Database - Bug #6829

H2 forces re-parse of all prepared statements when metadata is changed

10/07/2022 04:18 AM - Constantin Asofiei

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 - Feature #7060: Consider using wildcard selection to red...	**Test**

History

#1 - 10/07/2022 04:22 AM - Constantin Asofiei

In H2, there is this code in org.h2.command.Prepared:

```
    public boolean needRecompile() {
        Database db = session.getDatabase();
        if (db == null) {
            throw DbException.get(ErrorCode.CONNECTION_BROKEN_1, "database closed");
        }
        // parser: currently, compiling every create/drop/... twice
        // because needRecompile return true even for the first execution
        return prepareAlways ||
                modificationMetaId < db.getModificationMetaId() ||
                db.getSettings().recompileAlways;
    }
```

which is called by CommandContainer.recompileIfRequired():

```
    private void recompileIfRequired() {
        if (prepared.needRecompile()) {
          .....  parse ....
        }
    }
```

The Database.modificationMetaId gets changed each time a table gets dropped or created. This causes all the prepared statement to be parsed again.

Eric: maybe is worth it to change H2 to add a connection flag, to never re-compile the prepared statements if DB metadata has changed?

#2 - 10/07/2022 04:34 AM - Constantin Asofiei

As a side note: when executing some 60k H2 prepared statements, ~70% of the time is being spent in re-compiling the query.

#3 - 10/10/2022 12:34 PM - Constantin Asofiei

The FWD-level cache of prepared statements in TempTableDataSourceProvider's psCache will also keep statements related to dropped temp-tables; although they will be evicted from the LRUCache as other queries are prepared. But, there may be an issue: are dynamic temp-table names unique for the lifetime of a FWD server? Because a PreparedStatement will keep references to H2 structures related to index and/or table, which will be used by that instance, if accessed, even if the original temp-table is long gone.

#4 - 10/11/2022 12:29 PM - Constantin Asofiei

As a query can target multiple metadata (like tables, indexes, sequences), the cache needs to be notified to evict all PreparedStatement instances which are associated with that metadata. So, if a table is dropped/altered, all PreparedStatement instances using that table need to be evicted from the cache. This needs to be done at both FWD and H2 caches.

#5 - 10/13/2022 05:01 AM - Alexandru Lungu

> But, there may be an issue: are dynamic temp-table names unique for the lifetime of a FWD server? Because a PreparedStatement will keep references to H2 structures related to index and/or table, which will be used by that instance, if accessed, even if the original temp-table is long gone.

Dynamic temp-tables are named uniquely across all sessions using long tableId = tableCounter.getAndIncrement(); String sqlTableName = "dtt" + tableId;. Of course, temp-tables with similar structure, even dynamic, use multiplex. tableCounter is a cross-session atomic long variable.

> The FWD-level cache of prepared statements in TempTableDataSourceProvider's psCache will also keep statements related to dropped temp-tables; although they will be evicted from the LRUCache as other queries are prepared.

Can you help me understand how such state is achieved? I can't tell how FWD can drop physical H2 "local temporary" tables and still have prepared statements cached. After a short investigation, the drop statements are used only when closing a database session, which removes the temporary tables anyways. This also applies for dynamic temp-tables. As psCache is part of TempTableDataSourceProvider.Context, I expect it to belong to one single FWD user, which can't trigger a table drop without closing the session altogether (which ultimately clears the context, so the cache?). Therefore, is a psCache with prepared statements from a dropped table a valid FWD state?

#6 - 10/13/2022 06:13 AM - Constantin Asofiei

Alexandru Lungu wrote:

> The FWD-level cache of prepared statements in TempTableDataSourceProvider's psCache will also keep statements related to dropped temp-tables; although they will be evicted from the LRUCache as other queries are prepared.

> Can you help me understand how such state is achieved? I can't tell how FWD can drop physical H2 "local temporary" tables and still have prepared statements cached.

On FWD Session close, TemporaryBuffer.dropUnusedTables will be called - this will clean all temp-tables which are no longer in use, collected when TemporaryBuffer.doCloseMultiplexScope is executed (see pendingDrop usage). This has two effects:

- on a metadata change, H2 will clear its prepared-statement cache (it will do this on a CREATE for table or sequence, too)
- psCache will keep all prepared statements which were executed on this dropped table.

In FWD, the FWD persistence session for the temporary database will get opened and closed depending on the application usage.

#7 - 10/13/2022 07:06 AM - Alexandru Lungu

> In FWD, the FWD persistence session for the temporary database will get opened and closed depending on the application usage.

What do you mean by application usage? Presuming I am a Web client user which just got some work done (involving temp-tables), how can I trigger TemporaryBuffer.dropUnusedTables without disconnecting from my temporary database anyways? I think that simply closing the Web client will force the server database session to close and thus ALL my temporary tables will be dropped. My Context from the server will also be removed, so psCache will be deleted (or I may be wrong here).

It is clear to me the reasoning why the cache may become inconsistent, but I don't know how to reproduce this (at least theoretical).

#8 - 10/13/2022 07:45 AM - Constantin Asofiei

Alex, I was under the impression that a test like this, dyntable.p:

```
def var h as handle.
create temp-table h.
h:add-new-field("f1", "blob").
h:temp-table-prepare("tt").
delete object h.
```

when ran via a RUN statement from another program, like RUN dyntable.p, once dyntable.p ends, FWD will drop the temp-table, as noone else uses it. But the 'sessionUseCount' does not get incremented. This means that the persistence Session for the temp database will remain for the lifetime of the client - can never be closed.

