the new DDL for the H2 meta database (even if H2 is not used as persistent database).

Rebased 7108a from trunk (rev. 14497). This is a review for 7108a:

- Note that you apply s.replace(CASE_TYPE_TOK, ""); to any type. You can do a check before hand (similar s.indexOf(SCALE_TOK); > 0) to avoid replacing CASE_TYPE_TOK if it doesn't exist.
- Please avoid adding refs: to history entries. Addign refs: to the commit message is enough, as the history entries can be correlated with the commit message.
- For src/com/goldencode/p2j/persist/orm/DDLGeneratorWorker.java, can't you just use field.isCaseSensitive() straigh-away, instead of guarding it with isChar? Only character has CASE_TYPE_TOK, so it doesn't really matter if it is null or a an arbitrary true/false value. This will save us from an extra if check.
- There are several needTrim usages in FQLPreprocessor. I see that you are supressing rtrim for FQL aliases. However, you can do the same for other factors like properties and function calls, right? Please iterate the FQLPreprocessor walks again to ensure that you handled all rtrim/upper predicates.
- It is best to remove P2JH2Dialect.getComputedColumnPrefix. It should fallback to Dialect.getComputedColumnPrefix, which is null. The same goes for P2JDialect.isComputedColumn, P2JDialect.isCaseInsensitiveColumn and P2JDialect.getComputedColumnFormula. Please recheck all the usages of these methods and ensure that by removing them will not break the dependent code. We don't want any kind of computed column code in P2JH2Dialect as long as it doesn't need it.

Rebased 7108a_h2 from H2 trunk (rev. 12). This is a review for 7108a_h2:

- testFWD shows regressions. I can't tell if they are false positives, but you should look into it. I get java.lang.NullPointerException for org.h2.test.synth.TestCrashAPI.test in memory. Persistent and server tests are OK.
- Inside CompareModeDefault, you can use comp = collator.compare(a, b) if there is no cache. Even if this is not the common FWD use-case, please honor the ignore-case and rtrim for it. Therefore, you should compare a and b as right trimmed and upper-cased if needed using collator.compare(a, b).
- Avoid using wildcard import in H2 (import org.h2.util.*;)

#25 - 03/08/2023 08:16 AM - Alexandru Lungu

Alexandru Lungu wrote:

> - testFWD shows regressions. I can't tell if they are false positives, but you should look into it. I get java.lang.NullPointerException for org.h2.test.synth.TestCrashAPI.test in memory. Persistent and server tests are OK.

This is a false positive. I ran the regression tests before rebasing. I've retried running the regression tests after rebase and everything is alright now.

#26 - 03/09/2023 07:39 AM - Dănuț Filimon

Alexandru Lungu wrote:

- Note that you apply s.replace(CASE_TYPE_TOK, ""); to any type. You can do a check before hand (similar s.indexOf(SCALE_TOK); > 0) to avoid replacing CASE_TYPE_TOK if it doesn't exist.
- For src/com/goldencode/p2j/persist/orm/DDLGeneratorWorker.java, can't you just use field.isCaseSensitive() straigh-away, instead of guarding it with isChar? Only character has CASE_TYPE_TOK, so it doesn't really matter if it is null or a an arbitrary true/false value. This will save us from an extra if check.

I included these changes.

- There are several needTrim usages in FQLPreprocessor. I see that you are supressing rtrim for FQL aliases. However, you can do the same for other factors like properties and function calls, right? Please iterate the FQLPreprocessor walks again to ensure that you handled all rtrim/upper predicates.

upper and rtrim nodes are removed in FQLPreprocessor.simplifyCiProperties before FQLPreprocessor.mainWalk is called. After looking at the cases where needTrim is used, I made changes to allow trimming based on dialect.isAutoRtrimAndCi.

