# Database - Bug #8214

## Split Persister into SQLPersister and InMemoryPersister

01/24/2024 02:12 AM - Alexandru Lungu

| | | | |
|---|---|---|---|
| **Status:** | WIP | **Start date:** | |
| **Priority:** | Normal | **Due date:** | |
| **Assignee:** | Alexandru Lungu | **% Done:** | 50% |
| **Category:** | | **Estimated time:** | 0.00 hour |
| **Target version:** | | | |
| **billable:** | No | **case_num:** | |
| **vendor_id:** | GCD | | |

**Description**

### History

#### #1 - 01/24/2024 02:22 AM - Alexandru Lungu

*- Assignee set to Alexandru Lungu*

I constantly see Proxy.set... in the top bottlenecks we have in the FWD API. This is not because they are really slow, but because they are a lot, especially in the _temp table.

We did good efforts recently in optimizing Update inside H2, but we still need to pay whole overhead of record finding, trigger check, constraint check, lock check, transactions, SQL caching, preparing, executing and closing, etc.

I suggest having a more straight-forward to persist temporary records using direct-access. This will take effect only for NO-UNDO tables. The direct-access will receive the record id, column to be updated and the new value. It will simply extract the row from the MultiplexedScanIndex based on the id, do the in-memory row update, and rebuild all indexes if the column is indexed (**that is all**).

To do this, I am thinking of making Persister an interface, keep the current implementation as SQLPersister and create a new InMemoryPersister only for sessions over the temporary database. This will serve as a proxy for a SQLPersister over the _temp database, to handle UNDO tables + inserts and deletes which will be implemented in a second phase.

#### #2 - 01/24/2024 09:12 AM - Greg Shah

Long ago, the 4GL was changed to implement the ASSIGN statement.  One important reason (maybe **the** important reason) that they did this was to batch the edits to records.  This was found to be much faster than individual assignment statements.  It seems to me that the batching limits the checking overhead.

We have our own implementation of this batching, but I wonder if we have fully taken advantage of the things we don't have to do.  This is probaby at a higher level than what you are discussing here, but it seems connected.  If we can assign multiple values without having to honor a full set of extra checks (flushing, triggers... whatever), perhaps there is a win here to be had.

#### #3 - 01/24/2024 10:07 AM - Alexandru Lungu

I agree with you Greg, .batch itself is a very useful construct if you want to update a single record on multiple fields.

However, I've seen (in the profilings I did) way more patterns of type:

```
for each tt [...] NO-LOCK:
tt.f1 = [...].
end.
```

This can happen:

- in UI work, as the dynamic widgets of a frame are tracked using a temp-table. When updating something (the visibility, background color, a random flag, etc.), the pattern above is employed. This was found while working on #6355.
- in some cases, the temp-table is used a "collection" with fast query and update. In FWD, this was transformed in a heavy-weight pattern employing locking, transactions, SQLs, caching, etc. This was found when doing #7026, but not particularized.

I have the H2 changes already for this prototype. I am doing the FWD changes now and attempt some profiling tests.

#### #4 - 01/25/2024 06:39 AM - Alexandru Lungu

*- Status changed from New to WIP*

*- % Done changed from 0 to 50*

Comitted 8214a_h2 the changes in FWD-H2 to do update on a single column using direct-access.
I also have the FWD changes, but need to format them them before committing.

The ratio is really good according to my JMXs: 80%-90% of **all** updates on _temp are resolved using direct-access (on a large application performance tests). This looks promising (and quite risk-free).
I am finishing the FWD changes, commit, check them out on my bare-metal and do a round of performance tests.

#### #5 - 01/25/2024 01:23 PM - Eric Faulhaber

In the absence of the force-no-undo-temp-tables configuration option being set to true, any transaction on the _temp database can contain a mixture of undoable and NO-UNDO temp-table use. How straightforward is it on the H2 side, for a given transaction, to detect that there are exclusively NO-UNDO modifications? For such transactions, can we completely avoid the overhead associated with transaction processing (e.g., full transaction and savepoint commits/rollbacks)?