Eric: can this be a bug in the sesionUseCount?

#9 - 10/13/2022 08:53 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Alex, I was under the impression that a test like this, dyntable.p:
> [...]
>
> when ran via a RUN statement from another program, like RUN dyntable.p, once dyntable.p ends, FWD will drop the temp-table, as noone else uses it. But the 'sessionUseCount' does not get incremented. This means that the persistence Session for the temp database will remain for the lifetime of the client - can never be closed.

This is **exactly** what I tested earlier today. After running this twice from another program, the same physical dynamic table is used with another multiplex.

#10 - 10/13/2022 03:38 PM - Constantin Asofiei

- Start date deleted (10/07/2022)

Alex, there's something else I'd like to understand: I've increased the FWD's psCache to 65535 for a large customer app, and for some heavy testing psCache ended up pinning ~250MB of memory, from PreparedStatements. Please check if H2 isn't keeping the last result or something like this at the PreparedStatement, otherwise if a PreparedStatement really can be ~4KB, tuning this psCache will need to consider this.

#11 - 10/14/2022 07:10 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Alex, there's something else I'd like to understand: I've increased the FWD's psCache to 65535 for a large customer app, and for some heavy testing psCache ended up pinning ~250MB of memory, from PreparedStatements. Please check if H2 isn't keeping the last result or something like this at the PreparedStatement, otherwise if a PreparedStatement really can be ~4KB, tuning this psCache will need to consider this.

An UnclosablePreparedStatement, which is used to proxy prepared statements for temp-tables, is either checked-out (it is actively used by a query) or not (it is in psCache). When doing the check-in (returning the query back in psCache), the result-set is closed. This means that the cached prepared statements are all having closed result sets. In particular, a cache evicted prepared statement will be instantly closed or will be closed when it will be checked-in.

It is curious why a psCache statement is ~4KB. This means that the "no result-set/generated-keys when in psCache" invariant doesn't hold. Looking into it.

Update: Checked-out statements can reside in psCache. They are only marked as checked-out, but not removed from the cache. The invariant is rather: "no result-set/generated-keys when checked-in".

#12 - 11/14/2022 06:23 AM - Alexandru Lungu

I've done some simple debugging in Hotel GUI with UnclosablePreparedStatement. The psCache structure is an ExpiryCache:

- The key is a PSKey which also contains the SQL. In Hotel GUI, most of the statements I am exploring are UPDATE with ~50 characters. However, in large customer applications I encountered simple SELECT with ~700 characters (~400 for columns retrieved, ~200 for WHERE, ~100 for ORDER BY). There are also simple INSERT or UPDATE with ~500 characters or SELECT with join or sub-select with ~1000 characters. Queries which large field list, joins, sub-selects, large ORDER BY and large WHERE may even reach ~2000 characters.
- The value is, of course, an UnclosablePreparedStatement. The memory consumption is similar to the PreparedStatement generated by H2. I didn't enountered issues with ResultsSet being opened for checked-in statements. However, note that statements which are in use are not removed from psCache. You may encounter statements with a opened ResultSet in psCache, but they are also marked as checkedOut.
- The H2 PreparedStatement stores the initial command (so the SQL) and the prepared statement.

Considering all these, I expect an average of ~2kB to be used per psCache entry as the SQL String is stored twice per entry + other extra data which add up. The psCache may also have some queries (1/2) which are checked-out, so the ResultSet may contribute to the memory analysis.

The following is an example of a very simple field list from one of my queries, with 361 characters. Consider having more columns and longer table/field names.

```
tt_2_1__im0_.recid as id0_, tt_2_1__im0_._multiplex as column1_0_, tt_2_1__im0_._errorFlag as column2_0_, tt_2
_1__im0_._originRowid as column3_0_, tt_2_1__im0_._datasourceRowid as column4_0_, tt_2_1__im0_._errorString as
 column5_0_, tt_2_1__im0_._peerRowid as column6_0_, tt_2_1__im0_._rowState as column7_0_, tt_2_1__im0_.f1 as f
8_0_, tt_2_1__im0_.f11 as f9_0_
```

#13 - 11/14/2022 09:07 AM - Constantin Asofiei

Alexandru, I have in a heap dump UnclosablePreparedStatement instances with ~90KB retained heap. But in at least a case, the SELECT is ~15k chars long. The main issue I see is for a column named thisisaverylongcolumnname, a thisisaverylongcolumnname170_0_ alias is created, too, which doubles the size (and there are cases with 250 or more fields, and this will explode even more if we move to denormalized extent). More, the SQL is kept at org.h2.jdbc.JdbcPreparedStatement.sqlStatement, CommandContainer.sql, org.h2.command.dml.Select.sqlStatement, so we have 3 places where heap is consumed for the SQL.

I'm still trying to figure out if there are 'dropped tables' for which we cache the PreparedStatement on FWD side, but in any case, H2 should allow notifications to evict a cached PreparedStatement from its internal cache or the FWD cache.

Also, the Select.lastResult also may consume memory, depending on what was returned on the last query.