- It is best to remove P2JH2Dialect.getComputedColumnPrefix. It should fallback to Dialect.getComputedColumnPrefix, which is null. The same goes for P2JDialect.isComputedColumn, P2JDialect.isCaseInsensitiveColumn and P2JDialect.getComputedColumnFormula. Please recheck all the usages of these methods and ensure that by removing them will not break the dependent code. We don't want any kind of computed column code in P2JH2Dialect as long as it doesn't need it.

Removed P2JH2Dialect.getComputedColumnPrefix, P2JH2Dialect.isComputedColumn and P2JH2Dialect.getComputedColumnFormula. The P2JH2Dialect.isCaseInsensitiveColumn checks if a column is case insensitive based on it's name. Looking into the method, it is called by either deprecated methods, or methods that are never called (maybe it can be called in a converted application) so I made it return true. If there are any problems with this change, please let me know.

- Inside CompareModeDefault, you can use comp = collator.compare(a, b) if there is no cache. Even if this is not the common FWD use-case, please honor the ignore-case and rtrim for it. Therefore, you should compare a and b as right trimmed and upper-cased if needed using collator.compare(a, b).
- Avoid using wildcard import in H2 (import org.h2.util.*;)

Done so. Thanks for pointing out that the cache can be disabled and that I committed unnecessary imports.

I reconverted a customer application to test P2JH2Dialect as the main dialect for the database. Specific directory.xml settings were required to make the application start, but I managed to browse the application with no new issues, both before and after making the changes above.

Committed 7108a_h2/rev.14. Handle ignore-case and rtrim when collator cache is disabled (cache size is **0**).
Committed 7108a/rev.14499. Removed unnecessary methods from P2JH2Dialect related to computed columns and other small changes related to handling the CASE_TYPE_TOK and getSqlMappedType caseSensitive parameter.

#27 - 03/15/2023 06:38 AM - Dănuț Filimon

I noticed a problem related to unique index/primary keys where the value that needs to be inserted is rtrimmed, but already exists in the table. Currently looking into the issue.

#28 - 03/22/2023 06:34 AM - Dănuț Filimon

When I added rtrimming to H2, I set it so that it will always rtrim when comparing strings. Then I noticed an issue during conversion where it would try to insert a value which already exists in the table with a primary unique field. The difference between the two values was that one had an extra space at the end and when comparing, it would rtrim and return that the values were the same.

Committed 7108a_h2/rev.15. Added a RTRIM database property which is used by CompareMode(s) to choose between rtrimming strings when comparing or not.
Committed 7108a/rev.14500. Added RTRIM property to in-memory temp-tables.

The changes above fix the conversion issue.

#29 - 03/22/2023 07:19 AM - Dănuț Filimon

- Related to Bug #7220: Record not created when property is set to empty string added

#30 - 03/28/2023 08:25 AM - Alexandru Lungu

Rebased 7108a from trunk rev. 14515. 7108a is now at rev. 14518.

#31 - 03/28/2023 10:58 AM - Alexandru Lungu

Reviews

Review for 7108a

- SortCriterion instances are still generated with upper and rtrim! Please check constructor, ignoreCase and computedColumnPrefix attributes. This is a show stopper.
- getComputedColumnPrefix is returning null from Dialect. Please make it return the empty string. I see places where this prefix is not null checked (i.e. TemporaryBuffer.getComputedColumnPrefix). I didn't checked if null has a functional relevance here. Please check if moving to empty string is safe.

Note that after executing a simple query over a temp-table, I get an SQL like:

```
select [...] from tt3 tt_1_1__im0_ where tt_1_1__im0_._multiplex = ? and tt_1_1__im0_.f1 = 'X' order by tt_1_1
__im0_._multiplex asc, upper(rtrim(tt_1_1__im0_.f1)) asc nulls last, tt_1_1__im0_.recid asc lazy
```

It is nice to have tt_1_1__im0_.f1 = 'X' without any upper or rtrim. However, I still see upper(rtrim(tt_1_1__im0_.f1)) in the order by clause. This is not optimal, because the planner may have chosen to use an index to satisfy the ORDER BY. However, the index is on f1 alone, so the custom functions won't be implicitly solved by using the index. I am concerned that right now indexes won't aid the ORDER BY. Anyway, you should just remove the upper and rtrim, because f1 is CASE_INSENSITIVE and RTRIM connection setting is set.