Or is it more of a linear reduction in overhead? That is, for example, would a transaction with 25% undoable operations and 75% NO-UNDO operations (assuming functionally equivalent operations), roughly realize a 75% reduction in transaction/savepoint processing overhead with these changes? Or was such an improvement already realized when support for "native" NO-UNDO temp-tables was added to FWD-H2?

On the FWD side, we could detect whether savepoints are necessary at all for the _temp database for a given transaction, depending on whether an undoable temp-table was modified in that transaction. Do we do this already? Similarly, do we avoid transactions and savepoints for the _temp database altogether when force-no-undo-temp-tables is true? I don't recall whether I initially implemented it that way, but anyway my original implementation has changed significantly over time.

#### #6 - 01/25/2024 03:36 PM - Alexandru Lungu

Eric Faulhaber wrote:

> In the absence of the force-no-undo-temp-tables configuration option being set to true, any transaction on the _temp database can contain a mixture of undoable and NO-UNDO temp-table use. How straightforward is it on the H2 side, for a given transaction, to detect that there are exclusively NO-UNDO modifications? For such transactions, can we completely avoid the overhead associated with transaction processing (e.g., full transaction and savepoint commits/rollbacks)?

The PageStore model is quite simple: every change is logged in a changeLog and reflected in the PageStoreTable instances. If there is a rollback, the changeLog is iterated in reverse. The savepoint setting is just a marker in the changeLog. Rollbacking to a savepoint, means iterating the changeLog to that saved position. I find the savepoints and transactions in H2 (at least this embedded, in-memory, temporary model) quite light-weight.

We support NO-UNDO tables in H2. Such tables simply avoid logging in the changeLog, so basically they are "out-of-transaction".

> Or is it more of a linear reduction in overhead? That is, for example, would a transaction with 25% undoable operations and 75% NO-UNDO operations (assuming functionally equivalent operations), roughly realize a 75% reduction in transaction/savepoint processing overhead with these changes? Or was such an improvement already realized when support for "native" NO-UNDO temp-tables was added to FWD-H2?

I am not quite sure where the overhead is particularly. It is just concerning that we are doing "tons" of things for a simple column update. It is just that between JDBC and actual PageStoreTable, there are another 2 layers that may be (usually are) no-op, but are still traversed. Starting from connection closing check, to session closing check, to statement closing check. It is followed by parsing and planning of statement (or not). Followed by database synchronization (if any), transaction check, engine check (MvStore or PageStore). Followed by honoring the WHERE clause (finding the record by recid and multiplex). Followed by identifying the updated columns, triggers, constraints, user rights check, in-line ON UPDATE clause.

I am dramatic with the presentation above, but the point is that there is code executed just to change a value.

> On the FWD side, we could detect whether savepoints are necessary at all for the _temp database for a given transaction, depending on whether an undoable temp-table was modified in that transaction. Do we do this already? Similarly, do we avoid transactions and savepoints for the _temp database altogether when force-no-undo-temp-tables is true? I don't recall whether I initially implemented it that way, but anyway my original implementation has changed significantly over time.

Yes, if force-no-undo-temp-tables is set, there is no SavepointManager created. Also, SavepointManager acts (or should act) lazily, thus setting savepoints only when starting to change things (on undoable tables). The savepoint, transaction and commits on H2 "already hit the ground". My last hope here with #8214 is to eliminate the relation with these entirely - go "right to the core" with no-undo tables.

My changes are finished and quite well-tested. I am doing the profiling tomorrow morning.

#### #7 - 01/30/2024 11:12 AM - Alexandru Lungu

I attempted to test my changes on 8214a and 8214a_h2 but got no viable improvement. Interestingly, 80% of the updates actually hit the direct-access. I will pend the effort here.