#14 - 01/03/2023 07:08 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Alexandru, I have in a heap dump UnclosablePreparedStatement instances with ~90KB retained heap. But in at least a case, the SELECT is ~15k chars long. The main issue I see is for a column named thisisaverylongcolumnname, a thisisaverylongcolumnname170_0_ alias is created, too, which doubles the size (and there are cases with 250 or more fields, and this will explode even more if we move to denormalized extent). More, the SQL is kept at org.h2.jdbc.JdbcPreparedStatement.sqlStatement, CommandContainer.sql, org.h2.command.dml.Select.sqlStatement, so we have 3 places where heap is consumed for the SQL.

Do you think that an "SQL statement compressing technique" should be implemented to avoid caching long and relatively invaluable column names? This can also boost parsing time:

- Quantify column names without as: select tt.f1 as column1 to select tt.f1 "column1"
- Make order by clause use positional identifiers where possible: order by tt.f1 asc to order by 1 asc
- Consider shorter names for "reserved" fields: select tt._multiplex, tt._peerRowid, tt._errorFlag, tt._datasourceRowid to select tt._m, tt._peer, tt._error, tt.ds.
- Use column aliases in clauses : select tt.f1 as f1, tt.f2 as f2 from tt where tt.f2 = 1 to select tt.f1, tt.f2 as f2 from tt where f2 = 1. Also, consider quantifying unused columns with really short names (_c1, _c2, etc.). I think this is already in place, but something like column16_0_ is used.
- As a risky statement, the "*" wildcard can be used in many situations to reduce SQL size. However, I guess the field order is not necessarily consistent to our internal representation. Anyways, on a common case, we can emit "*" when the schema of a table wasn't altered from the creation and the H2 engine is also returning the columns in the order of their definition in the create statement. As a side note, H2 also allows a EXCLUDE statement for the wildcard.

There are few places where the meta-data column names of a result set are used. Therefore, this "compression" can be conditional. Unfortunately, most of my suggestion may provide 10% compression rate which may not be sufficient. The wildcard technique however may show up to a 50% compression rate.

#15 - 01/04/2023 05:28 AM - Constantin Asofiei

Alexandru, I think a first pass would be to change the FQL generation to use c1, c2, etc as aliases instead of the actual column name, like select tt.f1 as c1, tt.f2 as c2, ... from tt where c1 = 1 and c2 = 2 etc. And maybe create aliases only for columns which are part of SELECT or ORDER BY clauses. This will both keep the column order and reduce the length of the SQL.

More, this works 'out of the box' only if a single table is being used in the query - if you have joins or sub-selects, then the alias would need to be prefixed with the table name, like _t1_c1 (you can't prefix it with the table name, as the same table may be referenced twice or more in the same SQL).

Another issue would be: can H2 intern these SQL strings? The downside is they will not be garbage collected, and I don't know how Java will behave if the constant pool where these interned strings are kept gets very large.

> Consider shorter names for "reserved" fields:

I don't want to rename the reserved fields, keep them as is.

> Make order by clause use positional identifiers where possible:

If this can be done without introducing more overhead on FWD side, it may help.

#16 - 01/16/2023 08:03 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Alexandru, I think a first pass would be to change the FQL generation to use c1, c2, etc as aliases instead of the actual column name, like select tt.f1 as c1, tt.f2 as c2, ... from tt where c1 = 1 and c2 = 2 etc. And maybe create aliases only for columns which are part of SELECT or ORDER BY clauses. This will both keep the column order and reduce the length of the SQL.

Done this; planning to do a bit of time/memory check of this modification before committing. I looked a bit into the FqlToSqlConverter class and I see that these column aliases are used only in the select clause, but there seems to be of no use in the where or order by clauses. I may have overlooked their purpose, but I can't log any SQL with the generated aliases in use. Any hints on why we actually do this column aliases?

#17 - 01/17/2023 04:52 AM - Alexandru Lungu

> Alexandru, I have in a heap dump UnclosablePreparedStatement instances with ~90KB retained heap. But in at least a case, the SELECT is ~15k chars long

I've done some memory analysis for such case. With the compression, an UnclosablePreparedStatement with ~90KB retained heap for a SELECT of ~15k chars long was reduced (using shorter alias names) to ~85 KB retained heap with a SELECT of ~10k chars long. 30% compression rate is good, but the internal representation of such statement in H2 is still huge.

Out of a heap dump, using a dominator tree, I've got an org.h2.command.dml.Select instance with:

- an SQL using **20KB**
- the WHERE clause split into expression representations. For example, an instance of org.h2.expression.condition.ConditionAndOr retains **438B**
- the ORDER BY clause hold by an org.h2.result.SortOrder instance that retains **391B**
- a lot of org.h2.expression.Alias (~250 instances). Each of these uses between **104B** and **280B**, so lets say they roughly represent **50KB** in total. This is a problem! One such instance stores the alias (now in a compressed form like C243_) and the ExpressionColumn. Such expressions, from FWD generated SQLs, are usually column identifiers and can be strings of 10-30 characters.
- up to **10KB** of internal instances: ArrayList, LocalResultImpl, TableFilter, etc.

> Another issue would be: can H2 intern these SQL strings? The downside is they will not be garbage collected, and I don't know how Java will behave if the constant pool where these interned strings are kept gets very large.