Review for 7108a_h2

- I am OK with the changes

I suggest building a Test file explicitly for your rtrim setting. Please ask Radu, as he has already done some custom tests in H2 when working on #7061. It should basically reflect your test suite from FWD: test all combinations of upper/lower/rtrim etc. and check the answers are right with/without RTRIM setting. You can test both CompareModeDefault and CompareMode. Don't use any collation; the implicit collation is fine for testing. Allow this test suite of yours to be run for all testing configurations (memory, persistent and server). Run ./build.sh testFWD to make sure your tests are OK.

The test-suite is a low priority, for now. Make sure you address other high priority tasks first.

Test

I've done some slim regression testing for your changes over _temp, _dirty and _meta. All seems right.
I am looking forward to do some extensive testing over _temp, _dirty and _meta with other customer applications as well (the ones with RTRIM on). Also, I am planning to try using your changes in a customer application with H2 database. AFAIK, you already converted a large customer application and you had no issues while converting. No need to test this again.

Greg, once we will have 7108a integrated, the standard H2 connection string should change. Currently, we connect to persistent H2 with the URL from directory.xml (From HotelGUI: jdbc:h2:../db/[dbname];DB_CLOSE_DELAY=-1;MV_STORE=FALSE). This should be changed to jdbc:h2:../db/[dbname];DB_CLOSE_DELAY=-1;MV_STORE=FALSE;RTRIM=TRUE in order to allow H2 to do the right trimming for us, when comparing text values. This is a follow-up of removing upper and rtrim generation when using H2.

Performance

Overall, I get a slightly better performance (-0.2%), even if the SortCriterion is not optimal. This is not a regression, as the order should be upper and rtrim anyways. The matter is that the order of VARCHAR_IGNORECASE with RTRIM implies upper and rtrim. I expect to see a better performance after your changes. I will redo the performance tests then.

#32 - 03/29/2023 10:16 AM - Alexandru Lungu

Danut, please mark #7108-31 as high priority for now.

#33 - 03/30/2023 05:39 AM - Dănuț Filimon

- Priority changed from Normal to High

Committed 7108a/rev.14519. Skip adding upper and rtrim in the SortCriterion.toString represensation when necessary. It is safe to return an empty string in Dialect.getComputedColumnPrefix.

Ovidiu, is there any case where a rtrim is used but should not be removed when using MariaDbLenientDialect? A while back I also modified FQLPreprocessor.simplifyCiProperties to remove both upper and rtrim function nodes.

#34 - 03/30/2023 05:39 AM - Dănuț Filimon

- Priority changed from High to Normal

#35 - 03/30/2023 10:08 AM - Dănuț Filimon

Committed 7108a_h2/rev.18. Fix for cached CompareMode where the same instance is used but a different one is requested. This is related to shouldRtrim which is not checked when looking for a cached CompareMode.

#36 - 03/31/2023 08:58 PM - Ovidiu Maxiniuc

Dănuț Filimon wrote:

> Ovidiu, is there any case where a rtrim is used but should not be removed when using MariaDbLenientDialect? A while back I also modified FQLPreprocessor.simplifyCiProperties to remove both upper and rtrim function nodes.

I am not sure I understand your question. ATM, the only difference known to me between the MariaDbDialect and MariaDbLenientDialect is the way they handle sorting of null in indexed columns, and not related to case sensitivity or right-trimming. Indeed, both dialects (actually the character type

on SQL side) handle both of these last character processing natively, so upper and rtrim are actually useless here, at least when the fields are compared directly (like upper(field1)=rtrim(field2)). Things change when the 4GL programmer use the trim/right-trim/left-trim in a sub-expression and we are not interested in dropping it (consider that there is a wrapping length which is intended to count the extra spaces, too).

Note that, for the moment, only MariaDb support this kind of character data type, but in future the other dialects will benefit from it, more or less (citext in PostgreSQL and the result of this task for H2).

I hope I answer your question.

#37 - 04/03/2023 07:42 AM - Dănuț Filimon

Ovidiu Maxiniuc wrote:

> ... Indeed, both dialects (actually the character type on SQL side) handle both of these last character processing natively, so upper and rtrim are actually useless here, at least when the fields are compared directly (like upper(field1)=rtrim(field2)). ...

Yes. Thanks for answering.

Committed 7108a_h2/rev.19. Added a test for the rtrim database property.
I want to make a tutorial on how to create a test/add database options/add new keywords in H2_Database_Fork so that it would be easier to implement such changes in the future.

#38 - 04/04/2023 04:59 AM - Alexandru Lungu

Rebased 7108a with trunk (rev. 14523). 7108a is now at rev. 14527.

#39 - 04/04/2023 06:32 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> Dănuț Filimon wrote:
> Note that, for the moment, only MariaDb support this kind of character data type, but in future the other dialects will benefit from it, more or less (citext in PostgreSQL and the result of this task for H2).

I see that FQLPreprocessor.isAutoRtrimAndCi changes imply two effects. If the dialect returns true for such option:

- remove **all** upper and rtrim function calls from where clauses. This means that the database is able to do the case-insensitive/sensitive comparison on its own. upper(rtrim(tt.f1)) = ? is obsolete. tt.f1 = ? is enough to filter the tt records using case-insensitive comparison over f1, ignoring right spaces.
- remove **all** upper and rtrim function calls from sort criteria. This means that the database has ordered indexes that honor case sensitivity and right trimming on its own. order by upper(rtrim(tt.f1)) is obsolete. order by tt.f1 is enough to sort tt records by case-insensitive f1, ignoring right spaces.

In my examples, tt.f1 is defined as a case-insensitive character field. For FWD-H2, the changes in 7108a_h2 are enough to guarantee the above. For MariaDbLenientDialect, I can't confirm. For PostgreSQL and MariaDb dialects, the flag is false anyways. I think Danut wanted to stress out that rtrim is also removed from where and order-by for MariaDbLenientDialect. This wasn't happening before the 7108a changes.

#40 - 04/05/2023 04:17 AM - Alexandru Lungu

I tested several applications:

- customer application POC - no regressions
- 2 customer application regression testing - no regressions
- Hotel GUI (with and without H2 database) - no regressions

Profiled and got a -0.6% improvement. Updated H2 Database Fork.
Review for both 7108a and 7108a_h2. I am OK with the changes.

My last concern now is that the H2 regression testing fails. This is the only test failing on all environments: memory, persistent and remote.

```
11:11:41 00:00.044 org.h2.test.db.TestCompatibility Expected: T<*>RUE (4) actual: FALSE (5)
11:11:41 00:00.044 org.h2.test.db.TestCompatibility FAIL java.lang.AssertionError: Expected: T<*>RUE (4) actua
l: FALSE (5)
ERROR: FAIL (pageStore memory ) java.lang.AssertionError: Expected: T<*>RUE (4) actual: FALSE (5)  java.lang.A
ssertionError: Expected: T<*>RUE (4) actual: FALSE (5)  ------------------------------
java.lang.AssertionError: Expected: T<*>RUE (4) actual: FALSE (5)
    at org.h2.test.TestBase.fail(TestBase.java:327)
    at org.h2.test.TestBase.assertEquals(TestBase.java:653)
    at org.h2.test.TestBase.assertEquals(TestBase.java:666)
    at org.h2.test.TestBase.assertResult(TestBase.java:938)
    at org.h2.test.db.TestCompatibility.testDB2(TestCompatibility.java:650)
    at org.h2.test.db.TestCompatibility.test(TestCompatibility.java:53)
    at org.h2.test.TestBase.runTest(TestBase.java:140)
    at org.h2.test.TestAll.addTest(TestAll.java:1042)
    at org.h2.test.TestAll.test(TestAll.java:762)
    at org.h2.test.TestAll.run(TestAll.java:560)
    at org.h2.test.TestAll.run(TestAll.java:574)
    at org.h2.test.TestAll.main(TestAll.java:458)
```

Danut, please address this issue and I will move this to test.