Indeed, most of the memory wasted by a statement is on storing the column expressions, column names and **aliases**. In our case, interning the aliases may be a big deal as we use the same aliases again and again (C0_, C1_, C2_, etc.) for all query statements. Maybe ~**10KB** can be saved this way, for the statement I mentioned above.
For the column names, I guess that they are already stored inside the H2 engine to identify columns for a considerable amount of time. I agree that after a table drop, the column names (if interned) may lock the memory. On the other hand I suspect that there is a high probability in FWD to access the same columns again from different parts of the program, with different multiplex and even from different sessions.

#18 - 01/17/2023 05:58 AM - Alexandru Lungu

Alexandru Lungu wrote:

> - As a risky statement, the "*" wildcard can be used in many situations to reduce SQL size. However, I guess the field order is not necessarily consistent to our internal representation. Anyways, on a common case, we can emit "*" when the schema of a table wasn't altered from the creation and the H2 engine is also returning the columns in the order of their definition in the create statement. As a side note, H2 also allows a EXCLUDE statement for the wildcard.

Pardon the repetition, but I continued thinking of this "*" wildcard only for the H2 case. Considering the previous case, where ~50KB are used only for the select expressions inside the statement and maybe >10KB used for the column names and aliases inside the SQL string, we shall seriously think of integrating "*", at least for single-table queries. Eventually, we can work our way into the H2 engine to ensure that the columns are returned in a deterministic way to FWD. What do you think?

#19 - 01/17/2023 05:53 PM - Ovidiu Maxiniuc

We cannot rely on H2 or any other SQL server to provide the right order of columns for our properties. There are a lot of additional "hidden" columns specific to _temp tables, computed (generated columns,) multi-column properties and I have just added a new method (expand) for storing the extent fields. In the case of permanent table, some customers want direct access to database, I think, at some time, they will think of adding new columns to our tables.

I reviewed recently the FqlToSqlConverter and it seems we could drop the column aliases when the SQL is built. There is this FqlToSqlConverter.generateUniqueSqlColumnNames flag which is momentarily always set to false in FqlToSqlConverter's c'tor. This is the reason why the selected columns are always unique aliased. Although this was always the case, you may want to set it to false and redo the above tests to see what is the effect on H2 memory. There might be some issues with converted SQL code or even incorrectly generated code. If you encounter these issues and cannot easily fix, please post the issue here. As a side not, IIRC, Igor is working on other issue which apparently depends on this flag to remain on. In case your results are promising, we will discuss the specific matter.

#20 - 01/18/2023 06:19 AM - Alexandru Lungu

Ovidiu Maxiniuc wrote:

> We cannot rely on H2 or any other SQL server to provide the right order of columns for our properties. There are a lot of additional "hidden" columns specific to _temp tables, computed (generated columns,) multi-column properties and I have just added a new method (expand) for storing the extent fields. In the case of permanent table, some customers want direct access to database, I think, at some time, they will think of adding new columns to our tables.

I agree that on a general case, this is not feasible. However, I have many examples of very basic queries which are executed in large customer applications (single record find or filtered records selection)

```
select [~3000 characters of quantified column names and aliases] from [~20 characters of table name and alias]
 where [~100 characters of where clause] order by [~100 characters of order by]
```

That is, 90% of the SQL string are the column names (and usually this is the percentage). Even more, some simple, yet very used temp-tables, do not

have these "hidden" columns:

- Tables with DTZ_OFFSET shouldn't be retrieved with "table.*"
- Tables with denormalized extents shouldn't be retrieved with "table.*"
- Tables with both uppercase and lowercase character fields (due to word indexes) shouldn't be retrieved with "table.*"
- Maybe other cases too?

Unfortunately, I don't have a statistic of how many of these structures are structurally identical to their SQL table.

For the safety part, we can do this dialect-wise and allow it only for H2. As we have access to the H2 back-end, **we can ensure that the order of the columns is compatible with what we want in FWD**.
We can go the extra mile and make use of table.* EXCLUDE [hidden columns] as this clause should be shorter anyways. Ideally maybe is to add a column level keyword to suggest that we don't want to select that column with the wildcard marker. Something like CREATE LOCAL TEMPORARY TABLE tt (f1 INT HIDDEN) - much like we want to do with supporting NO-UNDO at the H2 level.

> I reviewed recently the FqlToSqlConverter and it seems we could drop the column aliases when the SQL is built. There is this FqlToSqlConverter.generateUniqueSqlColumnNames flag which is momentarily always set to false in FqlToSqlConverter's c'tor. This is the reason why the selected columns are always unique aliased. Although this was always the case, you may want to set it to false and redo the above tests to see what is the effect on H2 memory. There might be some issues with converted SQL code or even incorrectly generated code. If you encounter these issues and cannot easily fix, please post the issue here. As a side not, IIRC, Igor is working on other issue which apparently depends on this flag to remain on. In case your results are promising, we will discuss the specific matter.

I will test this right now.

#21 - 01/18/2023 07:36 AM - Alexandru Lungu

I've analyzed a heap dump without any alias generated for the columns. I got an ~8k chars long SQL string stored in a ~63KB UnclosablePreparedStatement. Comparing to #6829-17, I've got:

- an SQL using **16KB**
- the WHERE clause and ORDER BY clauses use the same memory as in #6829-17
- the org.h2.expression.Alias (~250 instances) are now simply small org.h2.expression.ExpressionColumn instances. Each of these uses between **70B and 186B**, so lets say they roughly represent **32KB** in total.
- up to 10KB of internal instances: ArrayList, LocalResultImpl, TableFilter, etc.