#41 - 04/05/2023 05:00 AM - Alexandru Lungu

- Status changed from Review to Test

Alexandru Lungu wrote:

> Danut, please address this issue and I will move this to test.

Never mind. This is not caused by your changes, it also regresses in FWD-H2 rev. 7 (at which point they worked). The testcase is SELECT CURRENT_TIME = CURRENT TIME is DB2 compatibility mode. The difference between now and when we fixed regression testing (February) is that now, here in Iasi, Romania, we switched to the daylight saving time (EEST+3 instead of EET+2). I think CURRENT_TIME and CURRENT TIME in DB2 is not aware of daylight saving time and fails. I suggest removing this test as FWD is not using H2 in DB2 compatibility mode anyway.

From my POV, 7108a is ready for testing and integration in trunk.
The only matter to solve is Ovidiu's review on the isAutoRtrimAndCi usage changes (#7108-39) which affects MariaDbLenientDialect.

#42 - 04/05/2023 07:47 AM - Alexandru Lungu

Note that the changes require reconversion of the application (or at least regeneration of DDL - convert.middle and jar). It took me a while to understand while my tests were failing before #7108-40; it was due to incorrect _meta DDL (which is stored as meta_index_ddl.sql and meta_table_ddl.sql in the application jar (dmo). Also, it requires database reimport if the persistent database is in H2.

#43 - 04/05/2023 08:45 PM - Ovidiu Maxiniuc

Alexandru Lungu wrote:

> Ovidiu Maxiniuc wrote:
>
> > Dănuț Filimon wrote:
> > Note that, for the moment, only MariaDb support this kind of character data type, but in future the other dialects will benefit from it, more or less (citext in PostgreSQL and the result of this task for H2).
>
> I see that FQLPreprocessor.isAutoRtrimAndCi changes imply two effects. If the dialect returns true for such option:
>
> - remove **all** upper and rtrim function calls from where clauses. This means that the database is able to do the case-insensitive/sensitive comparison on its own. upper(rtrim(tt.f1)) = ? is obsolete. tt.f1 = ? is enough to filter the tt records using case-insensitive comparison over f1, ignoring right spaces.
> - remove **all** upper and rtrim function calls from sort criteria. This means that the database has ordered indexes that honor case sensitivity and right trimming on its own. order by upper(rtrim(tt.f1)) is obsolete. order by tt.f1 is enough to sort tt records by case-insensitive f1, ignoring right spaces.
>
> In my examples, tt.f1 is defined as a case-insensitive character field. For FWD-H2, the changes in 7108a_h2 are enough to guarantee the above. For MariaDbLenientDialect, I can't confirm. For PostgreSQL and MariaDb dialects, the flag is false anyways. I think Danut wanted to stress out that rtrim is also removed from where and order-by for MariaDbLenientDialect. This wasn't happening before the 7108a changes.

Historically speaking (in P2J/FWD terms), the rtrim and upper were handled together because the 4GL handling of default character type and the lack of PostgreSQL/Java/H2 to handle this natively. However, things start to change. With the addition of MariaDb (which handle natively both), support in H2 and the new character types in latest PSQL, the FWD code must be more dialect aware (and maybe configuration aware since I understand some clients request specific datatype for their databases). So a first step would be to split the isAutoRtrimAndCi() and create dedicated methods for trimming and CI. The second would be to pass a reference to the field so that the dialect has more knowledge (if the specific field was converted to a specific data type).

#44 - 04/06/2023 03:23 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> Historically speaking (in P2J/FWD terms), the rtrim and upper were handled together because the 4GL handling of default character type and the lack of PostgreSQL/Java/H2 to handle this natively. However, things start to change. With the addition of MariaDb (which handle natively both), support in H2 and the new character types in latest PSQL, the FWD code must be more dialect aware (and maybe configuration aware since I understand some clients request specific datatype for their databases). So a first step would be to split the isAutoRtrimAndCi() and create dedicated methods for trimming and CI. The second would be to pass a reference to the field so that the dialect has more knowledge (if the specific field was converted to a specific data type).

Danut please address this. Splitting isAutoRtrimAndCi seems natural: isAutoRtrim and isAutoCi. The parameter should be a field, so the dialect is configuration aware. Only H2 and MariaDb are going to handle this; for now, let both dialects return true for both methods, no matter the field type. This is going to be extended at the right time.

#45 - 04/06/2023 07:34 AM - Dănuț Filimon

Committed 7108a/rev.14528. Removed isAutoRtrimAndCi and replaced it with isAutoRtrim and isAutoCi.
I think that using a parameter for the isAutoRtrim and isAutoCi may require a lot more.
The isAutoRtrimAndCi method is used in 3 places:

- DBUtils.composePropertyName: The dialect is given as a parameter, along with the name of the property (a String).
- FQLPreprocessor.preprocess: Here isAutoRtrimAndCi is used in a general way, based on dialect and not a single property.
- FQLPreprocessor.mainWalk: Similar to the previous one.

My point is that it would take time to find the required property for isAutoRtrim and isAutoCi based on a HQLAst node or a String value. Looking into how those properties can be obtained, we can use RecordBuffer.lookupBuffer, which returns a RecordBuffer based on a HQLAst node of type ALIAS. Then we can access the properties through the DMO.

Please advice.

#46 - 04/11/2023 04:40 AM - Alexandru Lungu

Review to 7108a/rev.14528

The changes are good. Last note to address before attempting to do a full test round and merge 7108a in trunk.

- In FQLPreprocessor, please keep the simplifyCiProperties guard. Otherwise, we risk a full AST traversal with both eliminateUpper and eliminateRtrim on false, which is a time loss. Execute simplifyCiProperties only if isAutoCi or isAutoRtrim.

#47 - 04/11/2023 08:10 AM - Dănuț Filimon

Alexandru Lungu wrote:

> In FQLPreprocessor, please keep the simplifyCiProperties guard. Otherwise, we risk a full AST traversal with both eliminateUpper and eliminateRtrim on false, which is a time loss. Execute simplifyCiProperties only if isAutoCi or isAutoRtrim.

Committed 7108a/rev.14529. Added back the guard for simplifyProperties.

#48 - 04/11/2023 09:55 AM - Alexandru Lungu

I am OK with the current implementation. This should be tested and merged into trunk.

#49 - 04/18/2023 06:51 AM - Alexandru Lungu

Constantin, can you run the ETF tests with 7108a?
I rebased 7108a with trunk rev. 14537.
I rebased 7108a_h2 with FWD-H2 trunk rev. 15.
I uploaded fwd-h2-1.20-7108a.jar build to devsrv01:/tmp/

Note that the changes require middle reconversion and jar (it regenerates the meta DDL which should be included in the new jar).
If you use the H2 persistent database, reimport is also required and the append of RTRIM=true to connection string is required.

The changes include string comparison optimizations in H2 and the elimination of computed columns for upper and rtrim. upper is handled by varchar_(casesensitive|ignorecase) and rtrim is solved by connection string RTRIM=true.
Temporary, meta and dirty databases have RTRIM=true in the connection string from FWD run-time.
H2 databases used at conversion time don't use RTRIM=true.

The final improvement from this changes alone is -0.6% (using PostgreSQL persistent database).

#50 - 04/18/2023 07:20 AM - Constantin Asofiei

Alexandru Lungu wrote:

> Constantin, can you run the ETF tests with 7108a?