I didn't encountered any obvious problems after removing the column aliases. The conclusion is that by removing the alias completely, I removed ~**27KB** redundant memory, for the query I analyze. This is exactly 30%.

UPDATE: Previously, the TempTableDataSoureProvider instances over a set of tests retained around **15MB**. Without column aliases, they retain **12MB** (only 20% total memory reduction).

#22 - 01/18/2023 04:29 PM - Ovidiu Maxiniuc

Alexandru Lungu wrote:

> I agree that on a general case, this is not feasible. However, I have many examples of very basic queries which are executed in large customer applications (single record find or filtered records selection)
> That is, 90% of the SQL string are the column names (and usually this is the percentage). Even more, some simple, yet very used temp-tables, do not have these "hidden" columns:
>
> - Tables with DTZ_OFFSET shouldn't be retrieved with "table.*"
> - Tables with denormalized extents shouldn't be retrieved with "table.*"
> - Tables with both uppercase and lowercase character fields (due to word indexes) shouldn't be retrieved with "table.*"
> - Maybe other cases too?
>   Unfortunately, I don't have a statistic of how many of these structures are structurally identical to their SQL table.

You are right. However, we CAN do an inspection before the FQL->SQL conversion and, if there are none of these special columns, the * could be safely used. At the start of FqlToSqlConverter.expandAlias, a quick iteration of dmoMeta.propsByName in which we test for computed columns and normalized extent fields should be enough for veto. Not encountering these, the rest of the columns should map to our property list and order (!).

Alexandru Lungu wrote:

> I didn't encountered any obvious problems after removing the column aliases. The conclusion is that by removing the alias completely, I removed ~**27KB** redundant memory, for the query I analyze. This is exactly 30%.
> *UPDATE*: Previously, the TempTableDataSoureProvider instances over a set of tests retained around **15MB**. Without column aliases, they retain **12MB** (only 20% total memory reduction).

These figures are good, IMO. Also I have one more idea in this regard, although the win will not be as much. Currently the FqlToSqlConverter generates a "pretty" SQL using indentations and eoln. While this is useful while developing, in production is not required. So, we could generate the SQLs based on a prettify flag: if off, the full SQL will be one-liner, with single-space separators. I do not expect more than 3-5% memory gain in this case.

#23 - 01/18/2023 05:13 PM - Eric Faulhaber

I applaud efforts to make the SQL more compact and thereby reduce the memory consumption, and I agree we should pursue this. But I have two questions on this point I'd like to understand better:

- Do we have any notion yet on what practical impact these improvements will have on performance (as in, time spent / CPU consumed)? I understand that less heap, less GC are good, but do we have any sense of a quantifiable improvement?
- Is the premise of this work that we will have a prepared statement cache in the size range 64K max elements? Even if we cut the size of a single entry down considerably (say, by 80%), is it scalable when a single instance of this cache can reach ~50MB in heap, considering this is just one of many caches in one of many contexts? Constantin, did you notice any measurable performance gain when bumping up the size of the statement cache to 64K? If not, are you assuming you did not because of all of additional the heap in use?

#24 - 01/18/2023 05:29 PM - Eric Faulhaber

Note that we also use H2's query cache (the default size is 8; we set it to 1024 in H2Helper.setCommonInMemoryProperties), which as I understand it, caches query results. However, it seems these cached results must be tied to a prepared statement having a useful life cycle.

Has anyone who has looked inside H2 gotten a sense of how that cache interacts with our prepared statement cache? Are we getting any benefit from it; are we wasting memory there; are we recompiling the prepared statements (and thus invalidating cached query results) too aggressively?

I set this size to 1024 ages ago, without really investigating how well it was working for us, and that was before our implementation of prepared statement caching. I think we (or may just I) need to have a better understanding of how these multiple levels of caching are working together, if at all. Any insight based on recent reviews or debugging of H2 is welcome.

#25 - 01/19/2023 05:05 AM - Alexandru Lungu

> You are right. However, we CAN do an inspection before the FQL->SQL conversion and, if there are none of these special columns, the * could be safely used. At the start of FqlToSqlConverter.expandAlias, a quick iteration of dmoMeta.propsByName in which we test for computed columns and normalized extent fields should be enough for veto. Not encountering these, the rest of the columns should map to our property list and order (!).

I already have a patch for this exactly in expandAlias. I still miss some "hidden columns" cases, but I will definitely run some profile tests once it is stable. This can provide us a feedback what a ~**50KB** (per big queries) cut-down can look like.

> I applaud efforts to make the SQL more compact and thereby reduce the memory consumption, and I agree we should pursue this. But I have two questions on this point I'd like to understand better:
>
> - Do we have any notion yet on what practical impact these improvements will have on performance (as in, time spent / CPU consumed)? I understand that less heap, less GC are good, but do we have any sense of a quantifiable improvement?

I was planning to do such performance test while a lower memory consumption was reached. I will do a test without column aliases and one with wildcards (**maybe** ~**15KB** vs ~**63KB** vs ~**90KB** on the biggest prepared statement).
I expect to see an improvement due to less parsing effort from H2, better memory management (allocation and clear-up) and even faster cache look-up. If things go OK, I will consider bigger values for the cache (this is in regard to the next comment).

> - Is the premise of this work that we will have a prepared statement cache in the size range 64K max elements? Even if we cut the size of a single entry down considerably (say, by 80%), is it scalable when a single instance of this cache can reach ~50MB in heap, considering this is just one of many caches in one of many contexts? Constantin, did you notice any measurable performance gain when bumping up the size of the statement cache to 64K? If not, are you assuming you did not because of all of additional the heap in use?

Right now, I profiled 11 contexts using **2-7MB** each to a total of **15M**, each context storing a cache of at most 2048 statements. I agree that increasing the cache size will lead to a considerable memory consumption (per context), but we can find a middle-ground or make it customizable from the server settings. Anyways, this change can be done **only** if we decrease the SQL string.
AFAIK, the major problem here is that we still have a large customer application in which this cache has a rate of ~1%. There are several reasons:

- The cache is too small: this can be solved only if we lower the statement size and increase the cache size
- The statements are not that similar: this can be solved by #7035

- The statements are recompiled when metadata is changed (the name of this task)

  Has anyone who has looked inside H2 gotten a sense of how that cache interacts with our prepared statement cache? Are we getting any benefit from it; are we wasting memory there; are we recompiling the prepared statements (and thus invalidating cached query results) too aggressively?

Enqueuing this investigation.

#26 - 01/19/2023 08:02 AM - Greg Shah

> However, we CAN do an inspection before the FQL->SQL conversion and, if there are none of these special columns, the * could be safely used. At the start of FqlToSqlConverter.expandAlias, a quick iteration of dmoMeta.propsByName in which we test for computed columns and normalized extent fields should be enough for veto. Not encountering these, the rest of the columns should map to our property list and order (!).

If it saves any significant processing, we can calculate this at conversion time and set a flag.

#27 - 01/20/2023 10:46 AM - Alexandru Lungu

Greg Shah wrote:

> > However, we CAN do an inspection before the FQL->SQL conversion and, if there are none of these special columns, the * could be safely used. At the start of FqlToSqlConverter.expandAlias, a quick iteration of dmoMeta.propsByName in which we test for computed columns and normalized extent fields should be enough for veto. Not encountering these, the rest of the columns should map to our property list and order (!).
>
> If it saves any significant processing, we can calculate this at conversion time and set a flag.

Committed 6129b/rev. 14378. This includes no alias columns for all SELECT queries and wildcards for temp-tables without computed columns. Done only some time profiling by now. Planning to do some memory checks as well.

#28 - 01/20/2023 10:54 AM - Constantin Asofiei

Alexandru Lungu wrote:

> Greg Shah wrote:
>
> > However, we CAN do an inspection before the FQL->SQL conversion and, if there are none of these special columns, the * could be safely used. At the start of FqlToSqlConverter.expandAlias, a quick iteration of dmoMeta.propsByName in which we test for computed columns and normalized extent fields should be enough for veto. Not encountering these, the rest of the columns should map to our property list and order (!).
>
> If it saves any significant processing, we can calculate this at conversion time and set a flag.
>
> Committed 6129b/rev. 14378. This includes no alias columns for all SELECT queries and wildcards for temp-tables without computed columns. Done only some time profiling by now. Planning to do some memory checks as well.

We need to release a stable version to the customer today. I'm testing this revision now, but otherwise, please don't commit without further notice.

#29 - 01/20/2023 12:42 PM - Eric Faulhaber

Alexandru Lungu wrote:

> Committed 6129b/rev. 14378. This includes no alias columns for all SELECT queries and wildcards for temp-tables without computed columns. Done only some time profiling by now. Planning to do some memory checks as well.

How are joins managed with no aliases in the SELECT clause? Wouldn't we at least need aliases for those columns involved in a join (and in the WHERE clause). Or does this apply only to single-table queries?

#30 - 01/20/2023 12:57 PM - Constantin Asofiei

- File 6892_14378_uncommitted.patch added

Alexandru, I've uncommited 6129b rev 14378. The changes in this rev are in the attached patch.

#31 - 01/23/2023 04:14 AM - Alexandru Lungu

Eric Faulhaber wrote:

> Alexandru Lungu wrote:
>
> > Committed 6129b/rev. 14378. This includes no alias columns for all SELECT queries and wildcards for temp-tables without computed columns. Done only some time profiling by now. Planning to do some memory checks as well.
>
> How are joins managed with no aliases in the SELECT clause? Wouldn't we at least need aliases for those columns involved in a join (and in the WHERE clause). Or does this apply only to single-table queries?

I extracted a generated SQL after the modification:

```
select tt.*, tt2.*
from first_table tt
cross join second_table tt2
where tt._multiplex = ? and tt2._multiplex = ? and [...]
order by tt._multiplex asc, tt.recid asc, [...]
```

For a testcase where the same table is used in the joining process:

```
select tt_im0_.*, tt_im1_.*
from first_table tt_im0_
cross join second_table tt_im1_
where tt_im0_._multiplex = ? and tt_im1_._multiplex = ? [...]
order by tt_im0_._multiplex asc, tt.recid asc, [..,]
```

All fields inside the WHERE clause and ORDER BY clause are quantified and the table aliases are uniquely generated. I couldn't find obvious errors in large customer applications when hydration; the order looks fine.

> Alexandru, I've uncommited 6129b rev 14378. The changes in this rev are in the attached patch.

OK. Is 6129b in a freeze state right now?