OK, I'll do this. Is this for performance or just validation/regression testing?

> Note that the changes require middle reconversion and jar (it regenerates the meta DDL which should be included in the new jar).

I'll need to reconvert anyway.

> If you use the H2 persistent database, reimport is also required and the append of RTRIM=true to connection string is required.

I'm using postgresql for the physical database - do I need to do anything for it? I'm not re-importing, but restoring from a master database.

#51 - 04/18/2023 08:29 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Alexandru Lungu wrote:
>
> > Constantin, can you run the ETF tests with 7108a?
>
> OK, I'll do this. Is this for performance or just validation/regression testing?

Regression testing mostly. I already run two customer application regression testing and one profiling POC and everything is OK. I want to make sure it is safe enough for merge.

> > If you use the H2 persistent database, reimport is also required and the append of RTRIM=true to connection string is required.
>
> I'm using postgresql for the physical database - do I need to do anything for it? I'm not re-importing, but restoring from a master database.

No. PostgreSQL works as before.

#52 - 04/19/2023 12:49 PM - Constantin Asofiei

ETF testing is OK with 7108a rev 14543 and fwd-h2-1.20-7108a.jar

#53 - 04/20/2023 04:56 AM - Alexandru Lungu

Constantin Asofiei wrote:

> ETF testing is OK with 7108a rev 14543 and fwd-h2-1.20-7108a.jar

Good.

Greg, Eric: Please let me know if I can merge this in trunk.

#54 - 04/25/2023 05:47 AM - Alexandru Lungu

Branch 7108a has been merged to trunk at revision 14550

#55 - 04/25/2023 05:47 AM - Alexandru Lungu

Branch 7108a_h2 has been merged to FWD-H2 trunk at revision 16.

#56 - 05/15/2023 09:23 AM - Alexandru Lungu

Dănuț Filimon wrote:

> I found several mismatched between FWD and 4GL. All of the tests are done in the following manner:
> [...]
>
> The following test cases had different results from each other:
>
> - compare case-insensitive database field with case-sensitive variable
>   - case-sensitivity
>   - compare database field 'A' with variable 'a': 4GL no, FWD yes
>   - compare database field 'a' with variable 'a': 4GL no, FWD yes
>   - rtrim
>   - compare database field 'a ' with variable 'a': 4GL no, FWD yes
>   - compare database field 'a' with variable 'a ': 4GL no, FWD yes
>   - compare database field 'a ' with variable 'a ': 4GL no, FWD yes
>   - other, no ltrim
>   - compare database field ' a' with variable ' a': 4GL no FWD yes
> - compare case-sensitive database field with case-insensitive variable
>   - case-sensitivity (this happens only if the FastFindCache is enabled)
>   - compare database field 'A' with variable 'a': 4GL no FWD yes
>   - compare database field 'a' with variable 'a': 4GL yes, FWD no

Danut, please create a task on Database project for this.
Relate to this task.
Mention what are the prerequisites (case-insensitive / rtrim look-up in where clauses). Mention that there are some issues already (with/without FastFindCache). Reference the test-suite you uploaded to testcases.

#57 - 05/15/2023 09:47 AM - Dănuț Filimon

- Related to Bug #7352: Mismatching results when comparing case-insensitive/case-sensitive values between 4GL and FWD added

#58 - 07/21/2023 10:35 AM - Eric Faulhaber

Can this be closed?

#59 - 07/21/2023 10:36 AM - Alexandru Lungu

Yes

#60 - 07/21/2023 10:49 AM - Eric Faulhaber

- Status changed from Test to Closed

Files

ffCache-20230214.patch	1.49 KB	02/14/2023	Dănuț Filimon
persistent-table-test.p	1.1 KB	02/22/2023	Dănuț Filimon
temp-table-test.p	1.07 KB	02/22/2023	Dănuț Filimon
results.txt	628 Bytes	02/22/2023	Dănuț Filimon