#32 - 01/23/2023 04:35 AM - Alexandru Lungu

- Related to Feature #7060: Consider using wildcard selection to reduce SQL size added

#33 - 01/23/2023 04:38 AM - Alexandru Lungu

I created #7060 to target exactly the issue of wildcard usage, as a performance/memory management standalone improvement. This task should independently handle the recompilation of statements and adjacent topics from which we strayed in the last comments.

#34 - 01/23/2023 04:50 PM - Eric Faulhaber

Alexandru Lungu wrote:

> Constantin Asofiei wrote:
>
> > Alexandru, I've uncommited 6129b rev 14378. The changes in this rev are in the attached patch.
>
> OK. Is 6129b in a freeze state right now?

Alexandru, I don't think Constantin saw this question; I just added him as a watcher.

We were delivering 6129b to a customer Friday and I think he considered this a risky change to include. It is still a risky change, IMO, so I'm not sure of the best way to handle it, but we can pick up that discussion in #7060.

#35 - 01/30/2023 07:03 AM - Alexandru Lungu

- Assignee set to Dănuț Filimon

- Status changed from New to WIP

Danut, please strictly refer to #6829-1 for the moment: consider the H2 performance of reparsing prepared statements when changing metadata. I will guide you through the H2 internals if needed.

We have many side-issues here which can be addressed independently later on. I will only list them here for better reference.

- #6829-3, #6829-4: consider a better eviction policy for ps cache, to eagerly remove statements over dropped tables.
- #6829-8: possible bug with sessionUseCount
- #6829-10: general effort to increase ps cache size
- most of the comments starting with #6829-11: general effort to reduce the size of prepared statements (#7060 is related)

#36 - 02/07/2023 06:44 AM - Dănuț Filimon

Committed 6829a_h2/rev.7. Stopped forced re-parsing for prepared statements.

Each SELECT/DELETE/UPDATE/INSERT query has a collectDependencies method which returns the tables that make up that query as DbObject instances. For DELETE/UPDATE/INSERT, the main table would not be included in the dependencies so I had to add it. The collectDependencies method is called for some action related to MVTables, which we do not use. By checking each dependency modification id and comparing it to the prepared statement modification id and if it's greater, recompile the statement. This change greatly reduces the amount of recompiled statements. I am currently testing a customer application and will make an update with results if necessary.

#37 - 02/08/2023 10:09 AM - Alexandru Lungu

Dănuț Filimon wrote:

> Committed 6829a_h2/rev.7. Stopped forced re-parsing for prepared statements.
>
> Each SELECT/DELETE/UPDATE/INSERT query has a collectDependencies method which returns the tables that make up that query as DbObject instances. For DELETE/UPDATE/INSERT, the main table would not be included in the dependencies so I had to add it. The collectDependencies method is called for some action related to MVTables, which we do not use. By checking each dependency modification id and comparing it to the prepared statement modification id and if it's greater, recompile the statement. This change greatly reduces the amount of recompiled statements. I am currently testing a customer application and will make an update with results if necessary.

I profiled this and couldn't notice a consistent improvement (± 0.1% of total execution time). I've done some debugging and I see that ~10ms is spent in the server start-up and ~20ms at run-time for collecting dependencies (in a cold run with a total execution time of ~12,000 ms).

Good news is that needsRecompile returns true very rarely. Only the following prepared statement is recompiled in my testcase:

```
DELETE FROM meta_myconnection_ WHERE recid = ? AND my_conn_user_id = fwdSession_1()
```

Therefore, the total time spent in checking the need of recompiling is in fact that ~30ms from collecting dependencies. **I wonder if this is an acceptable over-head comparing to the whole recompiling process.**

At this moment, dropping the recompiling may cause problems. We should eagerly remove all prepared statements using dropped objects if we want to ensure that "no recompiling" is safe. Note that this is not only about temp-tables, but also about meta tables, sequences, intermediate objects (like FastCopyHelper.MASTER__PK__MAPPING), etc. At this moment, due to the possible sessionUseCount issue from #6829-8, we don't ever drop temp-tables.

#38 - 02/08/2023 10:56 AM - Alexandru Lungu

Eric Faulhaber wrote:

> Has anyone who has looked inside H2 gotten a sense of how that cache interacts with our prepared statement cache? Are we getting any benefit from it; are we wasting memory there; are we recompiling the prepared statements (and thus invalidating cached query results) too aggressively?

org.h2.engine.Session.prepareLocal is mostly the same as our psCache logic: look-up for an existing prepared version of the statement in the cache and return it, otherwise do the preparing and cache. H2 is also using a LRU cache.

Debugging a large application, I see that our cache entirely dominates the H2 query cache. The only hits are on the _meta database, for which psCache is not configured. The issue of the H2 cache is that it is cleared at each metaModificationId change, so after a database object create/drop. ps cache is never cleared; the statements are recompiled only when/if needed (due to #6829-36). As we do a lot of H2 meta changes (creating and dropping tables/indexes, etc.), we should consider dropping the H2 query cache for _temp database. We can leave it only for _meta H2 database.

I will profile FWD without the QUERY_CACHE_SIZE setting for _temp database.

#39 - 02/14/2023 08:12 AM - Alexandru Lungu

- % Done changed from 0 to 80

Dănuț Filimon wrote:

> Committed 6829a_h2/rev.7. Stopped forced re-parsing for prepared statements.
>
> Each SELECT/DELETE/UPDATE/INSERT query has a collectDependencies method which returns the tables that make up that query as DbObject instances. For DELETE/UPDATE/INSERT, the main table would not be included in the dependencies so I had to add it. The collectDependencies method is called for some action related to MVTables, which we do not use. By checking each dependency modification id and comparing it to the prepared statement modification id and if it's greater, recompile the statement. This change greatly reduces the amount of recompiled statements. I am currently testing a customer application and will make an update with results if necessary.

I am planning to merge 6829a_h2 into FWD-H2 trunk ASAP. I **reviewed** the changes and they are alright. Even if we still need a "collectDependencies" phase to understand if we need to invalidate or not, I think it is acceptable comparing to the total overhead of actually reparsing and replanning.

However, I rebased 6829a_h2 with trunk and got some failing regression tests. One example is:

```
with recursive t(n) as (select 1 union all select n+1 from t where n<3) select * from t where n>?
```

Using this as a prepared statement and executing it twice results into a NPE. The second execution presumes that the "recursive" view is still available, even if it was implicitly removed before-hand. Its modificationId is the same as prepared statement's modificationId, because the view is implicitly created and removed in the same "database version" as the statement execution.

Because of this flaw, your changes considers that view's modificationId matches databases's modificationId, so it avoids recompiling. At this point, the same stale view is used.

I've done a small change to the database modificationId generation that does the trick - now all regression tests pass. I've committed it as 6829a_h2/rev. 11. **Danut, please run your test-cases for #6829 against this revision and make sure that the change doesn't reintroduce aggresive recompiling. Also, please recheck the ps cache hit/miss and profile the time.** If this final version still avoids doing useless recompiling

and shows a good performance improvement, I will merge it as it is right now.

#40 - 02/15/2023 03:35 AM - Dănuț Filimon

There are no problems with rev.11. I checked psCache hit/miss with rev.6 and rev.11 and didn't get very different results. And I also tested rev.9 and rev.11 and found an improvement in execution time. The change should be good to go.

#41 - 02/15/2023 03:59 AM - Alexandru Lungu

- % Done changed from 80 to 100

- Status changed from WIP to Review

Merged 6829a_h2 to FWD-H2 trunk including performance improvements regarding less recompiling of prepared statements due to meta modifications.

I will take a look into #6829-35 to identify other points of interest and create separate tasks for those that still represent a problem in the latest 6129c.

#42 - 01/19/2024 01:22 AM - Eric Faulhaber

Can this task be closed?

#43 - 01/19/2024 02:14 AM - Alexandru Lungu

- Status changed from Review to Internal Test

Recently, we done some changes that prevented us from using a "NEVER_RECOMPILE" kind of option. Back in #6829-1, it was not possible because we could have stale prepared statements in the cache. Now that we optimized fast-copy to instantiate the intermediary tables only once, I think "NEVER_RECOMPILE" is feasible.

Danut, please analyze performance with NEVER_RECOMPILE. I don't recall if we have something like that in FWD-H2 or not - I remember you had such option at some point. If there are no regressions and you see a performance improvement, lets go ahead with the change.

#44 - 01/19/2024 07:51 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Danut, please analyze performance with NEVER_RECOMPILE. I don't recall if we have something like that in FWD-H2 or not - I remember you had such option at some point. If there are no regressions and you see a performance improvement, lets go ahead with the change.

I've obtained really good results when using NEVER_RECOMPILE option, ~**1.25%** improvement for the **average of the last 20 runs** and ~**1.35%** for the **total average**. The results are good, should I go ahead and do more testing (e.g. smoke test another customer application, POC regression)?

Just as a note, NEVER_RECOMPILE database property was introduced in FWD-H2 revision 20 (ref: #7351).

#45 - 01/19/2024 07:58 AM - Alexandru Lungu

These are great news. Please do extensive testing: focus on smoke testing another customer application and regression tests.

#46 - 01/22/2024 05:49 AM - Dănuț Filimon

I ran POC regression tests, smoke tested a large customer application and also ran ChUI regression tests. There are no issues with the NEVER_RECOMPILE option.

#47 - 01/22/2024 05:59 AM - Alexandru Lungu

Lets go ahead with the change. Please create a FWD branch and commit the change only for temporary H2 database.

#48 - 01/22/2024 06:57 AM - Dănuț Filimon

Alexandru Lungu wrote:

> Lets go ahead with the change. Please create a FWD branch and commit the change only for temporary H2 database.

Created 6829a and committed revision **14936**.

#49 - 01/22/2024 08:44 AM - Alexandru Lungu

I am OK with the changes. As long as they are well-tested, we can do the merge.

#50 - 01/22/2024 08:46 AM - Greg Shah

It can be merged quickly if the testing is complete.

#51 - 01/22/2024 08:52 AM - Dănuț Filimon

I am merging 6829a right now.

#52 - 01/22/2024 08:54 AM - Greg Shah

- Status changed from Internal Test to Merge Pending

OK

#53 - 01/22/2024 09:05 AM - Dănuț Filimon

- Status changed from Merge Pending to Test

6829a was merged intro trunk as revision **14939** and archived.

#54 - 01/23/2024 04:44 PM - Eric Faulhaber

- Status changed from Test to Closed

There is no customer-specific testing for this improvement; any bugs will be dealt with separately. Accordingly, I'm closing this.

Files

6892_14378_uncommitted.patch	9.19 KB	01/20/2023	Constantin Asofiei